7 Dictionary

Dictionary:

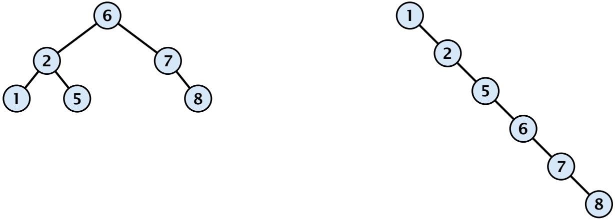
- S.**insert**(x): Insert an element x.
- S.**delete**(x): Delete the element pointed to by x.
- S.**search**(k): Return a pointer to an element e with $\text{key}[e] = k$ in S if it exists; otherwise return null.

7.1 Binary Search Trees

An (internal) binary search tree stores the elements in a binary tree. Each tree-node corresponds to an element. All elements in the left sub-tree of a node v have a smaller key-value than $\mathrm{key}[v]$ and elements in the right sub-tree have a larger-key value. We assume that all key-values are different.

(External Search Trees store objects only at leaf-vertices)

Examples:

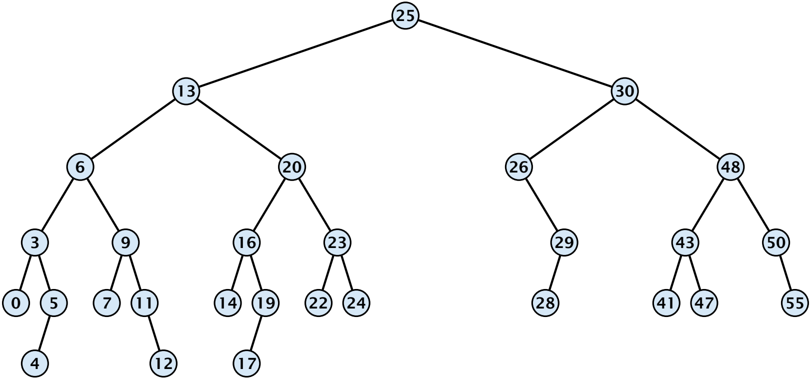


7.1 Binary Search Trees

We consider the following operations on binary search trees. Note that this is a super-set of the dictionary-operations.

- $T.\mathrm{insert}(x)$
- $T.\mathrm{delete}(x)$
- $T.\mathrm{search}(k)$
- $T.\mathrm{successor}(x)$
- $T.\mathrm{predecessor}(x)$
- $T.\mathrm{minimum}()$
- $T.\mathrm{maximum}()$

Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

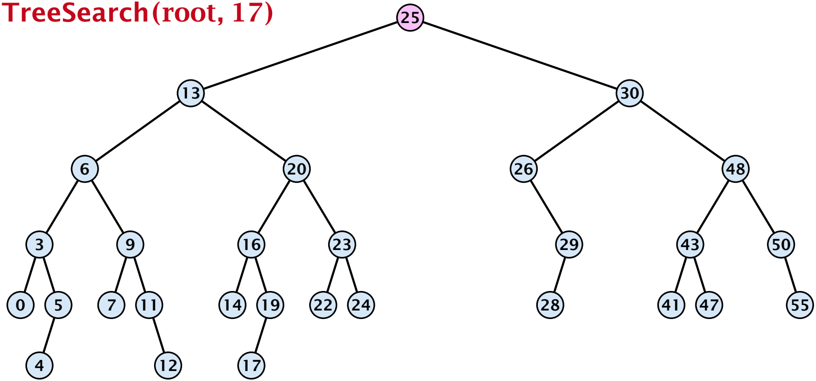
if x = null **or** k = key$[x]$ **return** x

if $k <$ key$[x]$ **return** TreeSearch(left$[x], k$)

else return TreeSearch(right$[x], k$)

Binary Search Trees: Searching

TreeSearch(root, 17)

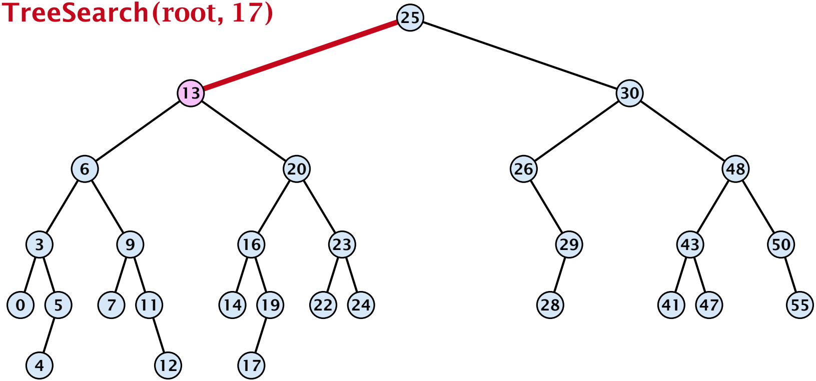


Algorithm 5 TreeSearch(x, k)

if x = null **or** k = key$[x]$ **return** x
if $k <$ key$[x]$ **return** TreeSearch(left$[x]$, k)
else return TreeSearch(right$[x]$, k)

Binary Search Trees: Searching

TreeSearch(root, 17)

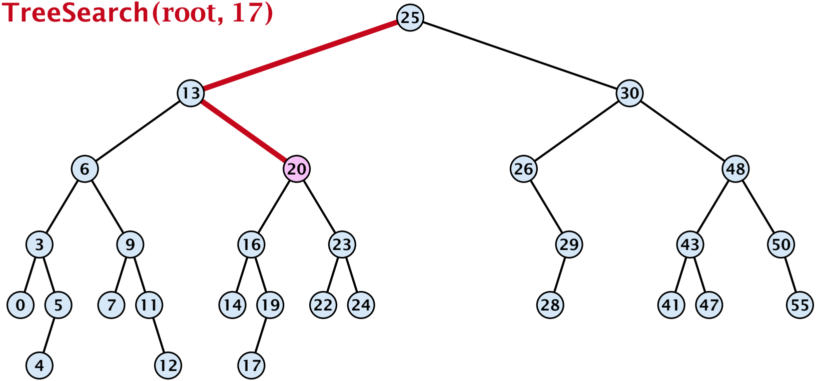


Algorithm 5 TreeSearch(x, k)

1: **if** x = null **or** k = key$[x]$ **return** x
2: **if** $k <$ key$[x]$ **return** TreeSearch(left$[x]$, k)
3: **else return** TreeSearch(right$[x]$, k)

Binary Search Trees: Searching

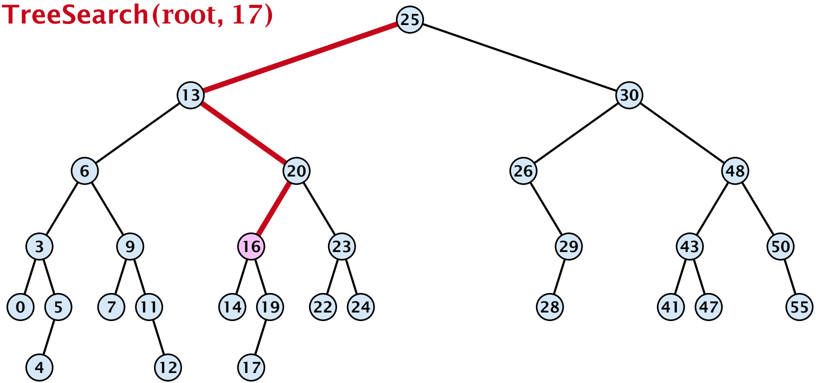
TreeSearch(root, 17)



Algorithm 5 TreeSearch(x, k)

if x = null **or** k = key$[x]$ **return** x
if $k <$ key$[x]$ **return** TreeSearch(left$[x], k$)
else return TreeSearch(right$[x], k$)

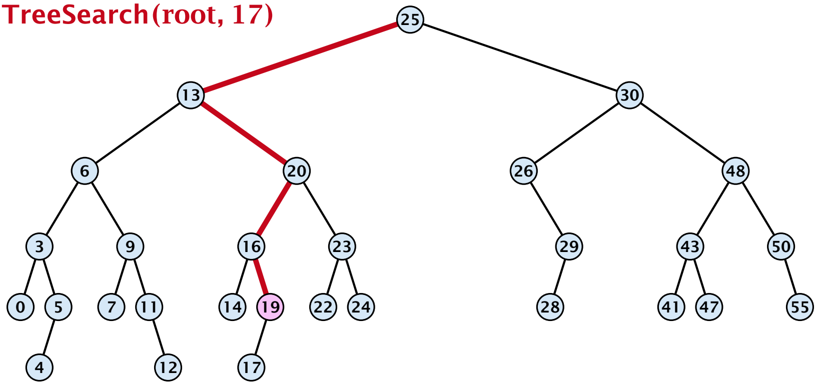
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

1: **if** $x = \text{null}$ **or** $k = \text{key}[x]$ **return** x
2: **if** $k < \text{key}[x]$ **return** TreeSearch$(\text{left}[x], k)$
3: **else return** TreeSearch$(\text{right}[x], k)$

Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

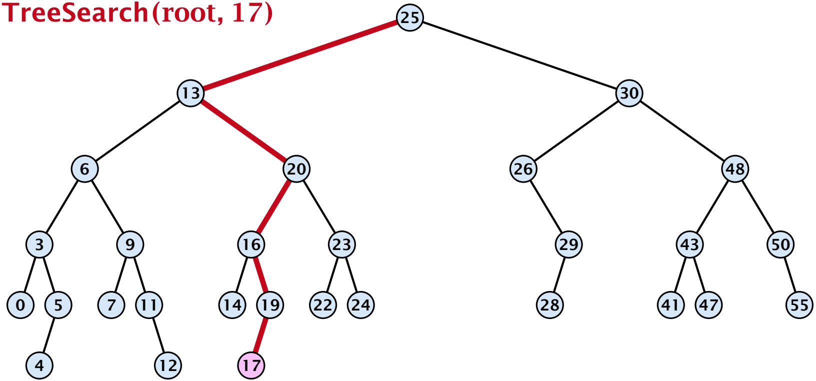
if x = null **or** $k = \text{key}[x]$ **return** x

if $k < \text{key}[x]$ **return** TreeSearch$(\text{left}[x], k)$

else return TreeSearch$(\text{right}[x], k)$

Binary Search Trees: Searching

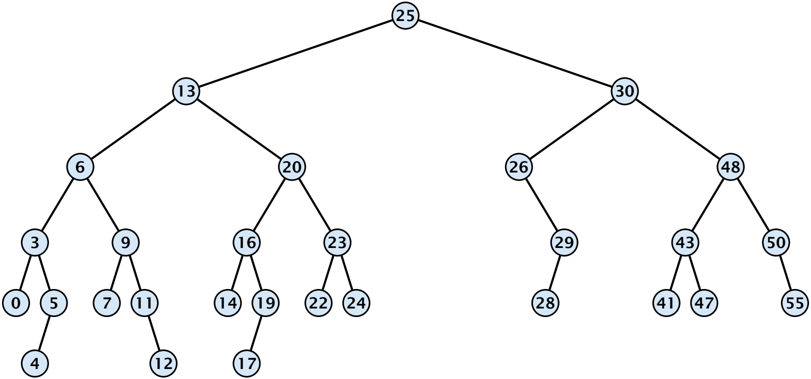
TreeSearch(root, 17)



Algorithm 5 TreeSearch(x, k)

1: **if** $x = \text{null}$ **or** $k = \text{key}[x]$ **return** x
2: **if** $k < \text{key}[x]$ **return** TreeSearch$(\text{left}[x], k)$
3: **else return** TreeSearch$(\text{right}[x], k)$

Binary Search Trees: Searching

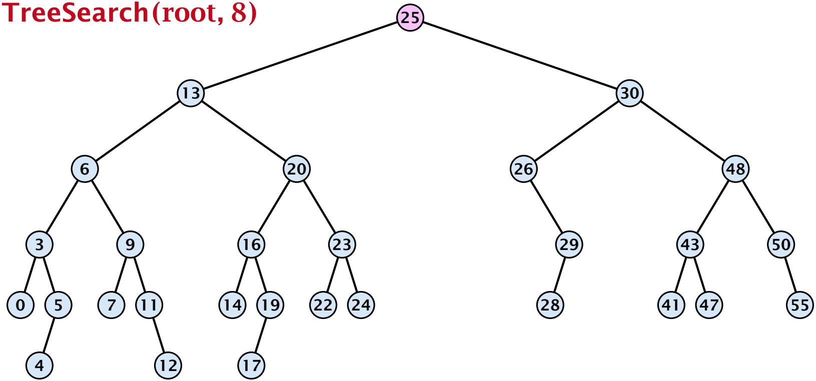


Algorithm 5 TreeSearch(x, k)

if x = null **or** k = key$[x]$ **return** x
if $k <$ key$[x]$ **return** TreeSearch(left$[x]$, k)
else return TreeSearch(right$[x]$, k)

Binary Search Trees: Searching

TreeSearch(root, 8)



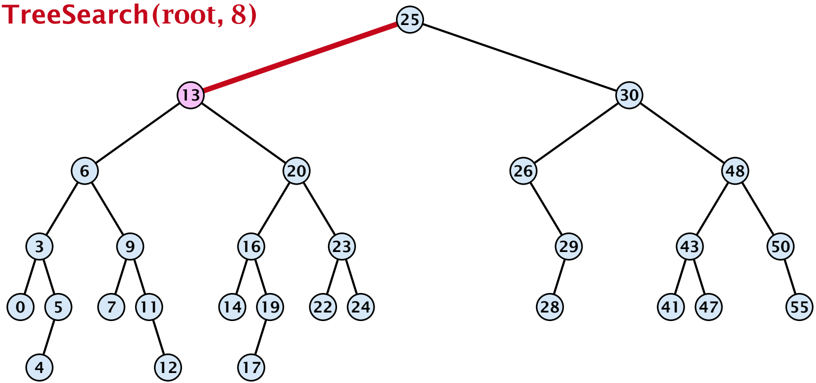
Algorithm 5 TreeSearch(x, k)

if x = null **or** k = key$[x]$ **return** x

if $k <$ key$[x]$ **return** TreeSearch(left$[x]$, k)

else return TreeSearch(right$[x]$, k)

Binary Search Trees: Searching

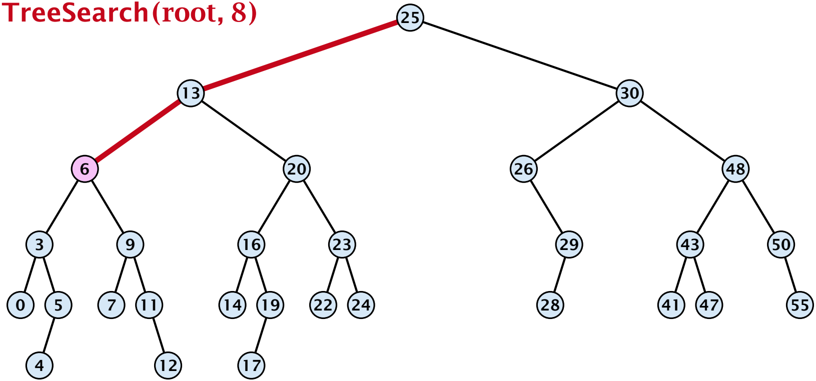


Algorithm 5 TreeSearch(x, k)

1: **if** x = null **or** k = key$[x]$ **return** x
2: **if** $k <$ key$[x]$ **return** TreeSearch(left$[x], k)$
3: **else return** TreeSearch(right$[x], k)$

Binary Search Trees: Searching

TreeSearch(root, 8)

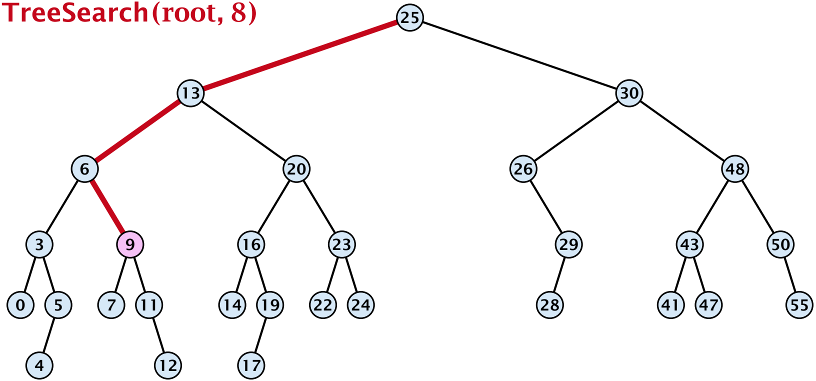


Algorithm 5 TreeSearch(x, k)

1: **if** x = null **or** $k = \text{key}[x]$ **return** x
2: **if** $k < \text{key}[x]$ **return** TreeSearch$(\text{left}[x], k)$
3: **else return** TreeSearch$(\text{right}[x], k)$

Binary Search Trees: Searching

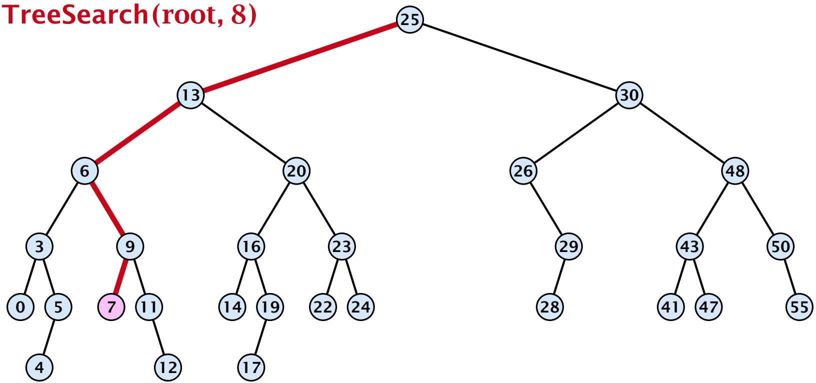
TreeSearch(root, 8)



Algorithm 5 TreeSearch(x, k)

if $x = \text{null}$ **or** $k = \text{key}[x]$ **return** x
if $k < \text{key}[x]$ **return** TreeSearch$(\text{left}[x], k)$
else return TreeSearch$(\text{right}[x], k)$

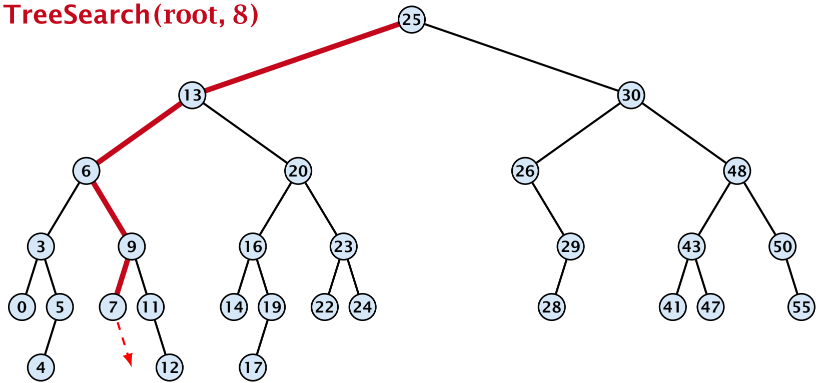
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

1: **if** x = null **or** k = key$[x]$ **return** x
2: **if** $k <$ key$[x]$ **return** TreeSearch(left$[x], k$)
3: **else return** TreeSearch(right$[x], k$)

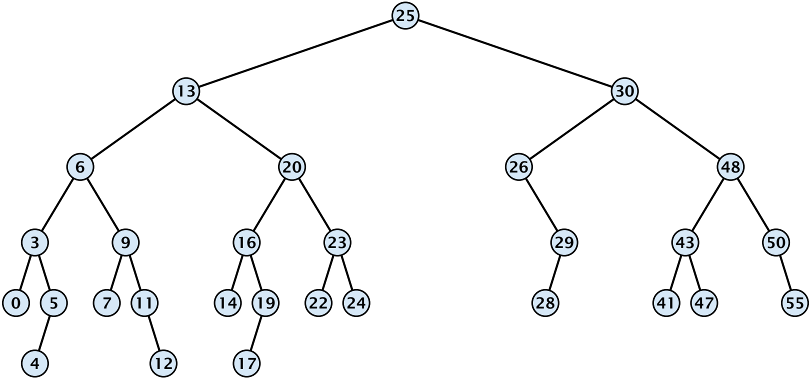
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

1: **if** $x = \text{null}$ **or** $k = \text{key}[x]$ **return** x
2: **if** $k < \text{key}[x]$ **return** TreeSearch$(\text{left}[x], k)$
3: **else return** TreeSearch$(\text{right}[x], k)$

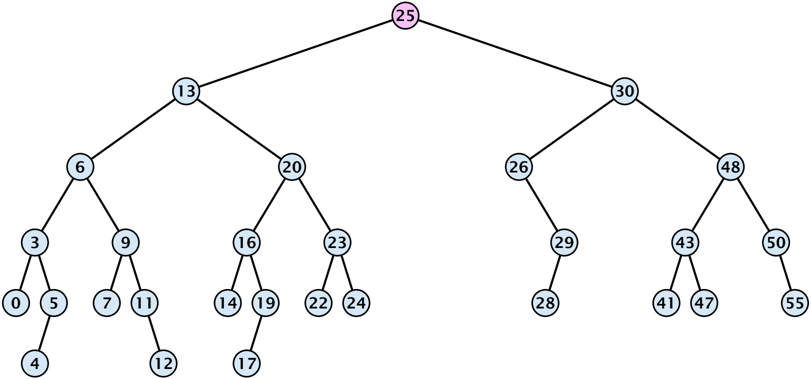
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

if x = null **or** left[x] = null **return** x
return TreeMin(left[x])

Binary Search Trees: Minimum

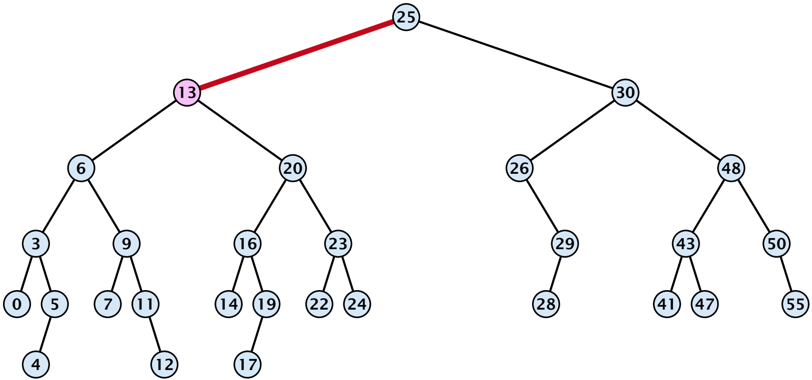


Algorithm 6 TreeMin(x)

1: **if** x = null **or** left[x] = null **return** x

2: **return** TreeMin(left[x])

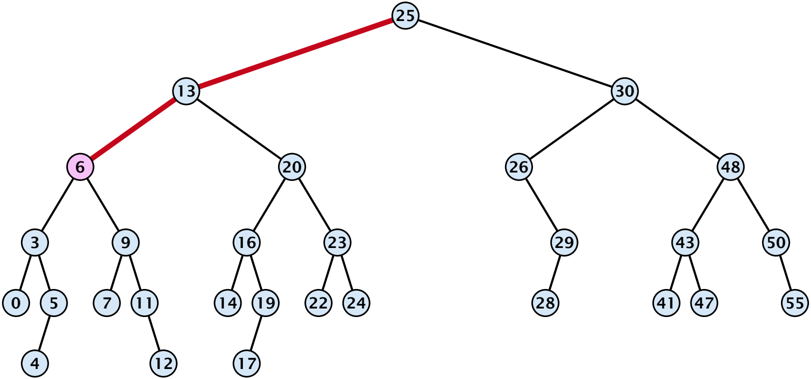
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

if x = null **or** left[x] = null **return** x
return TreeMin(left[x])

Binary Search Trees: Minimum

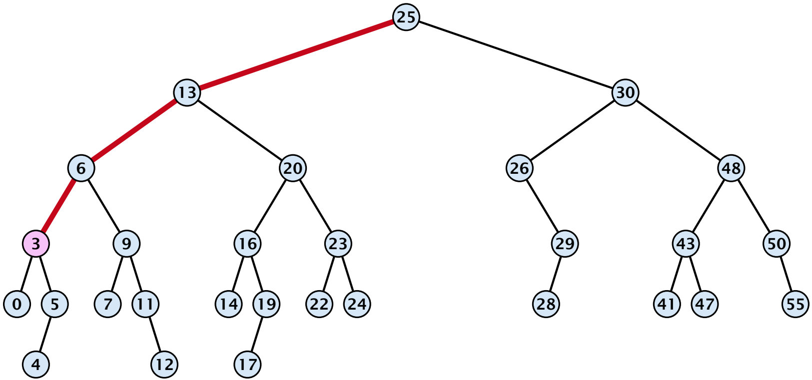


Algorithm 6 TreeMin(x)

1: **if** x = null **or** left$[x]$ = null **return** x

2: **return** TreeMin(left$[x]$)

Binary Search Trees: Minimum

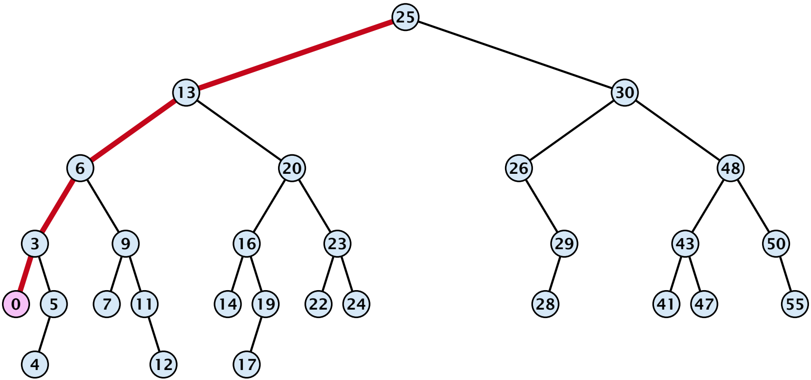


Algorithm 6 TreeMin(x)

1: **if** $x = \text{null}$ **or** $\text{left}[x] = \text{null}$ **return** x

2: **return** TreeMin($\text{left}[x]$)

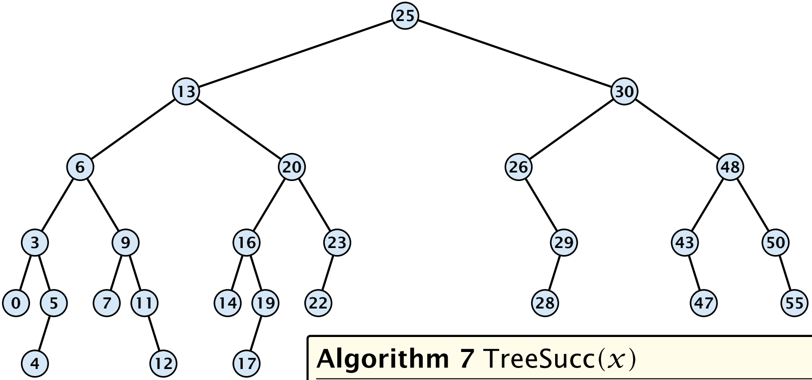
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

if x = null **or** left[x] = null **return** x
return TreeMin(left[x])

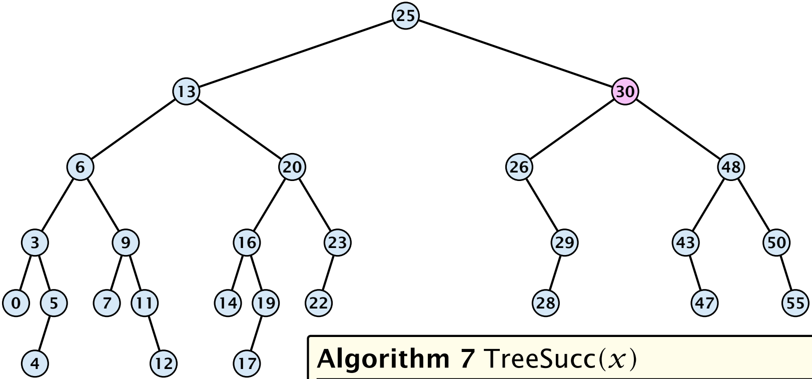
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

if right$[x] \neq$ null **return** TreeMin(right$[x]$)
$y \leftarrow$ parent$[x]$
while $y \neq$ null **and** $x =$ right$[y]$ **do**
    $x \leftarrow y; y \leftarrow$ parent$[x]$
return y;

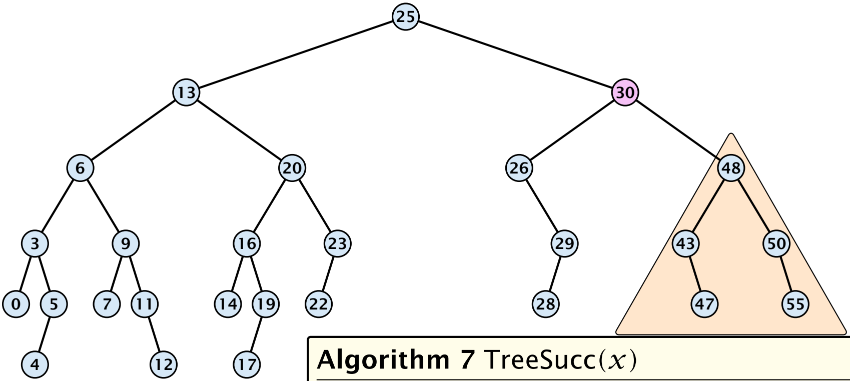
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

if right[x] $\neq$ null **return** TreeMin(right[x])
$y \leftarrow$ parent[x]
while $y \neq$ null **and** $x =$ right[y] **do**
    $x \leftarrow y$; $y \leftarrow$ parent[x]
return y;

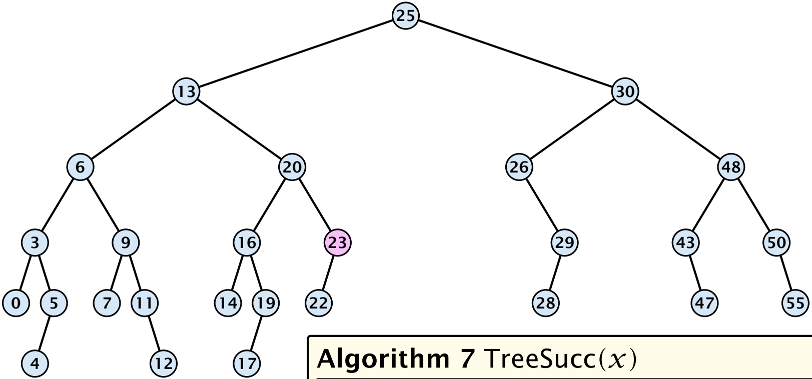
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

if right$[x] \neq$ null **return** TreeMin(right$[x]$)
$y \leftarrow$ parent$[x]$
while $y \neq$ null **and** $x =$ right$[y]$ **do**
    $x \leftarrow y; y \leftarrow$ parent$[x]$
return y;

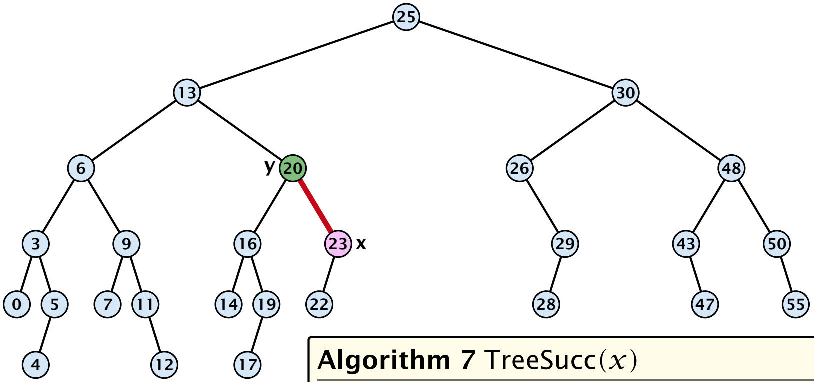
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

1: **if** right$[x] \neq$ null **return** TreeMin(right$[x]$)
2: $y \leftarrow$ parent$[x]$
3: **while** $y \neq$ null **and** $x =$ right$[y]$ **do**
4: $\quad x \leftarrow y; y \leftarrow$ parent$[x]$
5: **return** y;

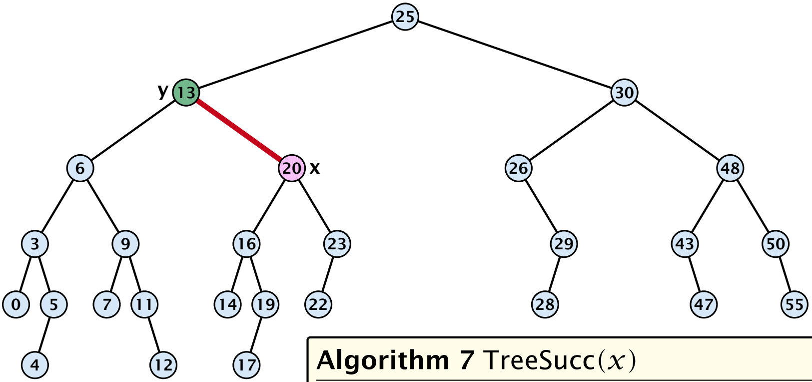
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

1: **if** right$[x] \neq$ null **return** TreeMin(right$[x]$)
2: $y \leftarrow$ parent$[x]$
3: **while** $y \neq$ null **and** $x =$ right$[y]$ **do**
4: $x \leftarrow y; y \leftarrow$ parent$[x]$
5: **return** y;

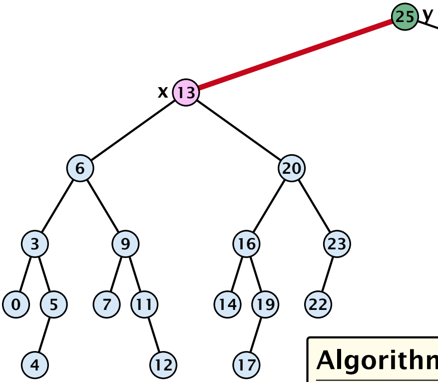
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

```
if right[x] ≠ null return TreeMin(right[x])
y ← parent[x]
while y ≠ null and x = right[y] do
        x ← y; y ← parent[x]
return y;
```

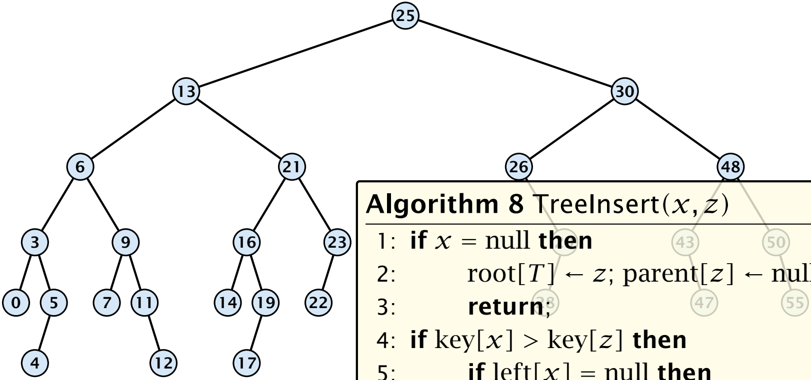
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

if right[x] $\neq$ null **return** TreeMin(right[x])
$y \leftarrow$ parent[x]
while $y \neq$ null **and** $x =$ right[y] **do**
    $x \leftarrow y$; $y \leftarrow$ parent[x]
return y;

Binary Search Trees: Insert

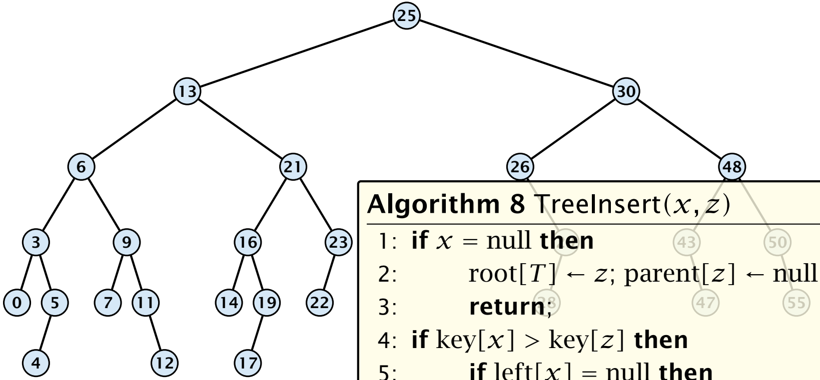


Algorithm 8 TreeInsert(x, z)

if x = null **then**
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] **then**
    if left[x] = null **then**
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null **then**
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);

Binary Search Trees: Insert

Insert element **not** in the tree.

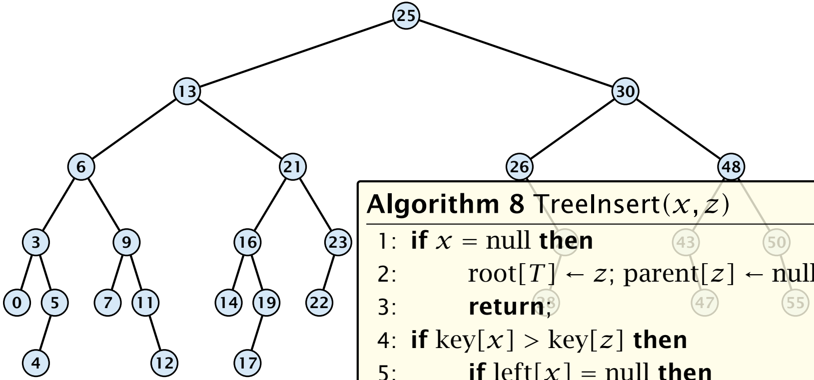


Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

Binary Search Trees: Insert

Insert element **not** in the tree.



Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

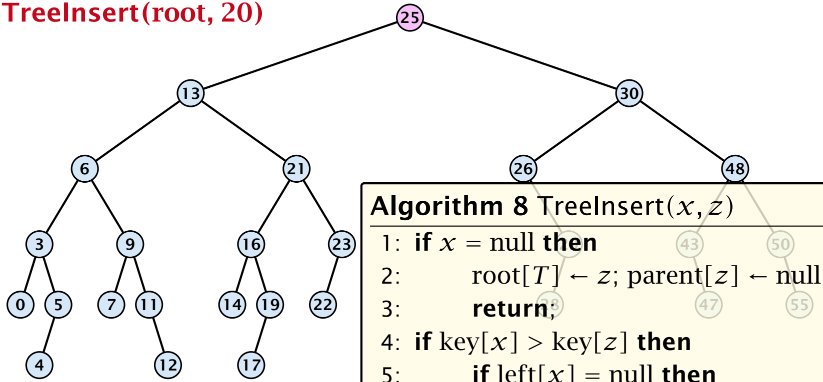
Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)



Algorithm 8 TreeInsert(x, z)

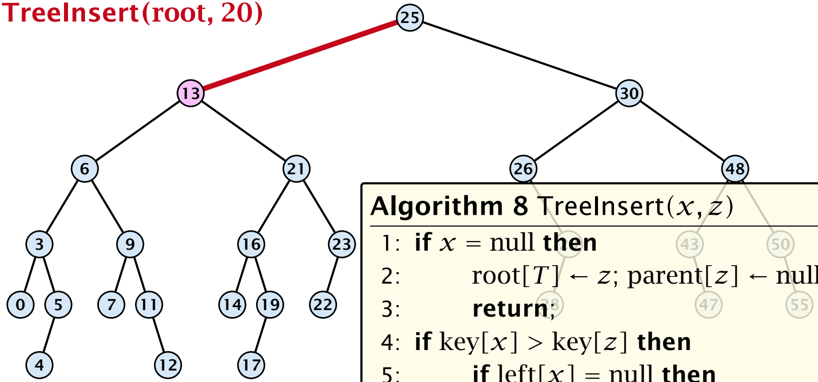
```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)



Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

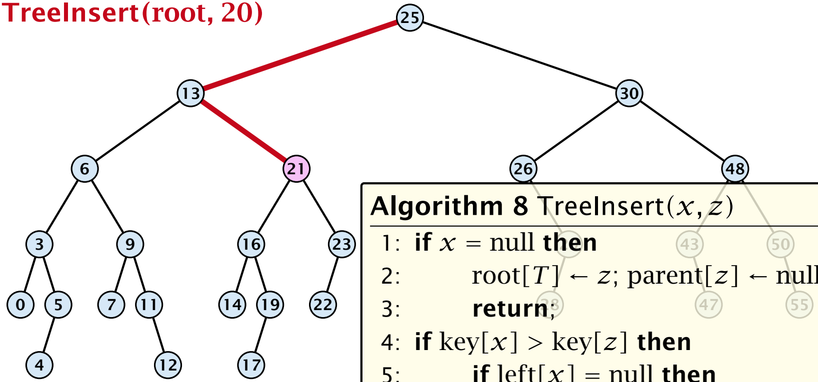
Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)



Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

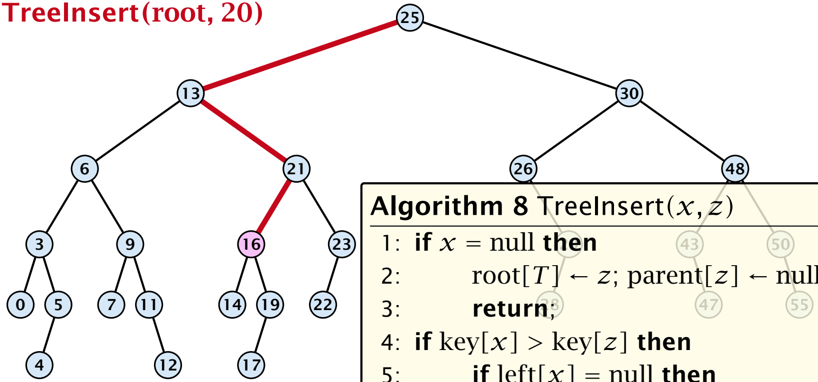
Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)



Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

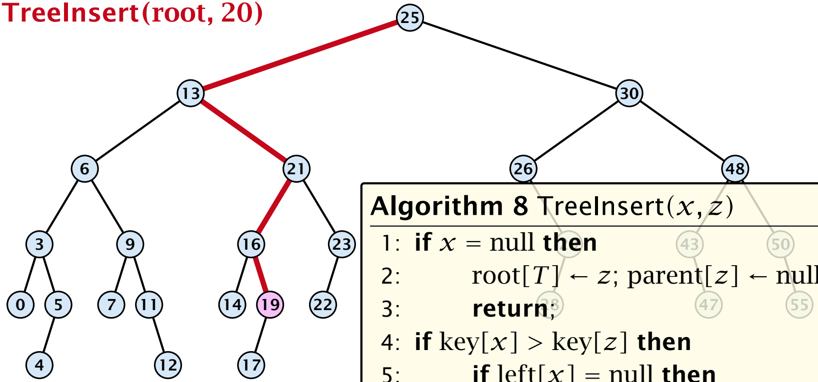
Algorithm 8 TreeInsert(x, z)

if x = null **then**
    root$[T] \leftarrow z$; parent$[z] \leftarrow$ null;
    return;
if key$[x] >$ key$[z]$ **then**
    if left$[x]$ = null **then**
        left$[x] \leftarrow z$; parent$[z] \leftarrow x$;
    else TreeInsert(left$[x], z$);
else
    if right$[x]$ = null **then**
        right$[x] \leftarrow z$; parent$[z] \leftarrow x$;
    else TreeInsert(right$[x], z$);

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)



Algorithm 8 TreeInsert(x, z)

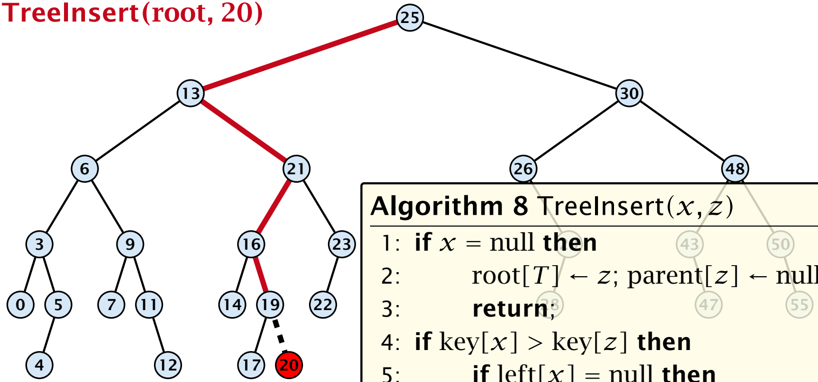
if x = null **then**
  root$[T] \leftarrow z$; parent$[z] \leftarrow$ null;
  return;
if key$[x] >$ key$[z]$ **then**
  if left$[x]$ = null **then**
    left$[x] \leftarrow z$; parent$[z] \leftarrow x$;
  else TreeInsert(left$[x]$, z);
else
  if right$[x]$ = null **then**
    right$[x] \leftarrow z$; parent$[z] \leftarrow x$;
  else TreeInsert(right$[x]$, z);

Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Binary Search Trees: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)

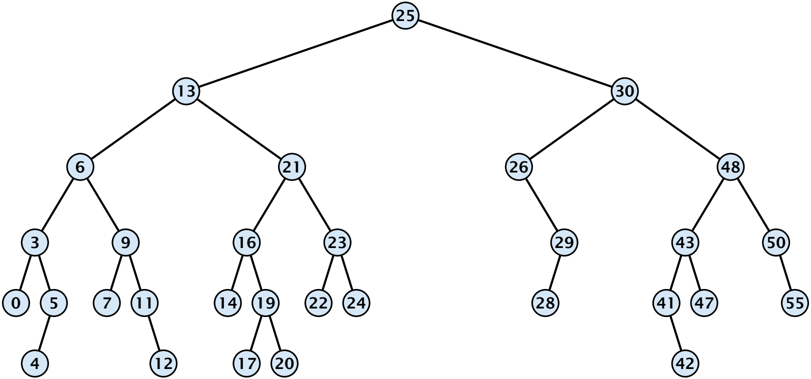


Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

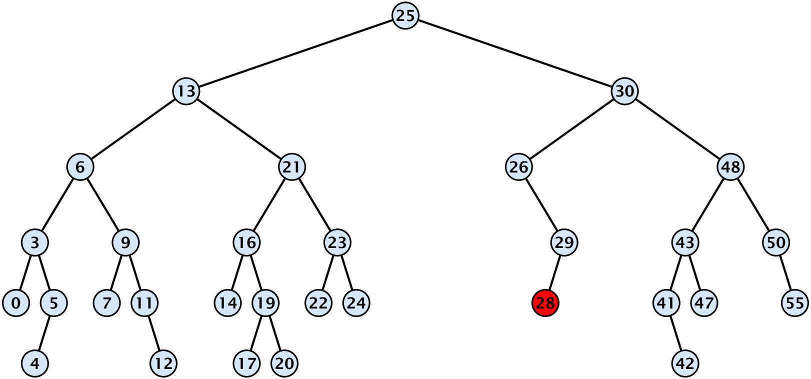
Algorithm 8 TreeInsert(x, z)

```
if x = null then
    root[T] ← z; parent[z] ← null;
    return;
if key[x] > key[z] then
    if left[x] = null then
        left[x] ← z; parent[z] ← x;
    else TreeInsert(left[x], z);
else
    if right[x] = null then
        right[x] ← z; parent[z] ← x;
    else TreeInsert(right[x], z);
```

Binary Search Trees: Delete



Binary Search Trees: Delete

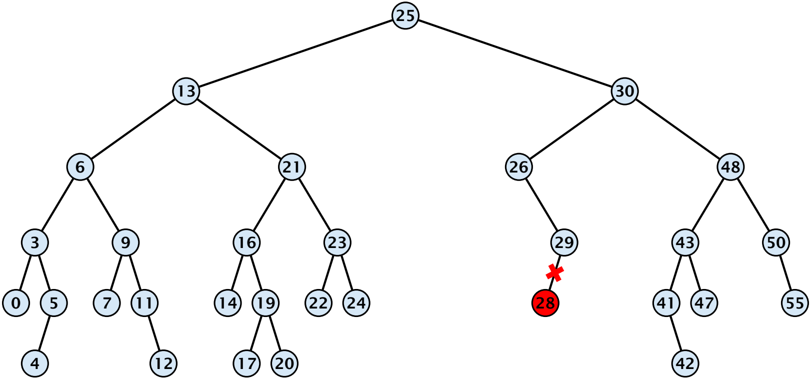


Case 1:

Element does not have any children

- Simply go to the parent and set the corresponding pointer to null.

Binary Search Trees: Delete

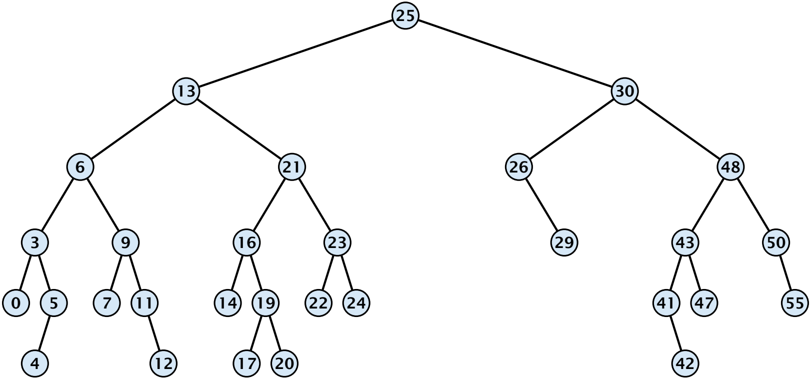


Case 1:

Element does not have any children

- Simply go to the parent and set the corresponding pointer to null.

Binary Search Trees: Delete

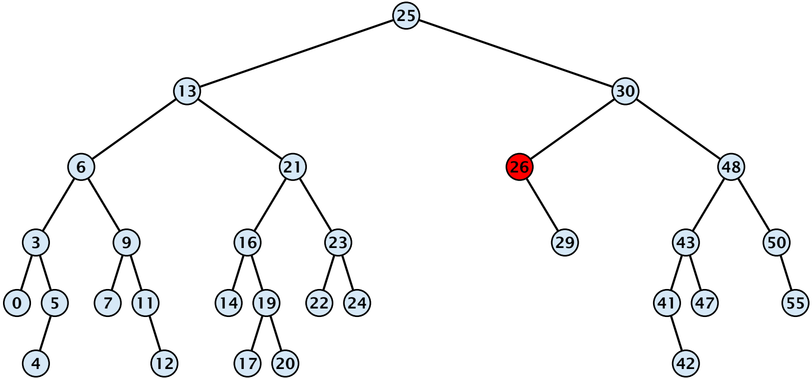


Case 1:

Element does not have any children

- Simply go to the parent and set the corresponding pointer to null.

Binary Search Trees: Delete

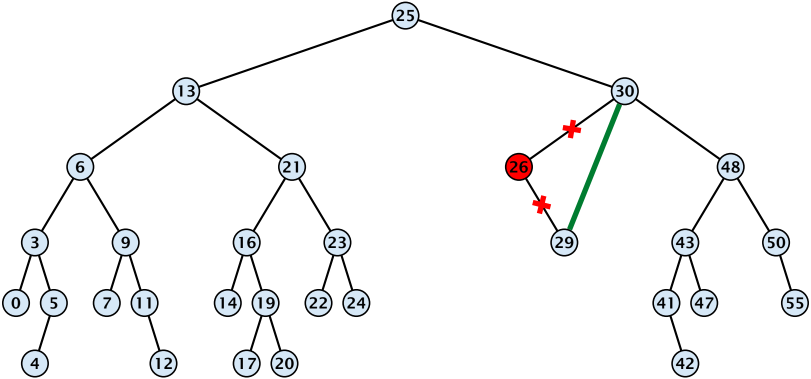


Case 2:

Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.

Binary Search Trees: Delete

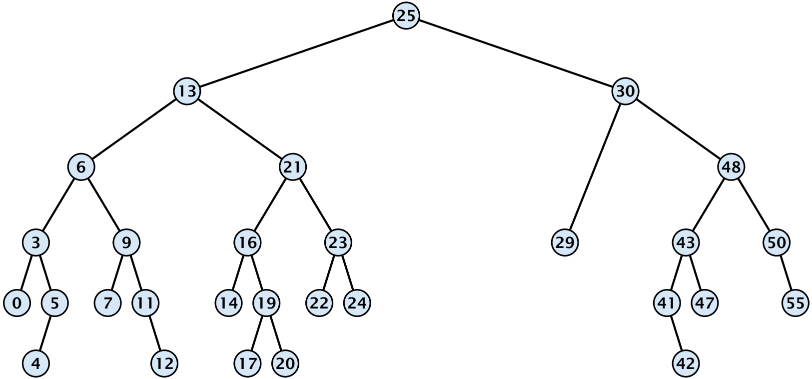


Case 2:

Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.

Binary Search Trees: Delete

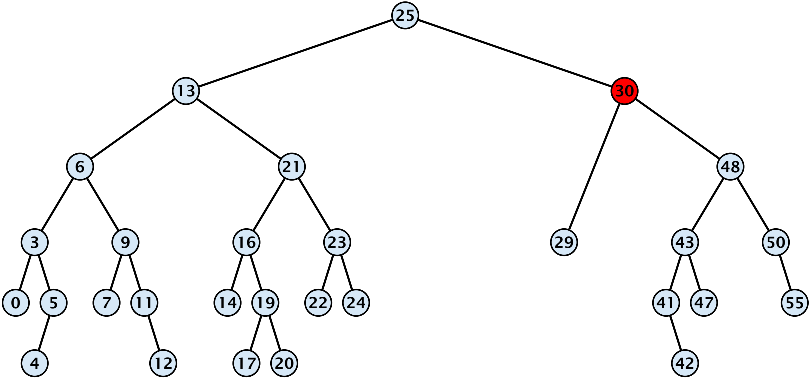


Case 2:

Element has exactly one child

- Splice the element out of the tree by connecting its parent to its successor.

Binary Search Trees: Delete

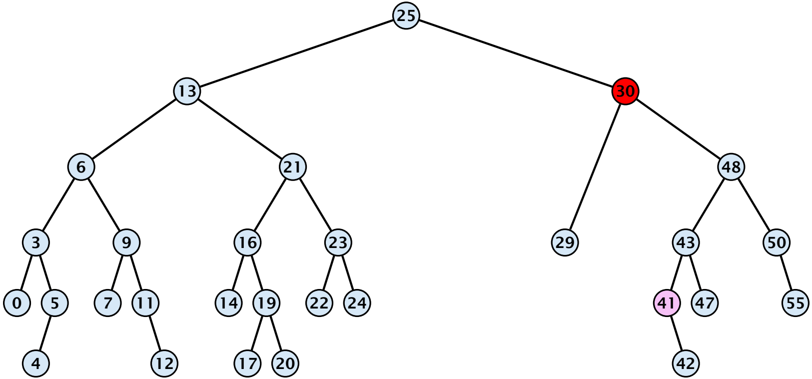


Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

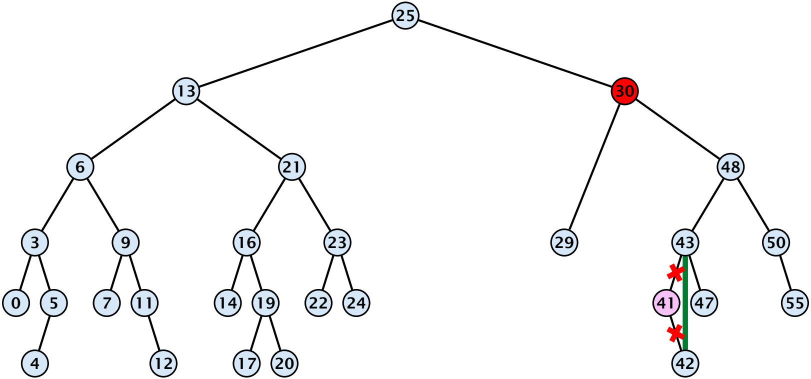


Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

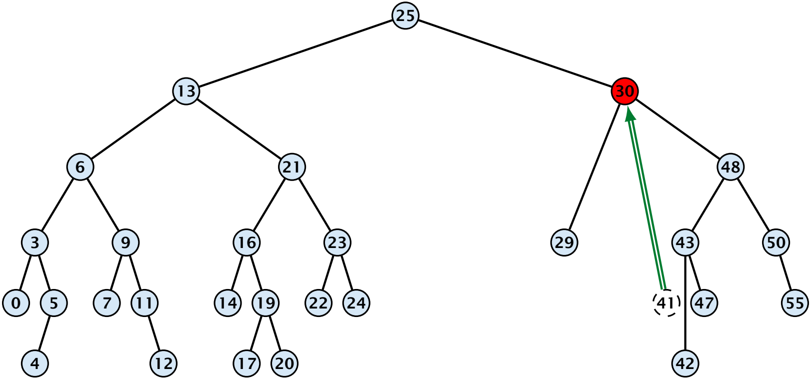


Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

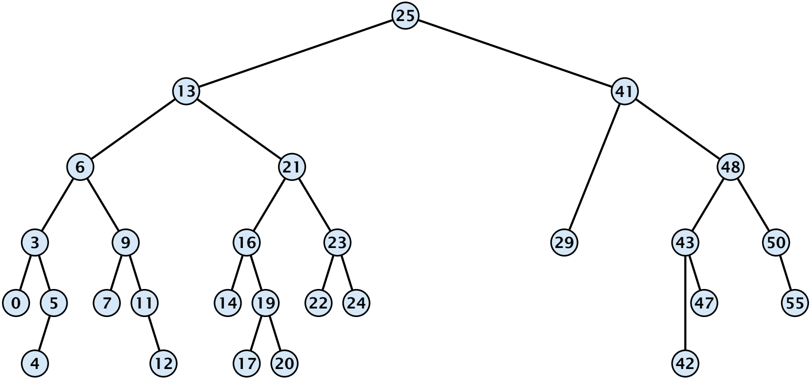


Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

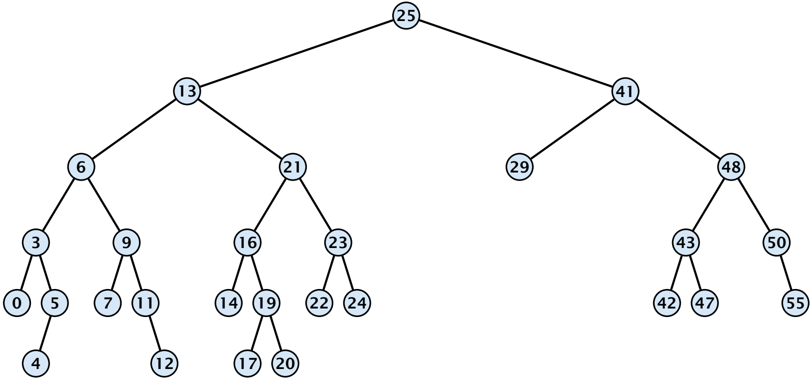


Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete



Case 3:

Element has two children

- Find the successor of the element
- Splice successor out of the tree
- Replace content of element by content of successor

Binary Search Trees: Delete

Algorithm 9 TreeDelete(z)

if left[z] = null **or** right[z] = null
    then $y \leftarrow z$ **else** $y \leftarrow$ TreeSucc(z);   select y to splice out
if left[y] $\neq$ null
    then $x \leftarrow$ left[y] **else** $x \leftarrow$ right[y]; x is child of y (or null)
if $x \neq$ null **then** parent[x] $\leftarrow$ parent[y];   parent[x] is correct
if parent[y] = null **then**
    root[T] $\leftarrow x$
else
    if y = left[parent[x]] **then**
        left[parent[y]] $\leftarrow x$
    else
        right[parent[y]] $\leftarrow x$
if $y \neq z$ **then** copy y-data to z

(lines 6–12: fix pointer to x)

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees

With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees

With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

Balanced Binary Search Trees

All operations on a binary search tree can be performed in time $\mathcal{O}(h)$, where h denotes the height of the tree.

However the height of the tree may become as large as $\Theta(n)$.

Balanced Binary Search Trees
With each insert- and delete-operation perform local adjustments to guarantee a height of $\mathcal{O}(\log n)$.

AVL-trees, Red-black trees, Scapegoat trees, 2-3 trees, B-trees, AA trees, Treaps

similar: SPLAY trees.

7.2 Red Black Trees

Definition 1

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

Definition 1

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

Definition 1

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

Definition 1

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

Definition 1

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

7.2 Red Black Trees

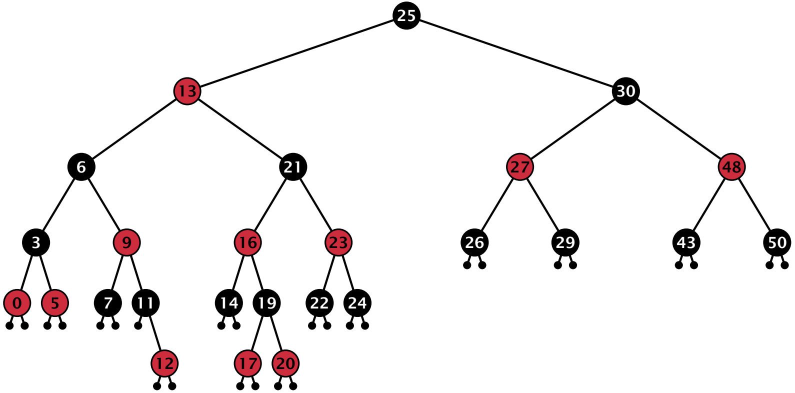
Definition 1

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data

Red Black Trees: Example



7.2 Red Black Trees

Lemma 2

A red-black tree with n internal nodes has height at most $\mathcal{O}(\log n)$.

Definition 3

The black height $\text{bh}(v)$ of a node v in a red black tree is the number of black nodes on a path from v to a leaf vertex (not counting v).

We first show:

Lemma 4

A sub-tree of black height $\text{bh}(v)$ in a red black tree contains at least $2^{\text{bh}(v)} - 1$ internal vertices.

7.2 Red Black Trees

Lemma 2

A red-black tree with n internal nodes has height at most $\mathcal{O}(\log n)$.

Definition 3

The black height $\mathrm{bh}(v)$ of a node v in a red black tree is the number of black nodes on a path from v to a leaf vertex (not counting v).

We first show:

Lemma 4

A sub-tree of black height $\mathrm{bh}(v)$ in a red black tree contains at least $2^{\mathrm{bh}(v)} - 1$ internal vertices.

7.2 Red Black Trees

Lemma 2

A red-black tree with n internal nodes has height at most $\mathcal{O}(\log n)$.

Definition 3

The black height $\mathrm{bh}(v)$ of a node v in a red black tree is the number of black nodes on a path from v to a leaf vertex (not counting v).

We first show:

Lemma 4

A sub-tree of black height $\mathrm{bh}(v)$ in a red black tree contains at least $2^{\mathrm{bh}(v)} - 1$ internal vertices.

7.2 Red Black Trees

Proof of Lemma 4.

Induction on the height of v.

base case ($\text{height}(v) = 0$)

- If height(v) (maximum distance btw. v and a node in the sub-tree rooted at v) is 0 then v is a leaf.
- The black height of v is 0.
- The sub-tree rooted at v contains $0 = 2^{\text{bh}(v)} - 1$ inner vertices.

7.2 Red Black Trees

Proof of Lemma 4.

Induction on the height of v.

base case $(\text{height}(v) = 0)$

- If height(v) (maximum distance btw. v and a node in the sub-tree rooted at v) is 0 then v is a leaf.
- The black height of v is 0.
- The sub-tree rooted at v contains $0 = 2^{\text{bh}(v)} - 1$ inner vertices.

7.2 Red Black Trees

Proof of Lemma 4.

Induction on the height of v.

base case (height$(v) = 0$)

- If height(v) (maximum distance btw. v and a node in the sub-tree rooted at v) is 0 then v is a leaf.
- The black height of v is 0.
- The sub-tree rooted at v contains $0 = 2^{\mathrm{bh}(v)} - 1$ inner vertices.

7.2 Red Black Trees

Proof of Lemma 4.

Induction on the height of v.

base case $(\text{height}(v) = 0)$

- If $\text{height}(v)$ (maximum distance btw. v and a node in the sub-tree rooted at v) is 0 then v is a leaf.
- The black height of v is 0.
- The sub-tree rooted at v contains $0 = 2^{\text{bh}(v)} - 1$ inner vertices.

7.2 Red Black Trees

Proof of Lemma 4.

Induction on the height of v.

base case $(\text{height}(v) = 0)$

- If $\text{height}(v)$ (maximum distance btw. v and a node in the sub-tree rooted at v) is 0 then v is a leaf.
- The black height of v is 0.
- The sub-tree rooted at v contains $0 = 2^{\text{bh}(v)} - 1$ inner vertices.

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\text{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

□

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\text{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\text{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\text{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\text{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

7.2 Red Black Trees

Proof (cont.)

induction step

- Supose v is a node with $\text{height}(v) > 0$.
- v has two children with strictly smaller height.
- These children (c_1, c_2) either have $\text{bh}(c_i) = \text{bh}(v)$ or $\text{bh}(c_i) = \text{bh}(v) - 1$.
- By induction hypothesis both sub-trees contain at least $2^{\text{bh}(v)-1} - 1$ internal vertices.
- Then T_v contains at least $2(2^{\text{bh}(v)-1} - 1) + 1 \geq 2^{\text{bh}(v)} - 1$ vertices.

□

7.2 Red Black Trees

Proof of Lemma 2.

Let h denote the height of the red-black tree, and let P denote a path from the root to the furthest leaf.

At least half of the node on P must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \leq n$.

Hence, $h \leq 2\log(n+1) = \mathcal{O}(\log n)$. □

7.2 Red Black Trees

Proof of Lemma 2.

Let h denote the height of the red-black tree, and let P denote a path from the root to the furthest leaf.

At least half of the node on P must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \le n$.

Hence, $h \le 2\log(n+1) = \mathcal{O}(\log n)$. □

7.2 Red Black Trees

Proof of Lemma 2.

Let h denote the height of the red-black tree, and let P denote a path from the root to the furthest leaf.

At least half of the node on P must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \leq n$.

Hence, $h \leq 2\log(n+1) = \mathcal{O}(\log n)$. □

7.2 Red Black Trees

Proof of Lemma 2.

Let h denote the height of the red-black tree, and let P denote a path from the root to the furthest leaf.

At least half of the node on P must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \leq n$.

Hence, $h \leq 2\log(n+1) = \mathcal{O}(\log n)$. □

7.2 Red Black Trees

Proof of Lemma 2.

Let h denote the height of the red-black tree, and let P denote a path from the root to the furthest leaf.

At least half of the node on P must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \leq n$.

Hence, $h \leq 2\log(n+1) = \mathcal{O}(\log n)$. □

7.2 Red Black Trees

Proof of Lemma 2.

Let h denote the height of the red-black tree, and let P denote a path from the root to the furthest leaf.

At least half of the node on P must be black, since a red node must be followed by a black node.

Hence, the black height of the root is at least $h/2$.

The tree contains at least $2^{h/2} - 1$ internal vertices. Hence, $2^{h/2} - 1 \leq n$.

Hence, $h \leq 2\log(n+1) = \mathcal{O}(\log n)$. □

7.2 Red Black Trees

Definition 5

A red black tree is a balanced binary search tree in which each internal node has two children. Each internal node has a color, such that

1. The root is black.
2. All leaf nodes are black.
3. For each node, all paths to descendant leaves contain the same number of black nodes.
4. If a node is red then both its children are black.

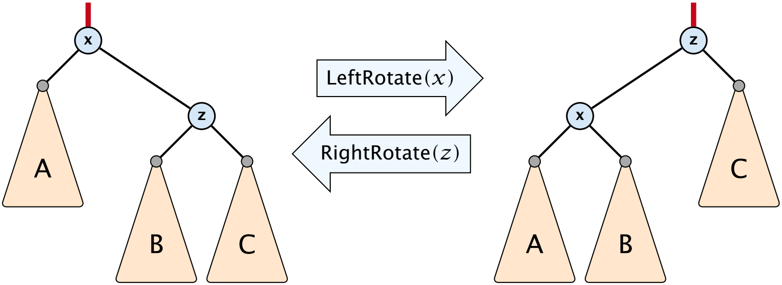
The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data.

7.2 Red Black Trees

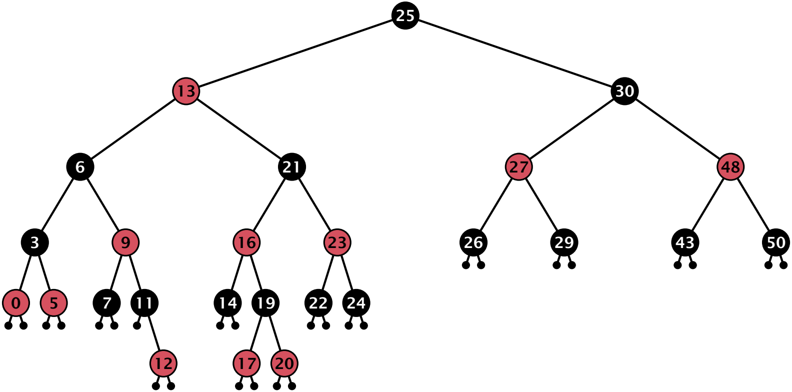
We need to adapt the insert and delete operations so that the red black properties are maintained.

Rotations

The properties will be maintained through rotations:



Red Black Trees: Insert

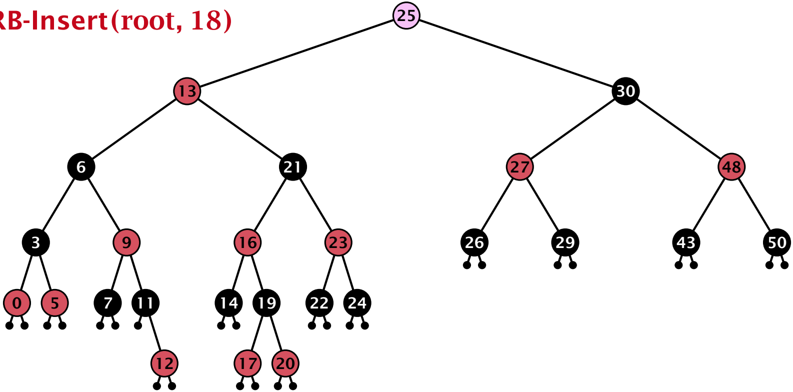


Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

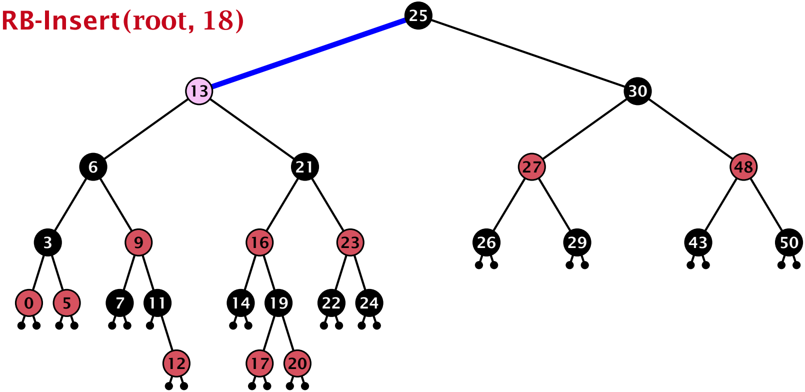
RB-Insert(root, 18)



Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

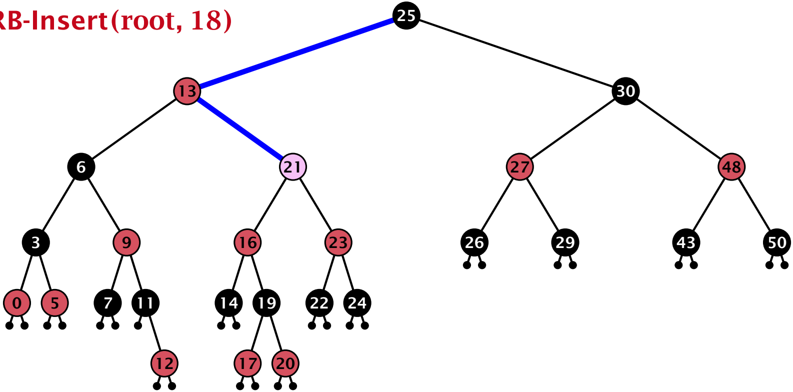


Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

RB-Insert(root, 18)

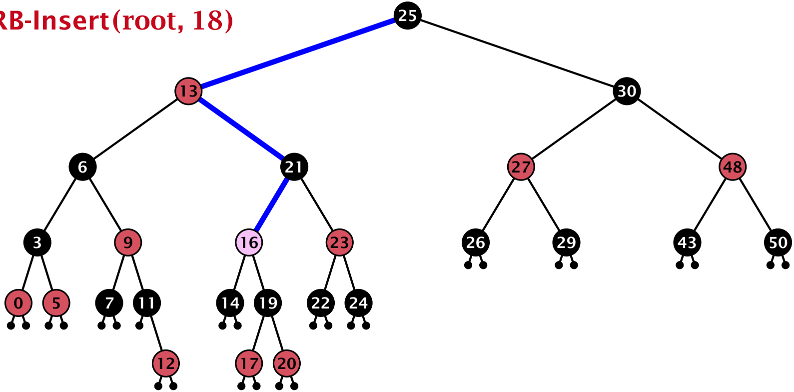


Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

RB-Insert(root, 18)

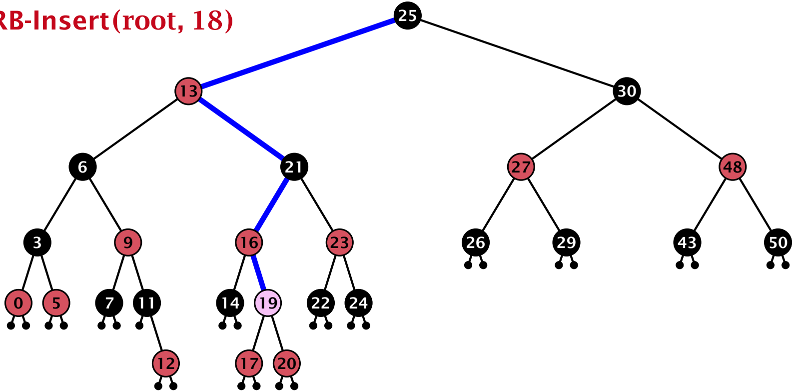


Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

RB-Insert(root, 18)

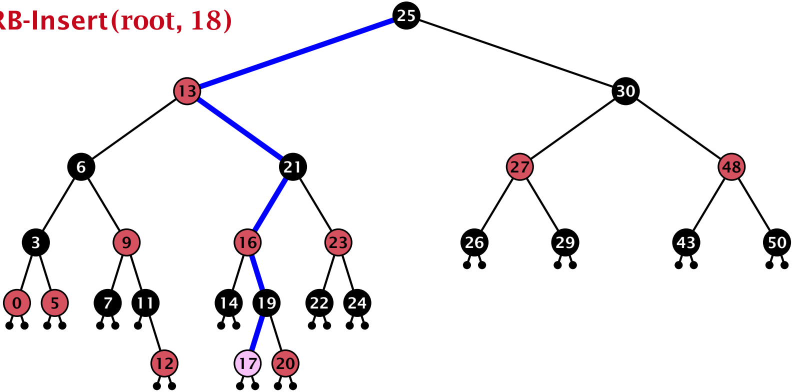


Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

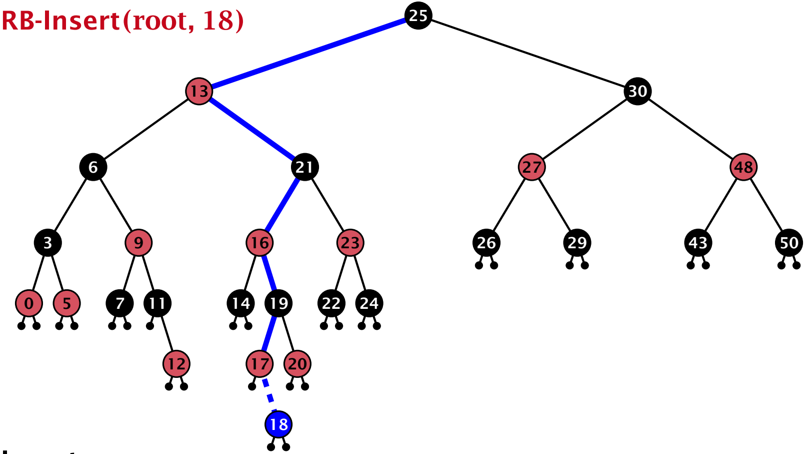
RB-Insert(root, 18)



Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

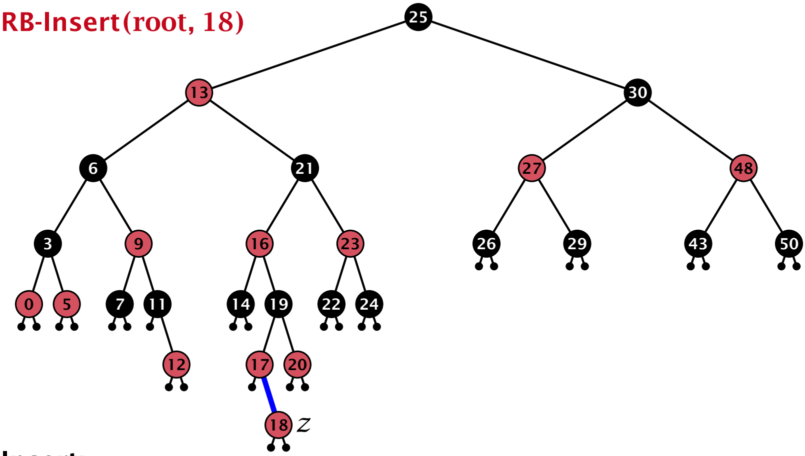
Red Black Trees: Insert



Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert



Insert:

- first make a normal insert into a binary search tree
- then fix red-black properties

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Invariant of the fix-up algorithm:

- z is a red node
- the black-height property is fulfilled at every node
- the only violation of red-black properties occurs at z and $\text{parent}[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

Algorithm 10 InsertFix(z)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then
        uncle ← right[grandparent[z]]
        if col[uncle] = red then
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else
            if z = right[parent[z]] then
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

Red Black Trees: Insert

Algorithm 10 InsertFix(z)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then          z in left subtree of grandparent
        uncle ← right[grandparent[z]]
        if col[uncle] = red then
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else
            if z = right[parent[z]] then
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

Red Black Trees: Insert

Algorithm 10 InsertFix(z)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then
        uncle ← right[grandparent[z]]
        if col[uncle] = red then                         Case 1: uncle red
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else
            if z = right[parent[z]] then
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

Red Black Trees: Insert

Algorithm 10 InsertFix(z)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then
        uncle ← right[grandparent[z]]
        if col[uncle] = red then
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else                                    Case 2: uncle black
            if z = right[parent[z]] then
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

Red Black Trees: Insert

Algorithm 10 InsertFix(z)

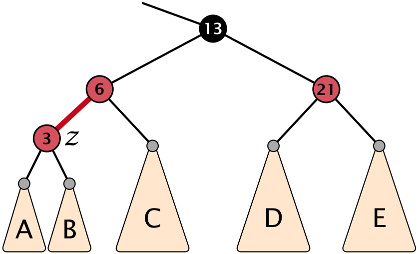
```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then
        uncle ← right[grandparent[z]]
        if col[uncle] = red then
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else
            if z = right[parent[z]] then                2a: z right child
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

Red Black Trees: Insert

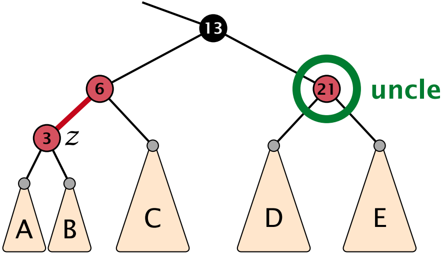
Algorithm 10 InsertFix(z)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then
        uncle ← right[grandparent[z]]
        if col[uncle] = red then
            col[p[z]] ← black; col[u] ← black;
            col[gp[z]] ← red; z ← grandparent[z];
        else
            if z = right[parent[z]] then
                z ← p[z]; LeftRotate(z);
            col[p[z]] ← black; col[gp[z]] ← red;   2b: z left child
            RightRotate(gp[z]);
    else same as then-clause but right and left exchanged
col(root[T]) ← black;
```

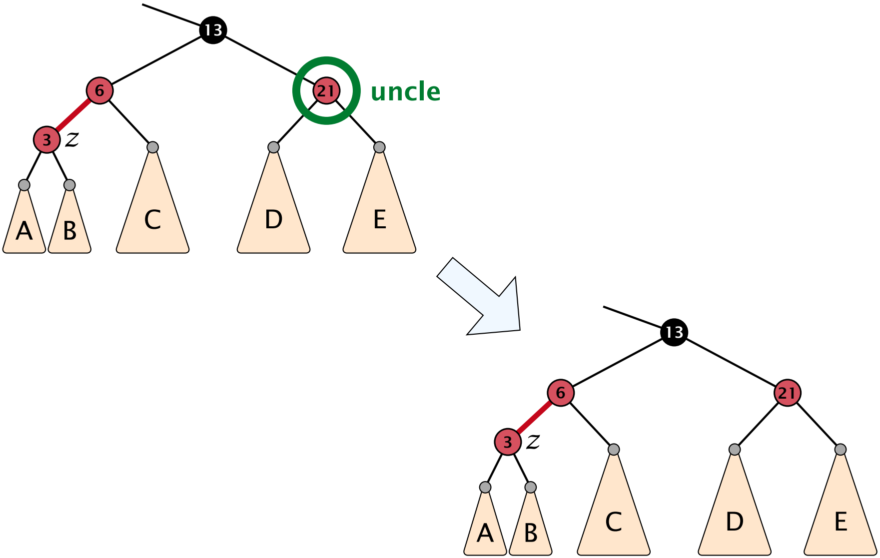
Case 1: Red Uncle



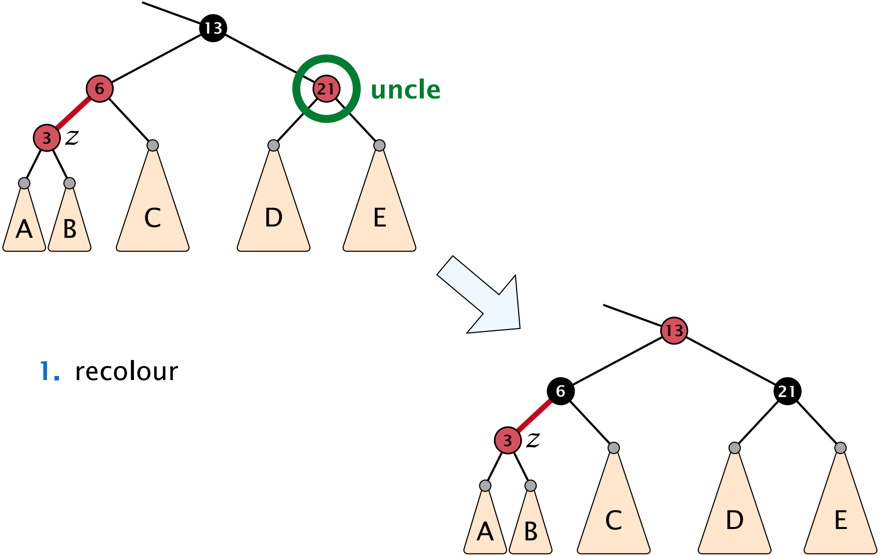
Case 1: Red Uncle



Case 1: Red Uncle

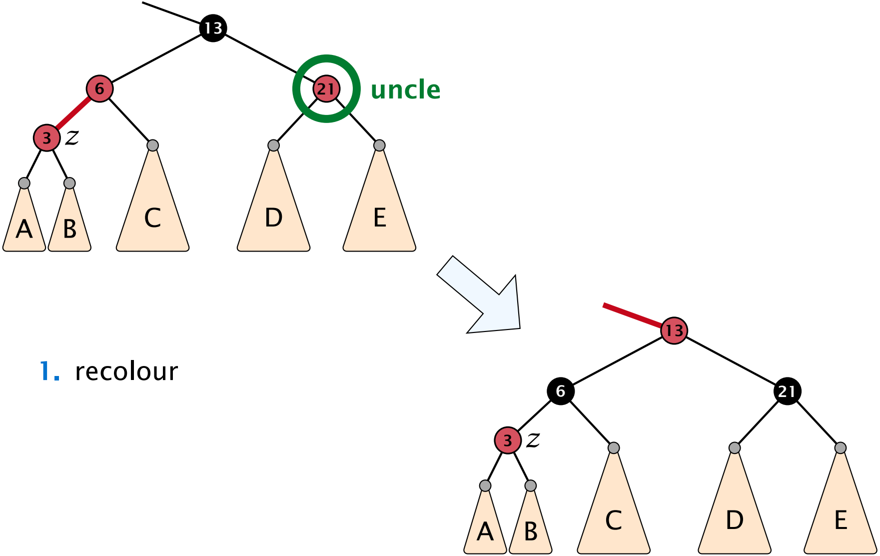


Case 1: Red Uncle



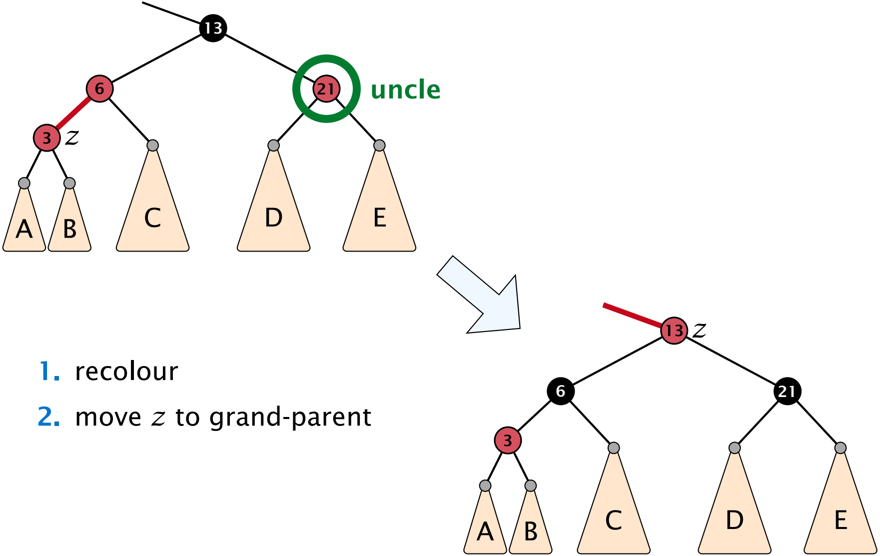
1. recolour

Case 1: Red Uncle



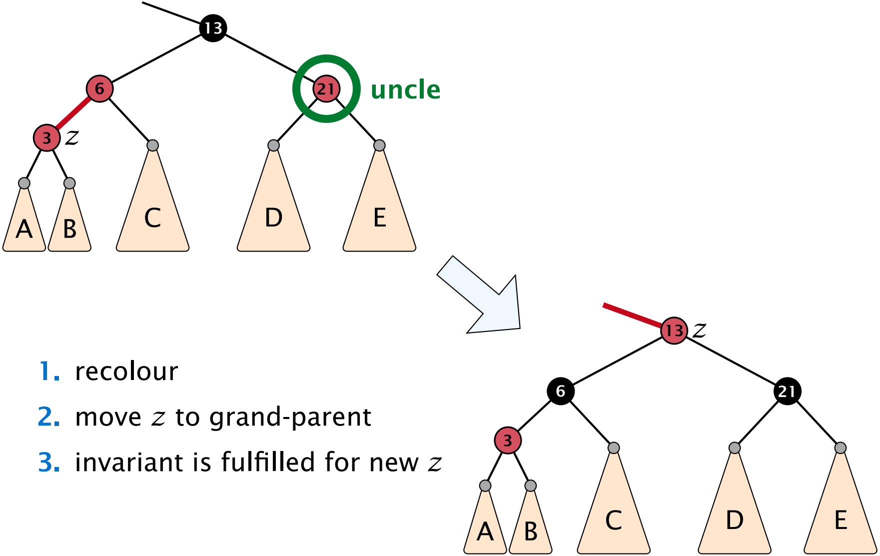
1. recolour

Case 1: Red Uncle



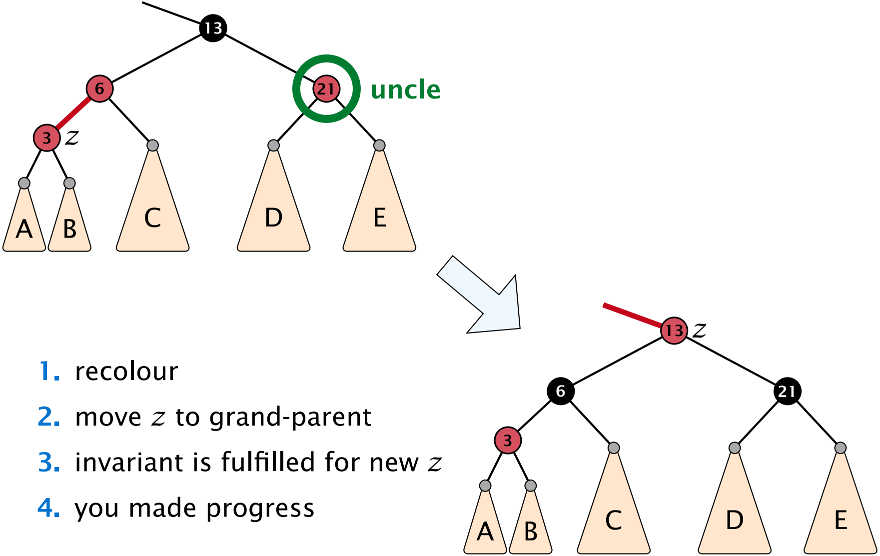
1. recolour
2. move z to grand-parent

Case 1: Red Uncle



1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z

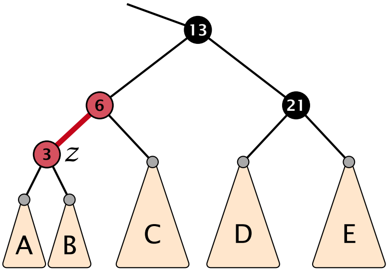
Case 1: Red Uncle



1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress

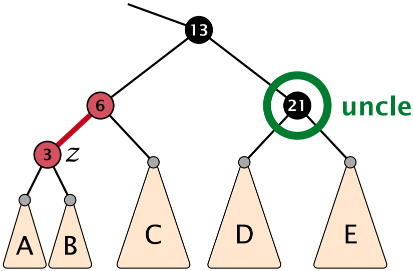
Case 2b: Black uncle and z is left child

1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree



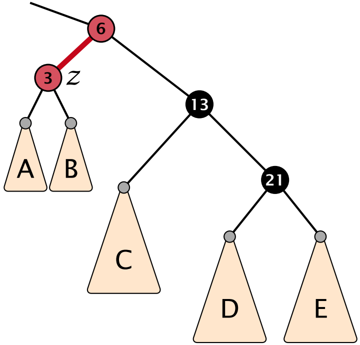
Case 2b: Black uncle and z is left child

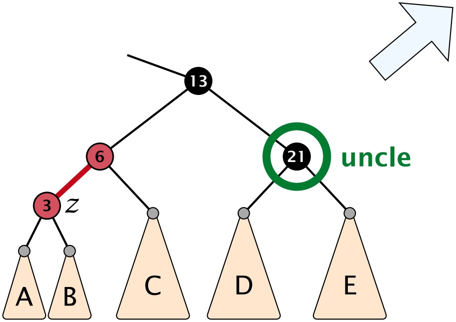
1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree



Case 2b: Black uncle and z is left child

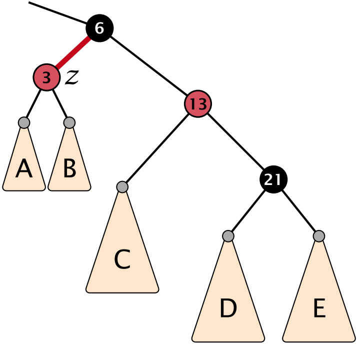
1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree

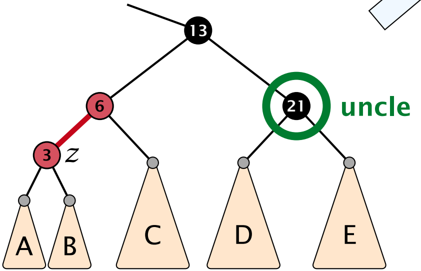




Case 2b: Black uncle and z is left child

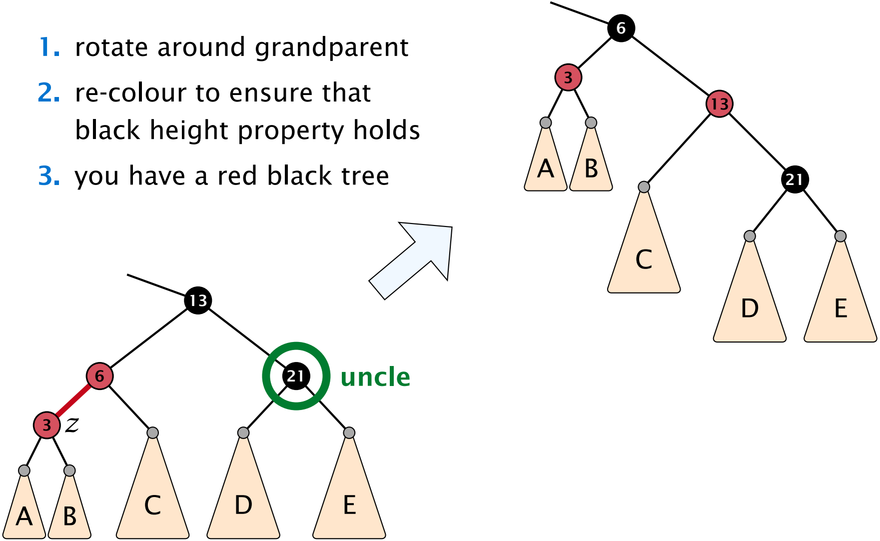
1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree





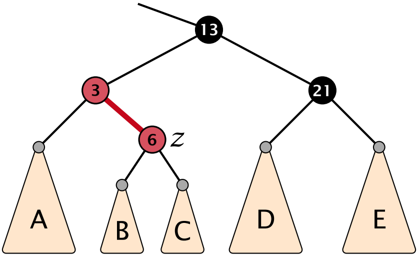
Case 2b: Black uncle and z is left child

1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree



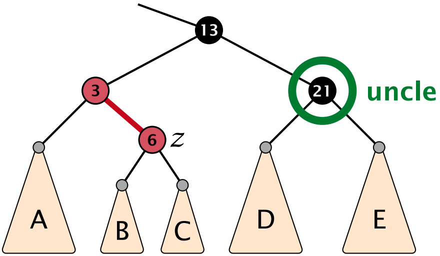
Case 2a: Black uncle and z is right child

1. rotate around parent
2. move z downwards
3. you have Case 2b.



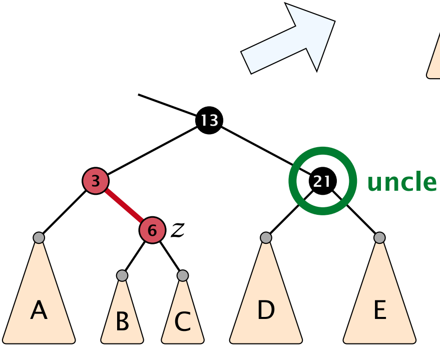
Case 2a: Black uncle and z is right child

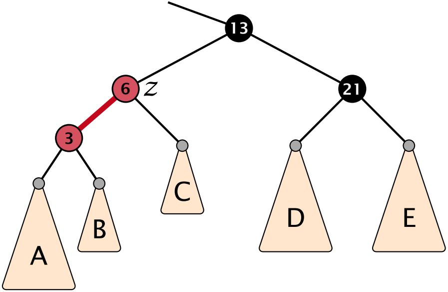
1. rotate around parent
2. move z downwards
3. you have Case 2b.



Case 2a: Black uncle and z is right child

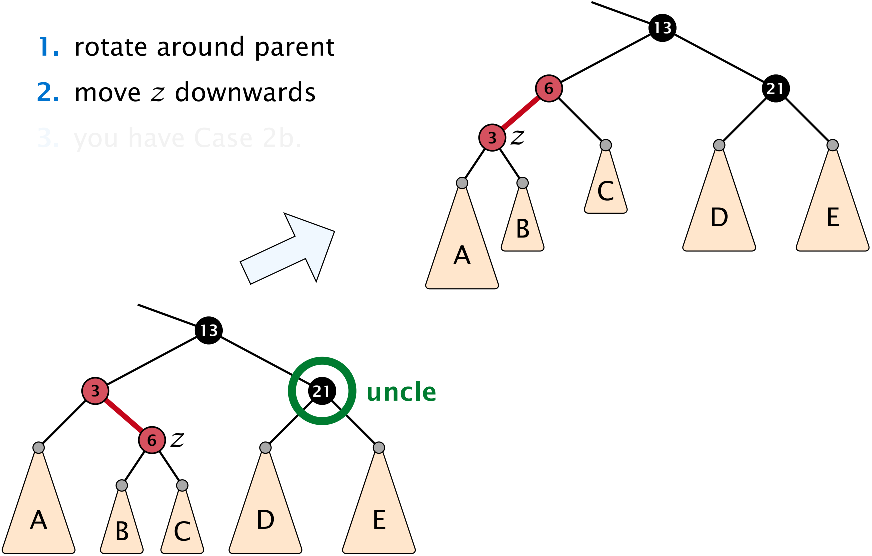
1. rotate around parent
2. move z downwards
3. you have Case 2b.





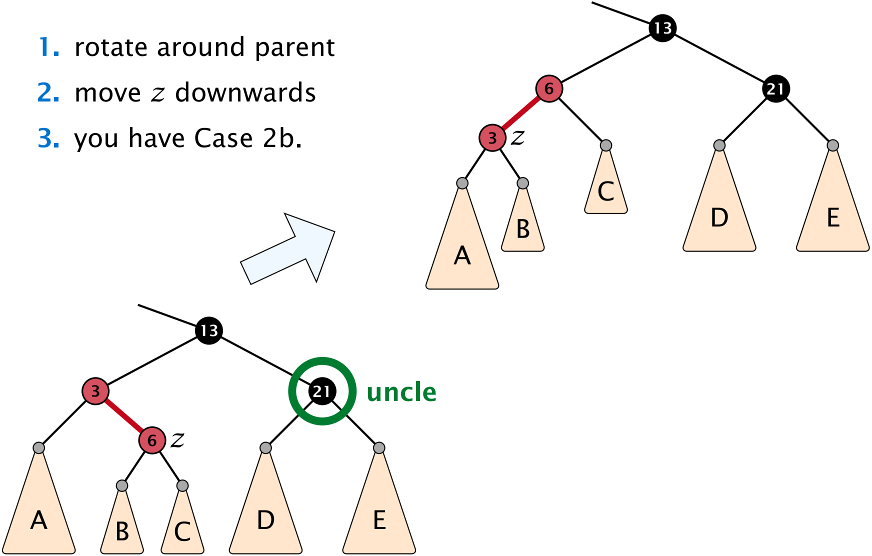
Case 2a: Black uncle and z is right child

1. rotate around parent
2. move z downwards
3. you have Case 2b.



Case 2a: Black uncle and z is right child

1. rotate around parent
2. move z downwards
3. you have Case 2b.



Red Black Trees: Insert

Running time:

- Only Case 1 may repeat; but only $h/2$ many steps, where h is the height of the tree.
- Case 2a → Case 2b → red-black tree
- Case 2b → red-black tree

Performing Case 1 at most $\mathcal{O}(\log n)$ times and every other case at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colorings and at most 2 rotations.

Red Black Trees: Insert

Running time:

- Only Case 1 may repeat; but only $h/2$ many steps, where h is the height of the tree.
- Case 2a → Case 2b → red-black tree
- Case 2b → red-black tree

Performing Case 1 at most $\mathcal{O}(\log n)$ times and every other case at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colorings and at most 2 rotations.

Red Black Trees: Insert

Running time:

- Only Case 1 may repeat; but only $h/2$ many steps, where h is the height of the tree.
- Case 2a → Case 2b → red-black tree
- Case 2b → red-black tree

Performing Case 1 at most $\mathcal{O}(\log n)$ times and every other case at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colorings and at most 2 rotations.

Red Black Trees: Insert

Running time:

- Only Case 1 may repeat; but only $h/2$ many steps, where h is the height of the tree.
- Case 2a → Case 2b → red-black tree
- Case 2b → red-black tree

Performing Case 1 at most $\mathcal{O}(\log n)$ times and every other case at most once, we get a red-black tree. Hence $\mathcal{O}(\log n)$ re-colorings and at most 2 rotations.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

First do a standard delete.

If the spliced out node x was red everyhting is fine.

If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete

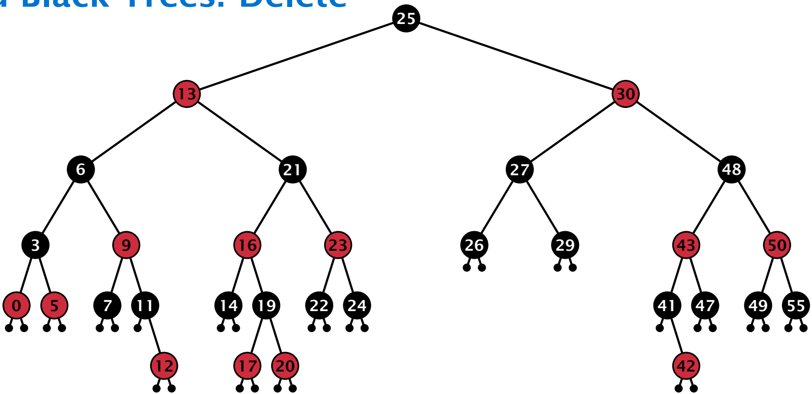
First do a standard delete.

If the spliced out node x was red everyhting is fine.

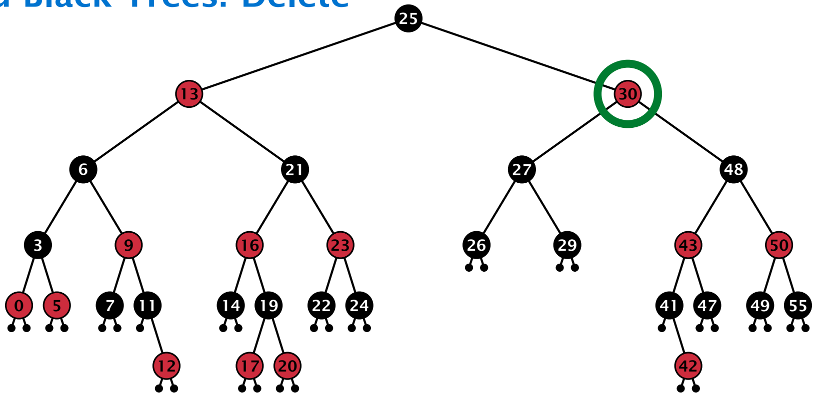
If it was black there may be the following problems.

- Parent and child of x were red; two adjacent red vertices.
- If you delete the root, the root may now be red.
- Every path from an ancestor of x to a descendant leaf of x changes the number of black nodes. Black height property might be violated.

Red Black Trees: Delete



Red Black Trees: Delete

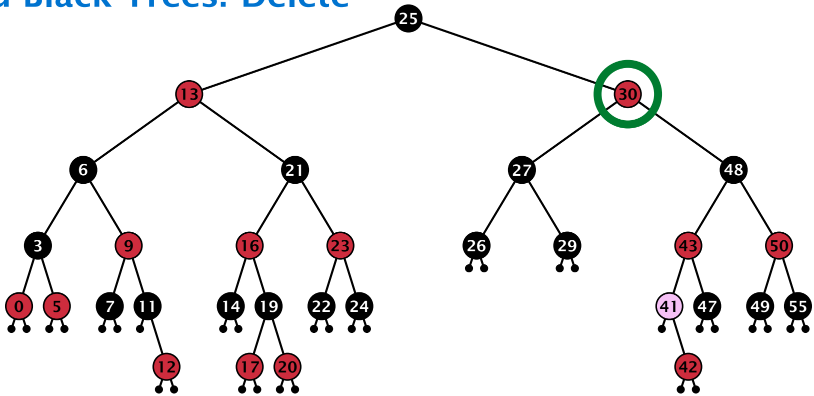


Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

Red Black Trees: Delete

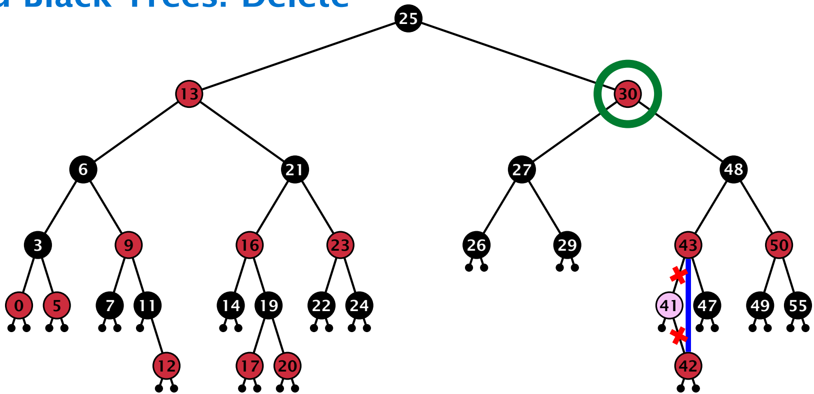


Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

Red Black Trees: Delete

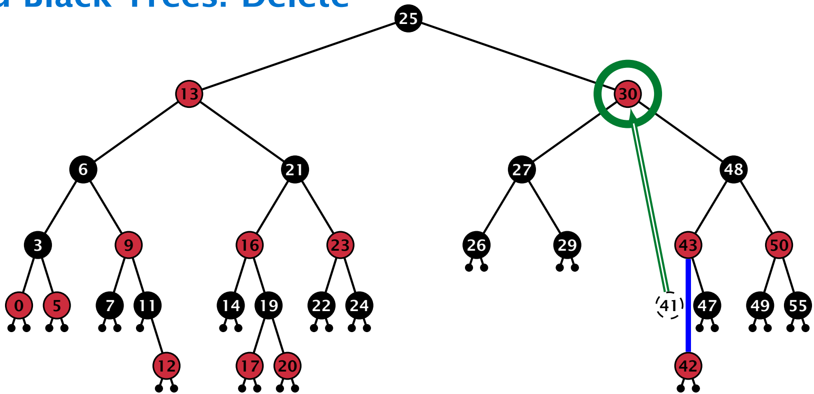


Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

Red Black Trees: Delete

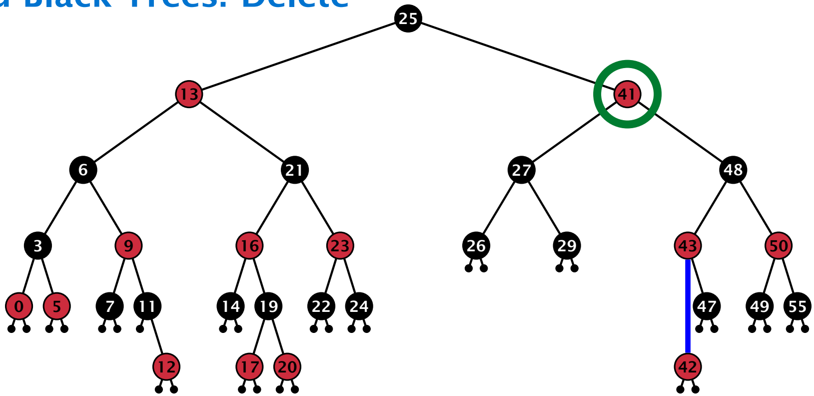


Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

Red Black Trees: Delete

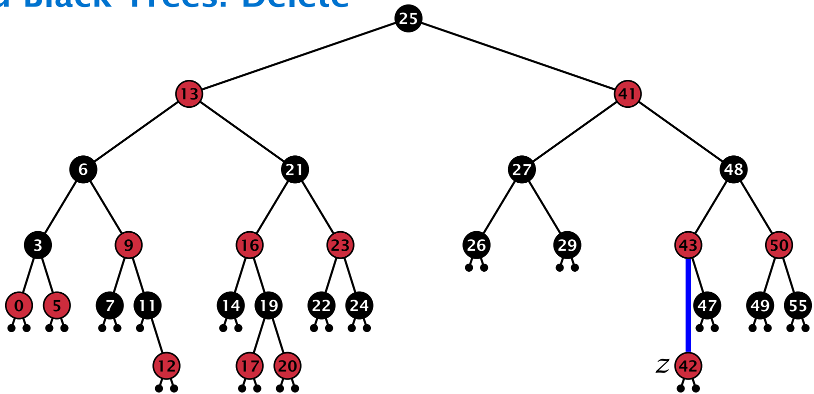


Case 3:

Element has two children

- do normal delete
- when replacing content by content of successor, don't change color of node

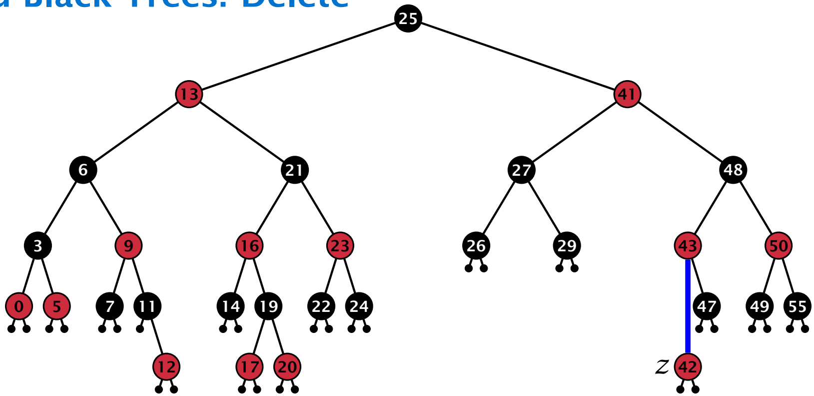
Red Black Trees: Delete



Delete:

- deleting black node messes up black-height property
- if z is red, we can simply color it black and everything is fine
- the problem is if z is black (e.g. a dummy-leaf); we call a fix-up procedure to fix the problem.

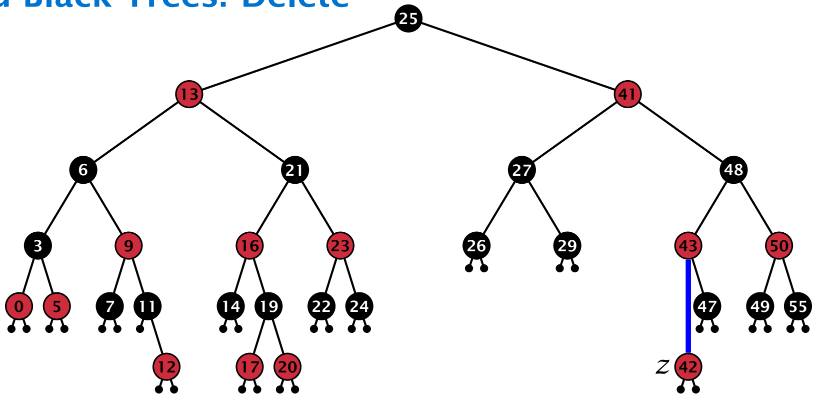
Red Black Trees: Delete



Delete:

- deleting black node messes up black-height property
- if z is red, we can simply color it black and everything is fine
- the problem is if z is black (e.g. a dummy-leaf); we call a fix-up procedure to fix the problem.

Red Black Trees: Delete



Delete:

- deleting black node messes up black-height property
- if z is red, we can simply color it black and everything is fine
- the problem is if z is black (e.g. a dummy-leaf); we call a fix-up procedure to fix the problem.

Red Black Trees: Delete

Invariant of the fix-up algorithm

- the node z is black
- if we “assign” a fake black unit to the edge from z to its parent then the black-height property is fulfilled

Goal: make rotations in such a way that you at some point can remove the fake black unit from the edge.

Red Black Trees: Delete

Invariant of the fix-up algorithm

- the node z is black
- if we "assign" a fake black unit to the edge from z to its parent then the black-height property is fulfilled

Goal: make rotations in such a way that you at some point can remove the fake black unit from the edge.

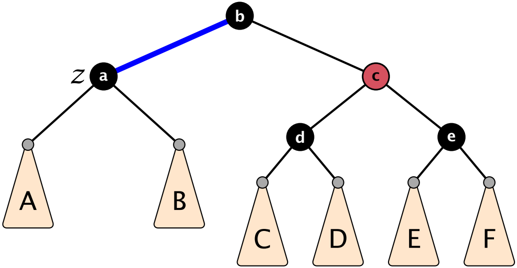
Red Black Trees: Delete

Invariant of the fix-up algorithm

- the node z is black
- if we “assign” a fake black unit to the edge from z to its parent then the black-height property is fulfilled

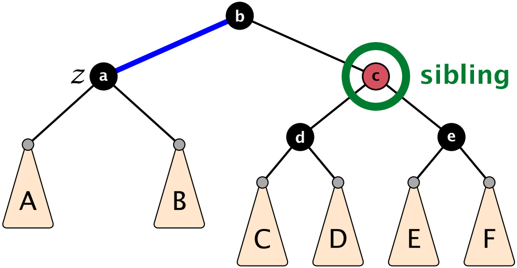
Goal: make rotations in such a way that you at some point can remove the fake black unit from the edge.

Case 1: Sibling of z is red



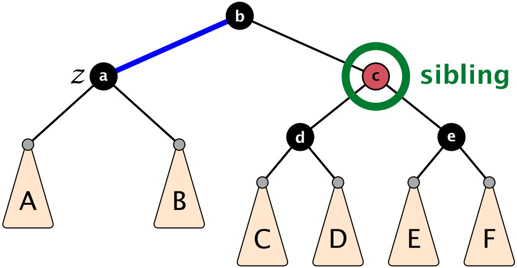
1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4

Case 1: Sibling of z is red

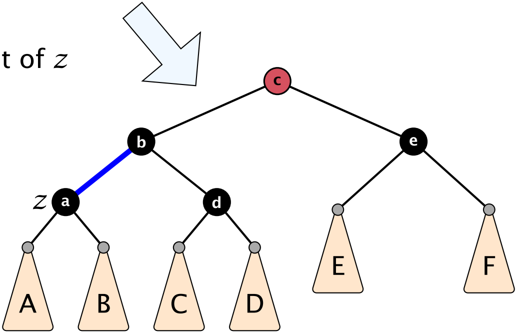


1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4

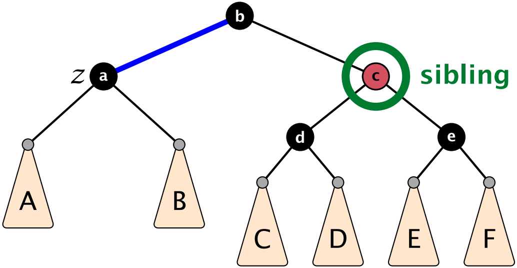
Case 1: Sibling of z is red



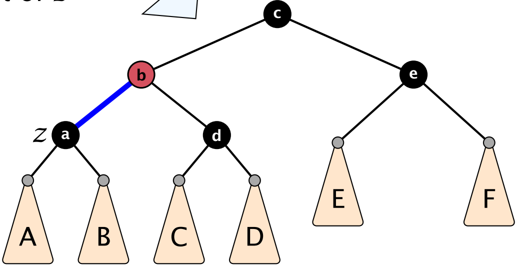
1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black (and parent of z is red)
4. Case 2 (special), or Case 3, or Case 4



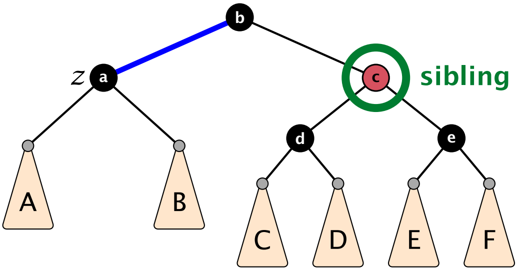
Case 1: Sibling of z is red



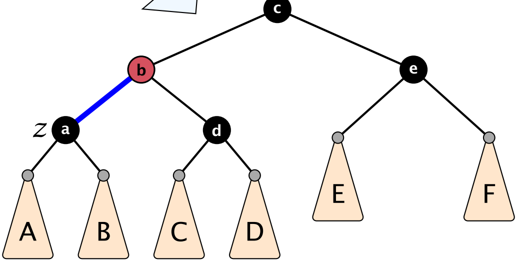
1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4



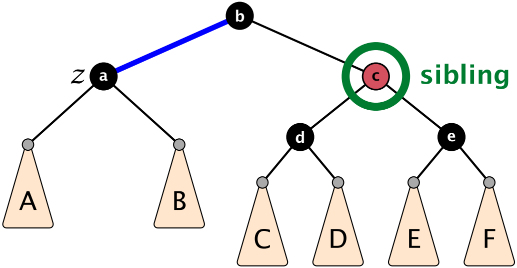
Case 1: Sibling of z is red



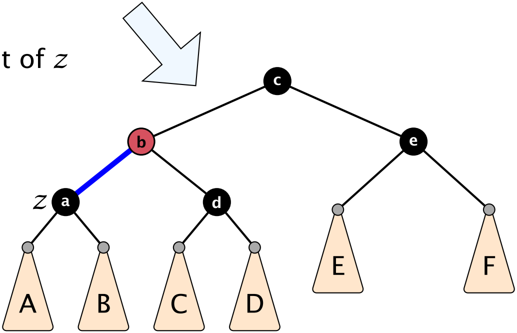
1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black
   (and parent of z is red)
4. Case 2 (special),
   or Case 3, or Case 4



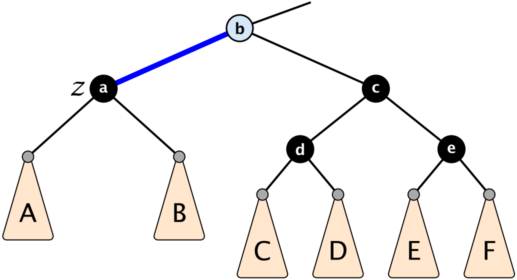
Case 1: Sibling of z is red



1. left-rotate around parent of z
2. recolor nodes b and c
3. the new sibling is black (and parent of z is red)
4. Case 2 (special), or Case 3, or Case 4

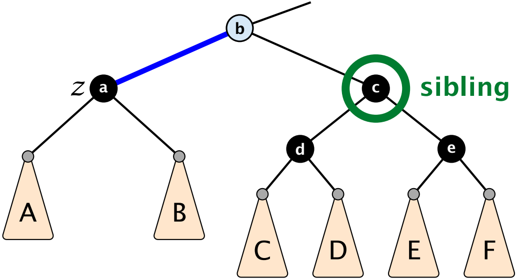


Case 2: Sibling is black with two black children



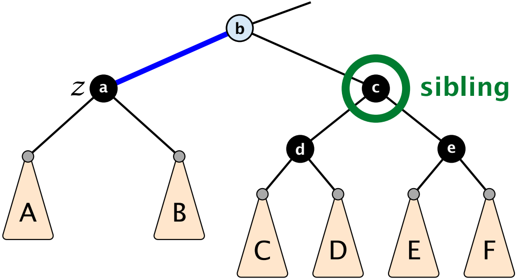
1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done

Case 2: Sibling is black with two black children

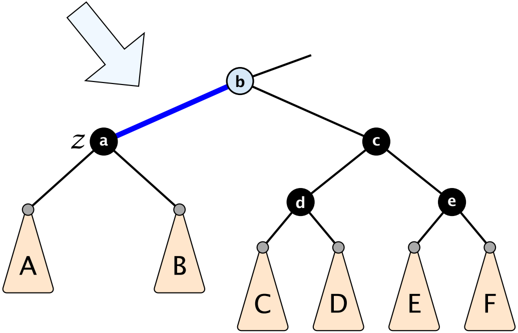


1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done

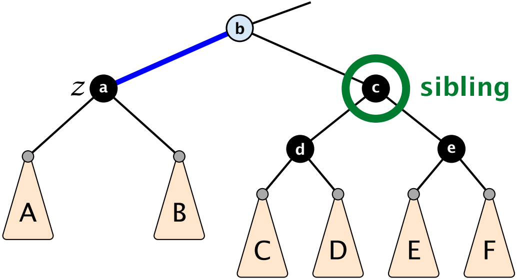
Case 2: Sibling is black with two black children



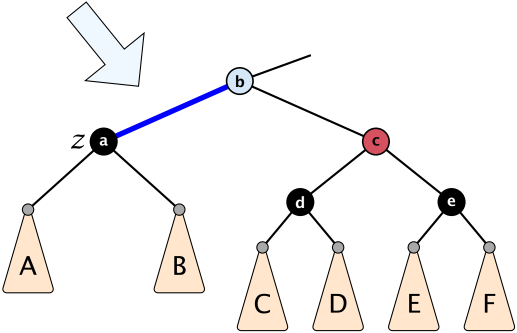
1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done



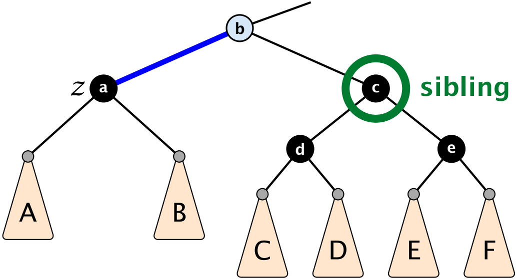
Case 2: Sibling is black with two black children



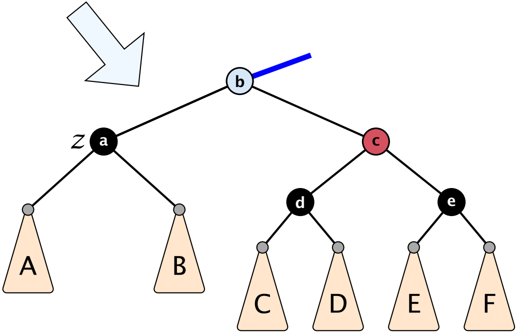
1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done



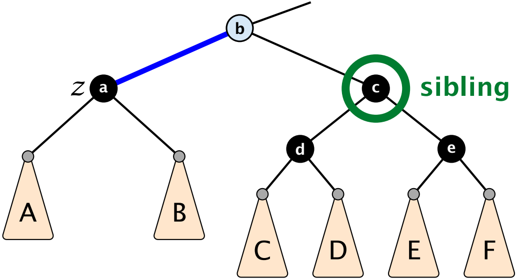
Case 2: Sibling is black with two black children



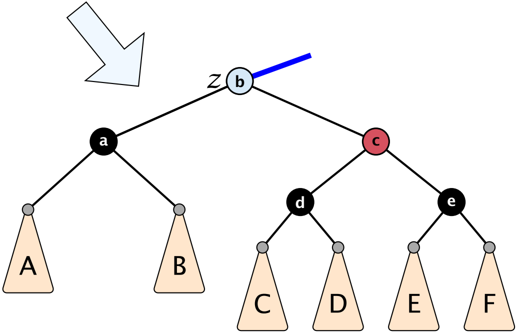
1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done



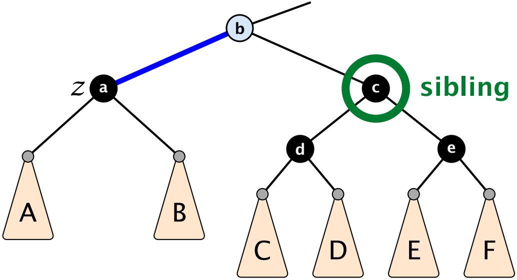
Case 2: Sibling is black with two black children



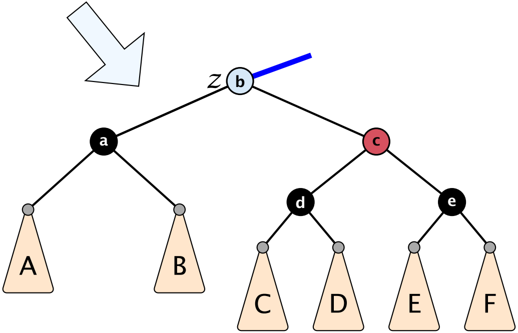
1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done



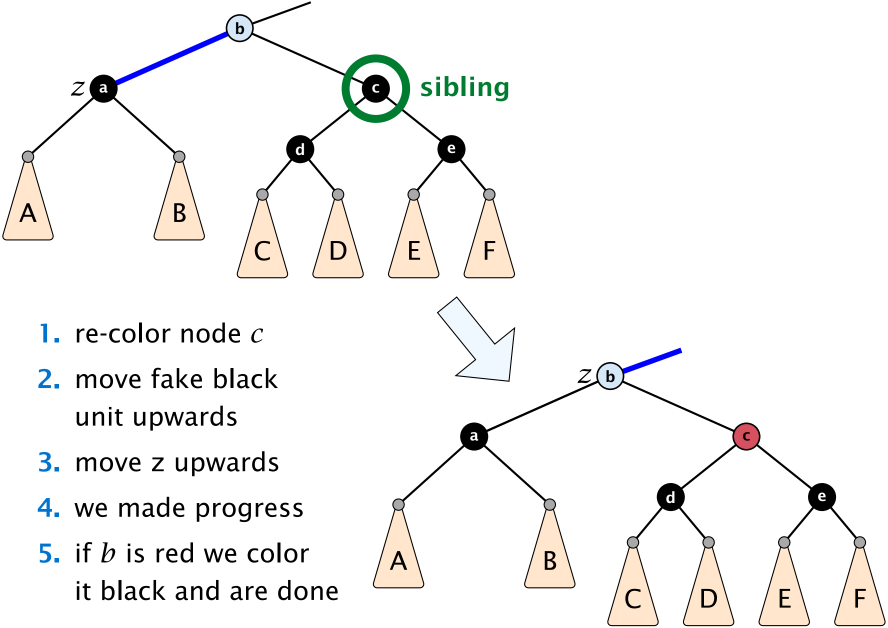
Case 2: Sibling is black with two black children



1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done



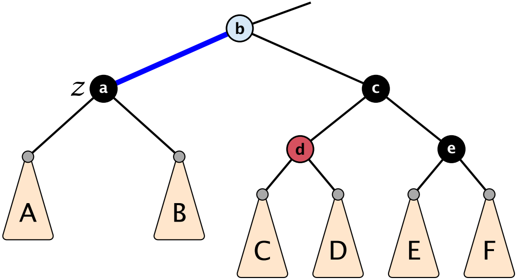
Case 2: Sibling is black with two black children



1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done

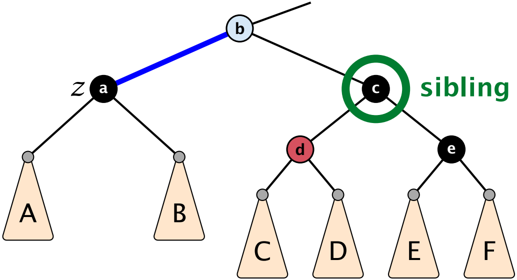
Case 3: Sibling black with one black child to the right

1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)



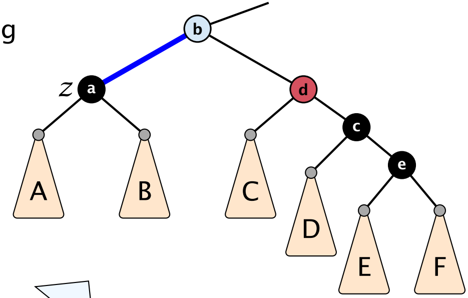
Case 3: Sibling black with one black child to the right

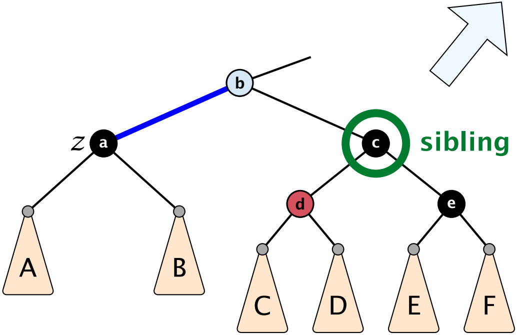
1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)



Case 3: Sibling black with one black child to the right

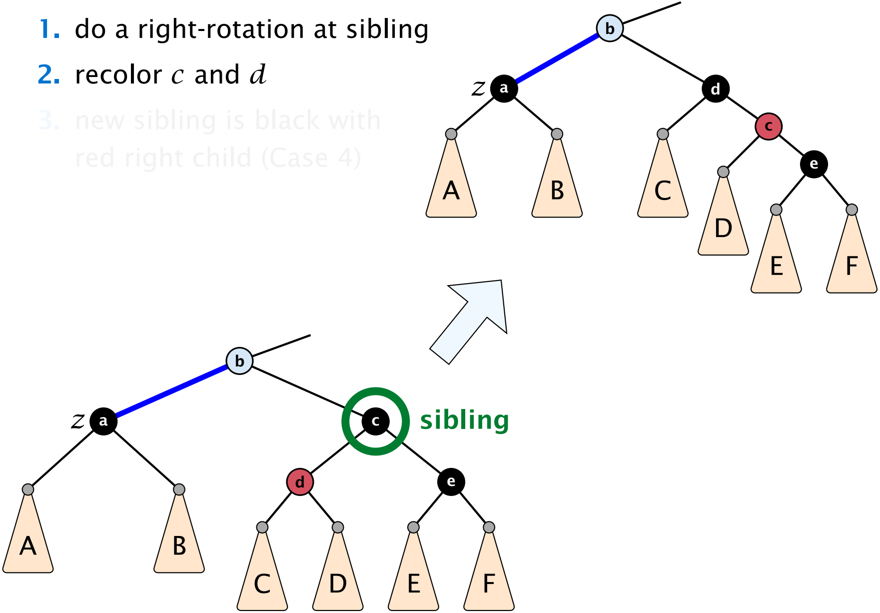
1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)





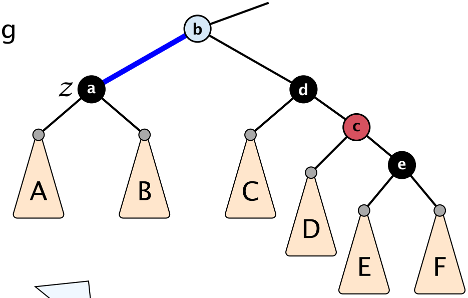
Case 3: Sibling black with one black child to the right

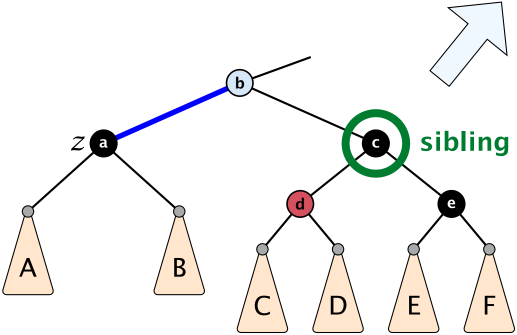
1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)



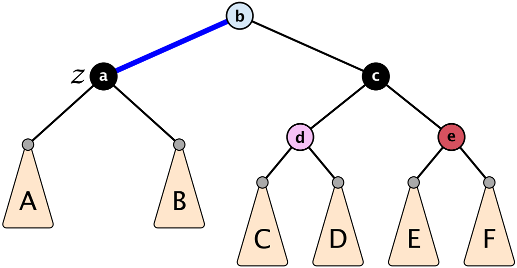
Case 3: Sibling black with one black child to the right

1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)



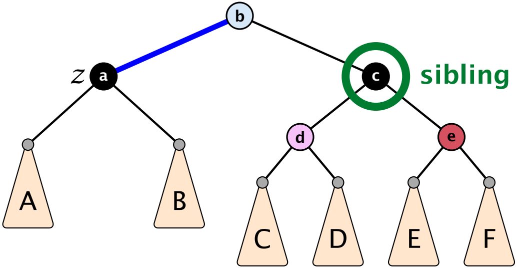


Case 4: Sibling is black with red right child



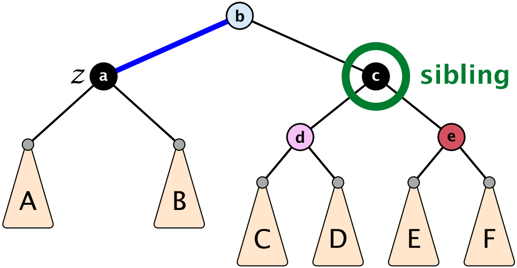
1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid red black tree

Case 4: Sibling is black with red right child

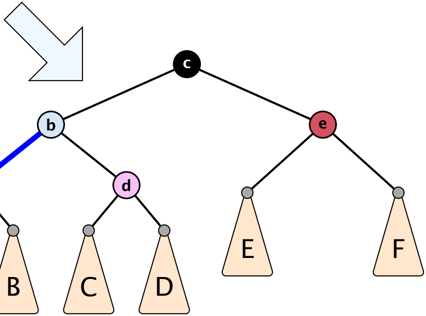


1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid red black tree

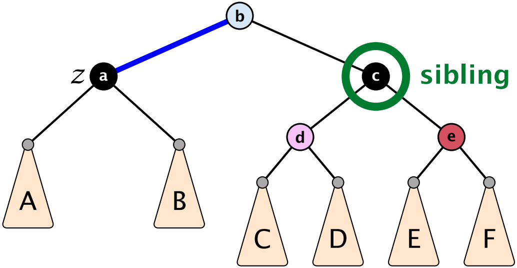
Case 4: Sibling is black with red right child



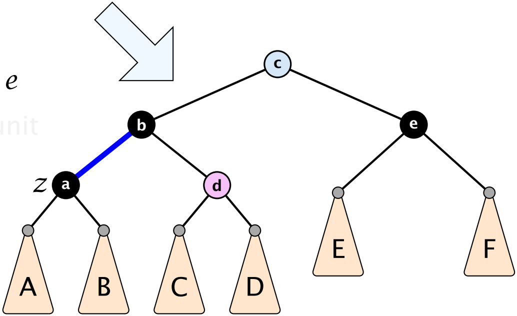
1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid red black tree



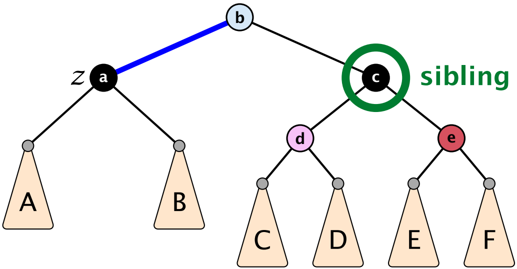
Case 4: Sibling is black with red right child



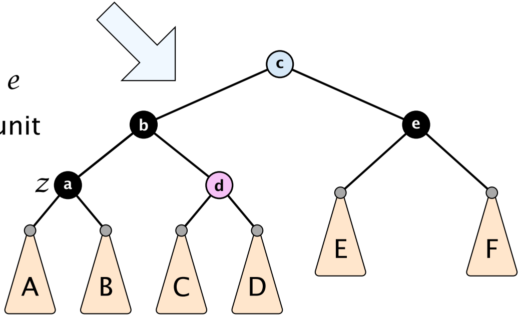
1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid
   red black tree



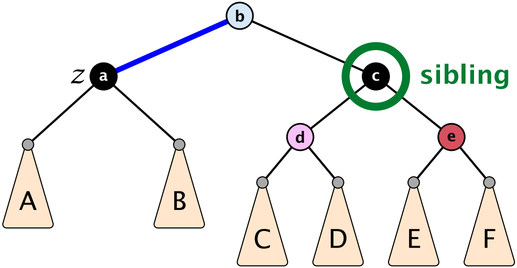
Case 4: Sibling is black with red right child



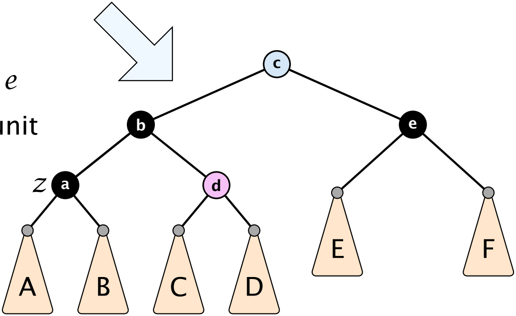
1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid red black tree



Case 4: Sibling is black with red right child



1. left-rotate around b
2. recolor nodes b, c, and e
3. remove the fake black unit
4. you have a valid
   red black tree



Running time:

- only Case 2 can repeat; but only h many steps, where h is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 at most $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colorings and at most 3 rotations.

Running time:

- only Case 2 can repeat; but only h many steps, where h is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 at most $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colorings and at most 3 rotations.

Running time:

- only Case 2 can repeat; but only h many steps, where h is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 at most $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colorings and at most 3 rotations.

Running time:

- only Case 2 can repeat; but only h many steps, where h is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 at most $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colorings and at most 3 rotations.

Running time:

- only Case 2 can repeat; but only h many steps, where h is the height of the tree
- Case 1 → Case 2 (special) → red black tree
  Case 1 → Case 3 → Case 4 → red black tree
  Case 1 → Case 4 → red black tree
- Case 3 → Case 4 → red black tree
- Case 4 → red black tree

Performing Case 2 at most $\mathcal{O}(\log n)$ times and every other step at most once, we get a red black tree. Hence, $\mathcal{O}(\log n)$ re-colorings and at most 3 rotations.

7.3 AVL-Trees

Definition 6

AVL-trees are binary search trees that fulfill the following balance condition. For every node v

$$|\text{height}(\text{left sub-tree}(v)) - \text{height}(\text{right sub-tree}(v))| \leq 1 \ .$$

Lemma 7

An AVL-tree of height h contains at least $F_{h+2} - 1$ and at most $2^h - 1$ internal nodes, where F_n is the n-th Fibonacci number ($F_0 = 0$, $F_1 = 1$), and the height is the maximal number of edges from the root to an (empty) dummy leaf.

7.3 AVL-Trees

Definition 6

AVL-trees are binary search trees that fulfill the following balance condition. For every node v

$$|\text{height}(\text{left sub-tree}(v)) - \text{height}(\text{right sub-tree}(v))| \leq 1 \ .$$

Lemma 7

An AVL-tree of height h contains at least $F_{h+2} - 1$ and at most $2^h - 1$ internal nodes, where F_n is the n-th Fibonacci number ($F_0 = 0$, $F_1 = 1$), and the height is the maximal number of edges from the root to an (empty) dummy leaf.

Proof.

The upper bound is clear, as a binary tree of height h can only contain

$$\sum_{j=0}^{h-1} 2^j = 2^h - 1$$

internal nodes.

Proof (cont.)

Induction (base cases):

1. an AVL-tree of height $h = 1$ contains at least one internal node, $1 \geq F_3 - 1 = 2 - 1 = 1$.
2. an AVL tree of height $h = 2$ contains at least two internal nodes, $2 \geq F_4 - 1 = 3 - 1 = 2$

Proof (cont.)

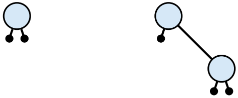
Induction (base cases):

1. an AVL-tree of height $h = 1$ contains at least one internal node, $1 \geq F_3 - 1 = 2 - 1 = 1$.
2. an AVL tree of height $h = 2$ contains at least two internal nodes, $2 \geq F_4 - 1 = 3 - 1 = 2$

Proof (cont.)

Induction (base cases):

1. an AVL-tree of height $h = 1$ contains at least one internal node, $1 \geq F_3 - 1 = 2 - 1 = 1$.
2. an AVL tree of height $h = 2$ contains at least two internal nodes, $2 \geq F_4 - 1 = 3 - 1 = 2$

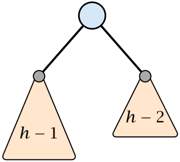


Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.

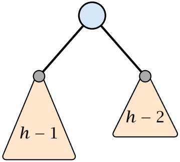
Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.

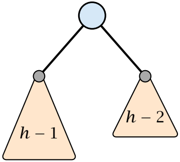


Let

$$g_h := 1 + \text{minimal size of AVL-tree of height } h \ .$$

Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h - 1$ and $h - 2$, respectively. Both, sub-trees have minmal node number.



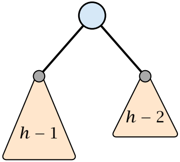
Let

$$g_h := 1 + \text{minimal size of AVL-tree of height } h \ .$$

Then

Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



Let

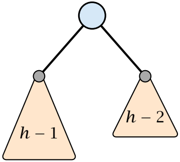
$$g_h := 1 + \text{minimal size of AVL-tree of height } h \; .$$

Then

$$g_1 = 2 \qquad\qquad = F_3$$

Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



Let

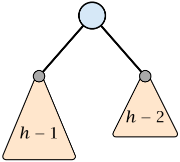
$$g_h := 1 + \text{minimal size of AVL-tree of height } h \ .$$

Then

$$\begin{aligned} g_1 &= 2 && = F_3 \\ g_2 &= 3 && = F_4 \end{aligned}$$

Induction step:

An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



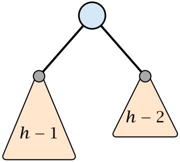
Let

$$g_h := 1 + \text{minimal size of AVL-tree of height } h \ .$$

Then

$$\begin{aligned} g_1 &= 2 && = F_3 \\ g_2 &= 3 && = F_4 \\ g_h - 1 &= 1 + g_{h-1} - 1 + g_{h-2} - 1 \,, && \text{hence} \end{aligned}$$

Induction step:
An AVL-tree of height $h \geq 2$ of minimal size has a root with sub-trees of height $h-1$ and $h-2$, respectively. Both, sub-trees have minmal node number.



Let

$$g_h := 1 + \text{minimal size of AVL-tree of height } h\ .$$

Then

$$\begin{aligned} g_1 &= 2 && = F_3 \\ g_2 &= 3 && = F_4 \\ g_h - 1 &= 1 + g_{h-1} - 1 + g_{h-2} - 1\,, && \text{hence} \\ g_h &= g_{h-1} + g_{h-2} && = F_{h+2} \end{aligned}$$

7.3 AVL-Trees

An AVL-tree of height h contains at least $F_{h+2} - 1$ internal nodes. Since

$$n + 1 \geq F_{h+2} = \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^h\right) \ ,$$

we get

$$n \geq \Omega\left(\left(\frac{1+\sqrt{5}}{2}\right)^h\right) \ ,$$

and, hence, $h = \mathcal{O}(\log n)$.

7.3 AVL-Trees

We need to maintain the balance condition through rotations.

For this we store in every internal tree-node v the balance of the node. Let v denote a tree node with left child c_ℓ and right child c_r.

$$\text{balance}[v] := \text{height}(T_{c_\ell}) - \text{height}(T_{c_r}) \ ,$$

where T_{c_ℓ} and T_{c_r}, are the sub-trees rooted at c_ℓ and c_r, respectively.

7.3 AVL-Trees

We need to maintain the balance condition through rotations.

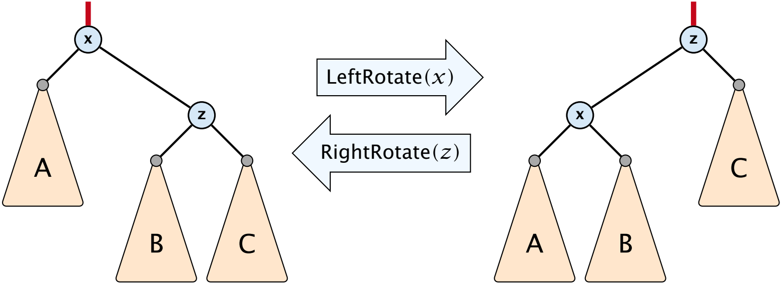
For this we store in every internal tree-node v the balance of the node. Let v denote a tree node with left child c_ℓ and right child c_r.

$$\text{balance}[v] := \text{height}(T_{c_\ell}) - \text{height}(T_{c_r}) \ ,$$

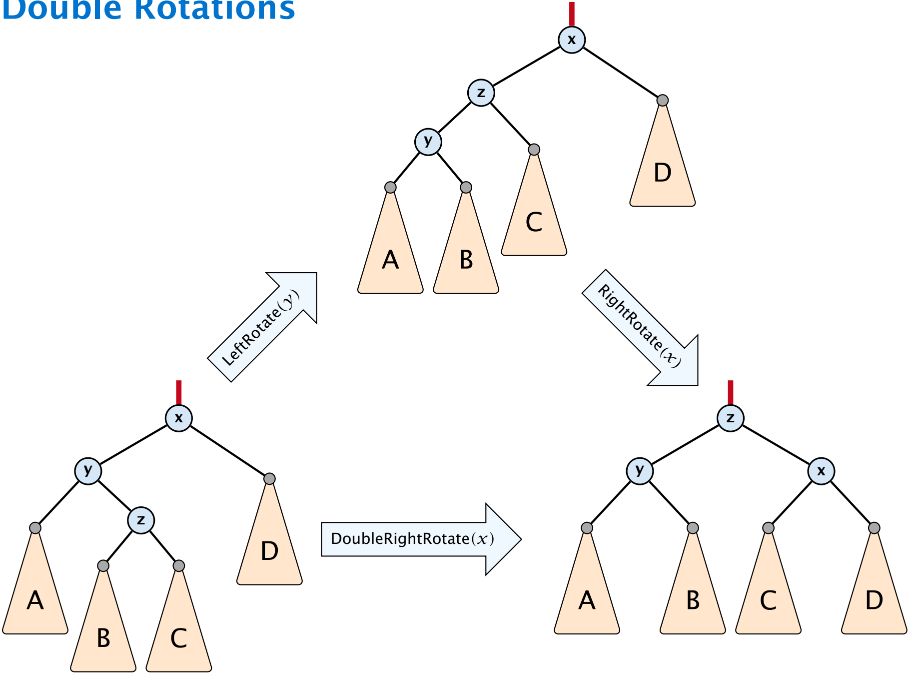
where T_{c_ℓ} and T_{c_r}, are the sub-trees rooted at c_ℓ and c_r, respectively.

Rotations

The properties will be maintained through rotations:



Double Rotations



AVL-trees: Insert

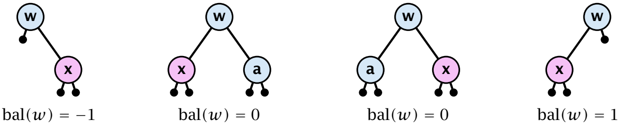
- Insert like in a binary search tree.

AVL-trees: Insert

- Insert like in a binary search tree.
- Let w denote the parent of the newly inserted node x.

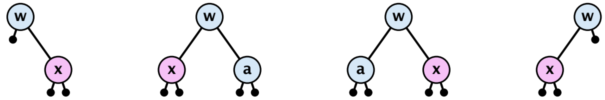
AVL-trees: Insert

- Insert like in a binary search tree.
- Let w denote the parent of the newly inserted node x.
- One of the following cases holds:



AVL-trees: Insert

- Insert like in a binary search tree.
- Let w denote the parent of the newly inserted node x.
- One of the following cases holds:

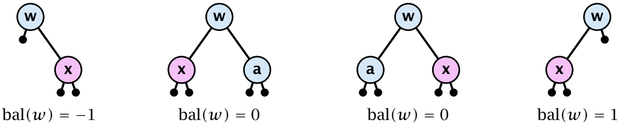


$\text{bal}(w) = -1$ $\quad$ $\text{bal}(w) = 0$ $\quad$ $\text{bal}(w) = 0$ $\quad$ $\text{bal}(w) = 1$

- If $\text{bal}[w] \neq 0$, T_w has changed height; the balance-constraint may be violated at ancestors of w.

AVL-trees: Insert

- Insert like in a binary search tree.
- Let w denote the parent of the newly inserted node x.
- One of the following cases holds:



- If $\mathrm{bal}[w] \neq 0$, T_w has changed height; the balance-constraint may be violated at ancestors of w.
- Call AVL-fix-up-insert$(\mathrm{parent}[w])$ to restore the balance-condition.

AVL-trees: Insert

Invariant at the beginning of AVL-fix-up-insert(v):

1. The balance constraints hold at all descendants of v.
2. A node has been inserted into T_c, where c is either the right or left child of v.
3. T_c has increased its height by one (otw. we would already have aborted the fix-up procedure).
4. The balance at node c fulfills $\text{balance}[c] \in \{-1, 1\}$. This holds because if the balance of c is 0, then T_c did not change its height, and the whole procedure would have been aborted in the previous step.

AVL-trees: Insert

Invariant at the beginning of AVL-fix-up-insert(v):

1. The balance constraints hold at all descendants of v.
2. A node has been inserted into T_c, where c is either the right or left child of v.
3. T_c has increased its height by one (otw. we would already have aborted the fix-up procedure).
4. The balance at node c fulfills balance$[c] \in \{-1, 1\}$. This holds because if the balance of c is 0, then T_c did not change its height, and the whole procedure would have been aborted in the previous step.

AVL-trees: Insert

Invariant at the beginning of AVL-fix-up-insert(v):

1. The balance constraints hold at all descendants of v.
2. A node has been inserted into T_c, where c is either the right or left child of v.
3. T_c has increased its height by one (otw. we would already have aborted the fix-up procedure).
4. The balance at node c fulfills $\text{balance}[c] \in \{-1, 1\}$. This holds because if the balance of c is 0, then T_c did not change its height, and the whole procedure would have been aborted in the previous step.

AVL-trees: Insert

Invariant at the beginning of AVL-fix-up-insert(v):

1. The balance constraints hold at all descendants of v.
2. A node has been inserted into T_c, where c is either the right or left child of v.
3. T_c has increased its height by one (otw. we would already have aborted the fix-up procedure).
4. The balance at node c fulfills balance$[c] \in \{-1, 1\}$. This holds because if the balance of c is 0, then T_c did not change its height, and the whole procedure would have been aborted in the previous step.

AVL-trees: Insert

Invariant at the beginning of AVL-fix-up-insert(v):

1. The balance constraints hold at all descendants of v.
2. A node has been inserted into T_c, where c is either the right or left child of v.
3. T_c has increased its height by one (otw. we would already have aborted the fix-up procedure).
4. The balance at node c fulfills $\text{balance}[c] \in \{-1, 1\}$. This holds because if the balance of c is 0, then T_c did not change its height, and the whole procedure would have been aborted in the previous step.

AVL-trees: Insert

Algorithm 11 AVL-fix-up-insert(v)

1: **if** $\mathrm{balance}[v] \in \{-2, 2\}$ **then** DoRotationInsert(v);
2: **if** $\mathrm{balance}[v] \in \{0\}$ **return**;
3: AVL-fix-up-insert($\mathrm{parent}[v]$);

We will show that the above procedure is correct, and that it will do at most one rotation.

AVL-trees: Insert

Algorithm 12 DoRotationInsert(v)

```
if balance[v] = −2 then // insert in right sub-tree
    if balance[right[v]] = −1 then
        LeftRotate(v);
    else
        DoubleLeftRotate(v);
else // insert in left sub-tree
    if balance[left[v]] = 1 then
        RightRotate(v);
    else
        DoubleRightRotate(v);
```

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

It is clear that the invariant for the fix-up routine holds as long as no rotations have been done.

We have to show that after doing one rotation **all** balance constraints are fulfilled.

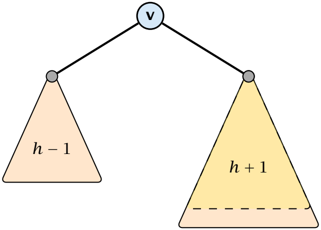
We show that after doing a rotation at v:

- v fulfills balance condition.
- All children of v still fulfill the balance condition.
- The height of T_v is the same as before the insert-operation took place.

We only look at the case where the insert happened into the right sub-tree of v. The other case is symmetric.

AVL-trees: Insert

We have the following situation:

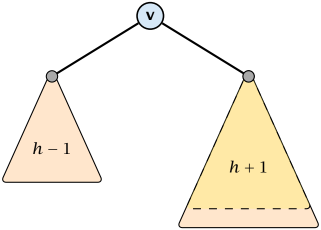


The right sub-tree of v has increased its height which results in a balance of -2 at v.

Before the insertion the height of T_v was $h+1$.

AVL-trees: Insert

We have the following situation:



The right sub-tree of v has increased its height which results in a balance of -2 at v.

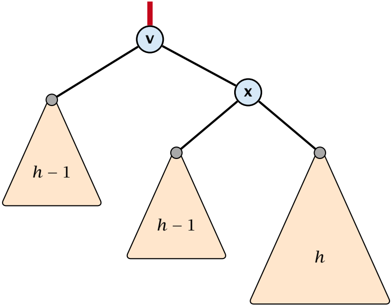
Before the insertion the height of T_v was $h+1$.

Case 1: balance[right[v]] $= -1$

We do a left rotation at v

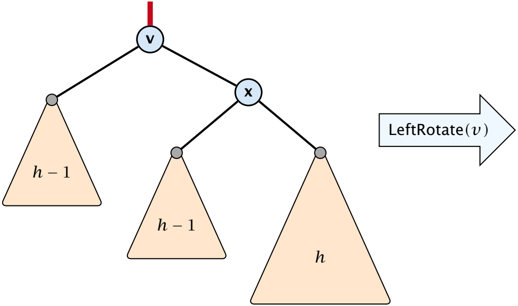
Case 1: balance[right[v]] = -1

We do a left rotation at v



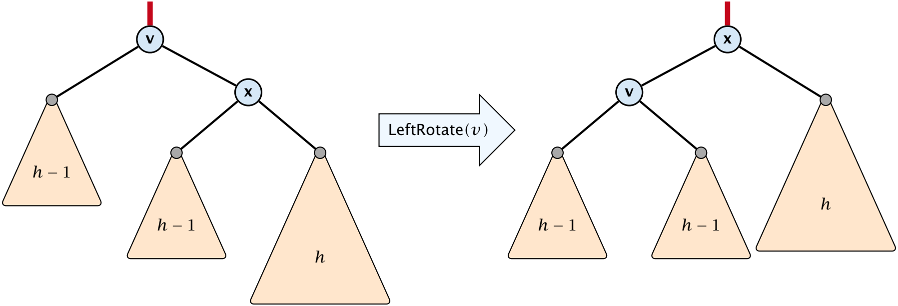
Case 1: balance[right[v]] = -1

We do a left rotation at v



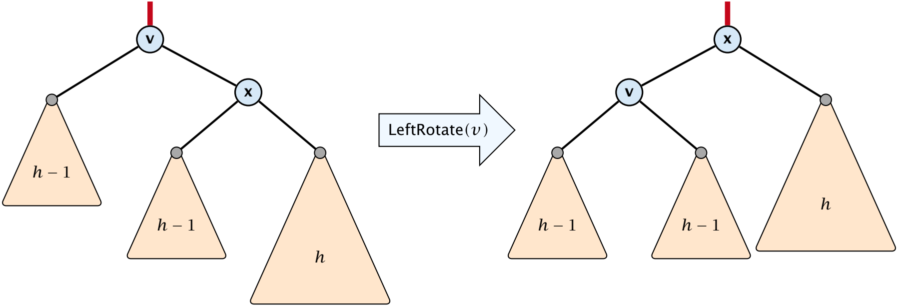
Case 1: balance[right[v]] = −1

We do a left rotation at v



Case 1: balance[right[v]] $= -1$

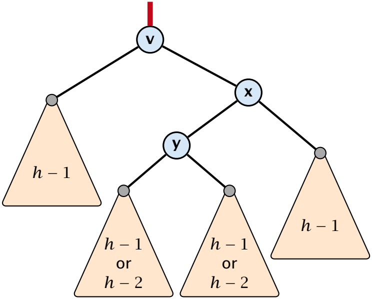
We do a left rotation at v



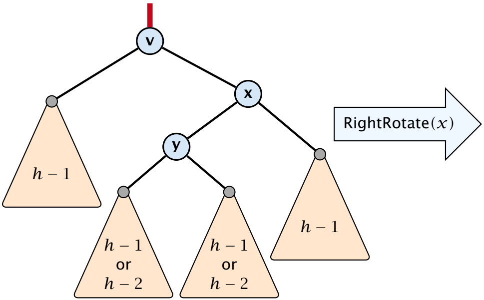
Now, the subtree has height $h + 1$ as before the insertion.
Hence, we do not need to continue.

Case 2: balance[right[v]] = 1

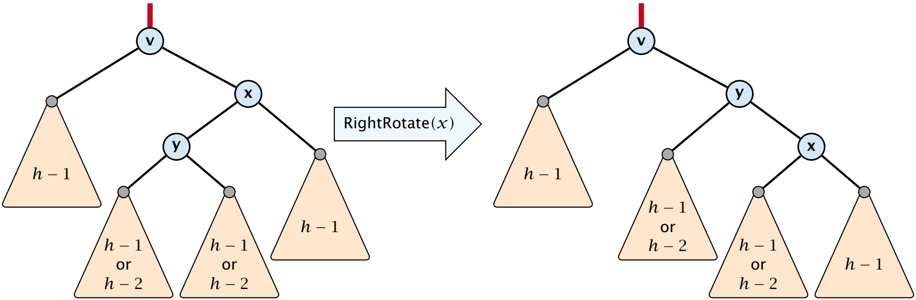
Case 2: balance[right[v]] = 1



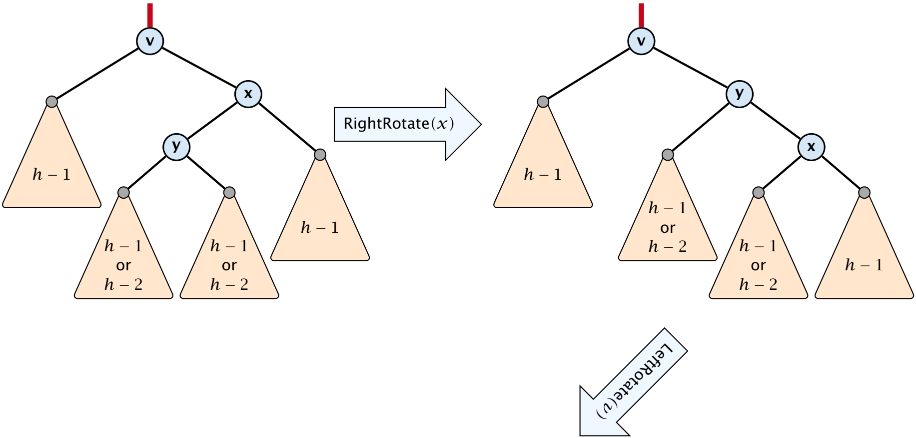
Case 2: balance[right[v]] = 1



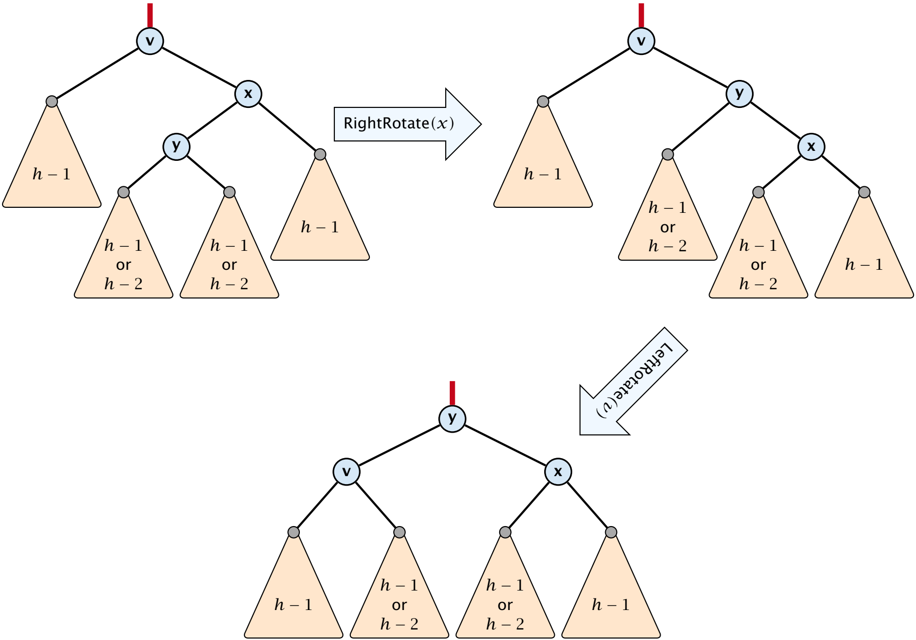
Case 2: balance[right[v]] = 1



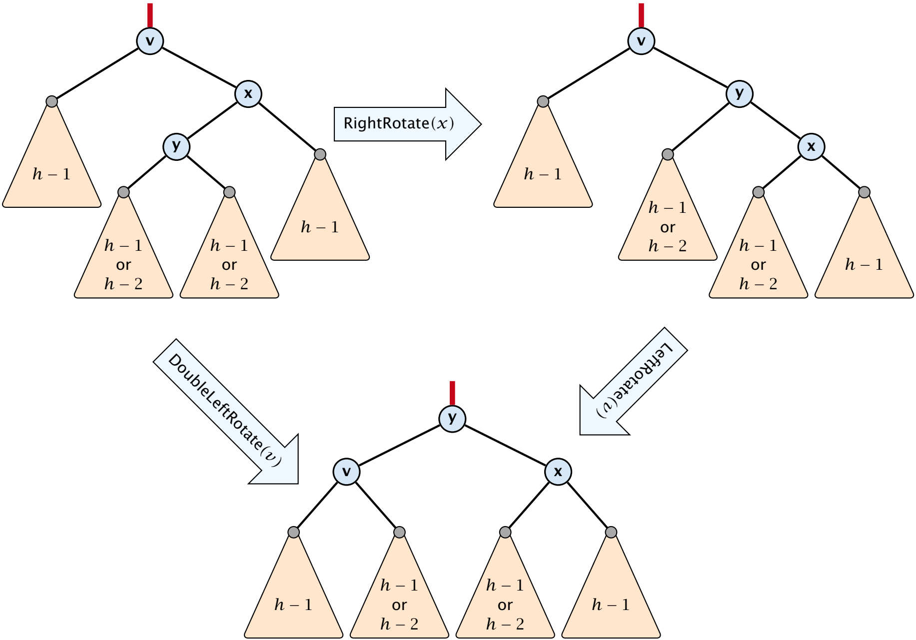
Case 2: balance[right[v]] = 1



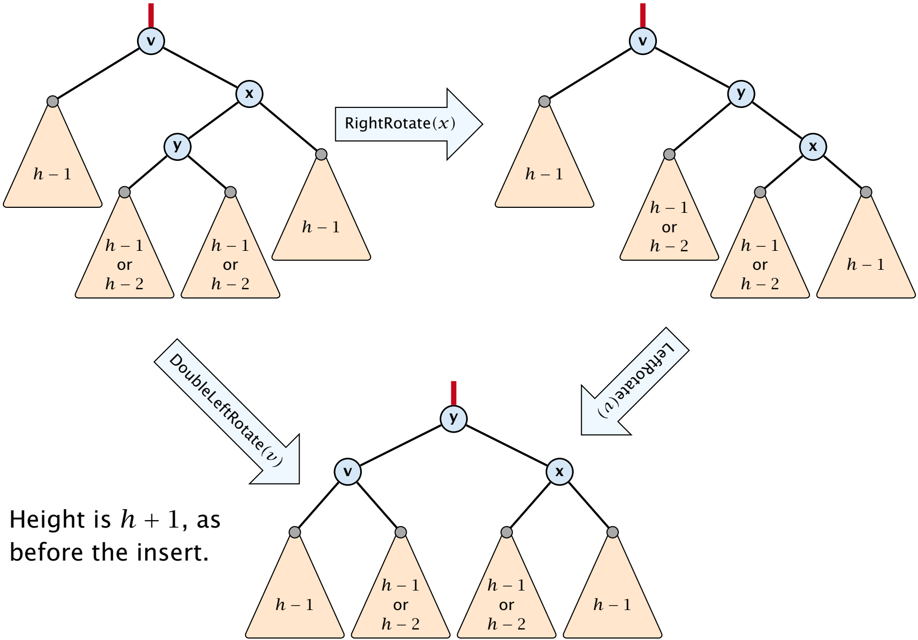
Case 2: balance[right[v]] = 1



Case 2: balance[right[v]] = 1



Case 2: balance[right[v]] = 1



Height is $h+1$, as before the insert.

AVL-trees: Delete

- Delete like in a binary search tree.
- Let v denote the parent of the node that has been spliced out.
- The balance-constraint may be violated at v, or at ancestors of v, as a sub-tree of a child of v has reduced its height.
- Initially, the node c—the new root in the sub-tree that has changed—is either a dummy leaf or a node with two dummy leafs as children.

Case 1 Case 2

In both cases $\text{bal}[c] = 0$.

- Call AVL-fix-up-delete(v) to restore the balance-condition.

AVL-trees: Delete

- Delete like in a binary search tree.
- Let v denote the parent of the node that has been spliced out.
- The balance-constraint may be violated at v, or at ancestors of v, as a sub-tree of a child of v has reduced its height.
- Initially, the node c—the new root in the sub-tree that has changed—is either a dummy leaf or a node with two dummy leafs as children.

In both cases $\mathrm{bal}[c] = 0$.

- Call AVL-fix-up-delete(v) to restore the balance-condition.

AVL-trees: Delete

- Delete like in a binary search tree.
- Let v denote the parent of the node that has been spliced out.
- The balance-constraint may be violated at v, or at ancestors of v, as a sub-tree of a child of v has reduced its height.
- Initially, the node c—the new root in the sub-tree that has changed—is either a dummy leaf or a node with two dummy leafs as children.

Case 1

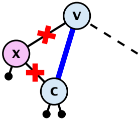
Case 2

In both cases $\text{bal}[c] = 0$.

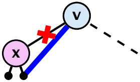
- Call AVL-fix-up-delete(v) to restore the balance-condition.

AVL-trees: Delete

- Delete like in a binary search tree.
- Let v denote the parent of the node that has been spliced out.
- The balance-constraint may be violated at v, or at ancestors of v, as a sub-tree of a child of v has reduced its height.
- Initially, the node c—the new root in the sub-tree that has changed—is either a dummy leaf or a node with two dummy leafs as children.



Case 1



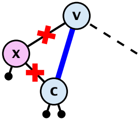
Case 2

In both cases $\mathrm{bal}[c] = 0$.

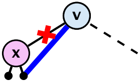
- Call AVL-fix-up-delete(v) to restore the balance-condition.

AVL-trees: Delete

- Delete like in a binary search tree.
- Let v denote the parent of the node that has been spliced out.
- The balance-constraint may be violated at v, or at ancestors of v, as a sub-tree of a child of v has reduced its height.
- Initially, the node c—the new root in the sub-tree that has changed—is either a dummy leaf or a node with two dummy leafs as children.



Case 1



Case 2

  In both cases $\text{bal}[c] = 0$.

- Call AVL-fix-up-delete(v) to restore the balance-condition.

AVL-trees: Delete

Invariant at the beginning AVL-fix-up-delete(v):

1. The balance constraints holds at all descendants of v.
2. A node has been deleted from T_c, where c is either the right or left child of v.
3. T_c has decreased its height by one.
4. The balance at the node c fulfills $\text{balance}[c] = 0$. This holds because if the balance of c is in $\{-1, 1\}$, then T_c did not change its height, and the whole procedure would have been aborted in the previous step.

AVL-trees: Delete

Invariant at the beginning AVL-fix-up-delete(v):

1. The balance constraints holds at all descendants of v.
2. A node has been deleted from T_c, where c is either the right or left child of v.
3. T_c has decreased its height by one.
4. The balance at the node c fulfills $\text{balance}[c] = 0$. This holds because if the balance of c is in $\{-1, 1\}$, then T_c did not change its height, and the whole procedure would have been aborted in the previous step.

AVL-trees: Delete

Invariant at the beginning AVL-fix-up-delete(v):

1. The balance constraints holds at all descendants of v.
2. A node has been deleted from T_c, where c is either the right or left child of v.
3. T_c has decreased its height by one.
4. The balance at the node c fulfills $\text{balance}[c] = 0$. This holds because if the balance of c is in $\{-1, 1\}$, then T_c did not change its height, and the whole procedure would have been aborted in the previous step.

AVL-trees: Delete

Invariant at the beginning AVL-fix-up-delete(v):

1. The balance constraints holds at all descendants of v.
2. A node has been deleted from T_c, where c is either the right or left child of v.
3. T_c has decreased its height by one.
4. The balance at the node c fulfills $\mathrm{balance}[c] = 0$. This holds because if the balance of c is in $\{-1, 1\}$, then T_c did not change its height, and the whole procedure would have been aborted in the previous step.

AVL-trees: Delete

Algorithm 13 AVL-fix-up-delete(v)

1: **if** balance[v] $\in \{-2, 2\}$ **then** DoRotationDelete(v);
2: **if** balance[v] $\in \{-1, 1\}$ **return**;
3: AVL-fix-up-delete(parent[v]);

We will show that the above procedure is correct. However, for the case of a delete there may be a logarithmic number of rotations.

AVL-trees: Delete

Algorithm 13 AVL-fix-up-delete(v)

1: **if** $\text{balance}[v] \in \{-2, 2\}$ **then** DoRotationDelete(v);
2: **if** $\text{balance}[v] \in \{-1, 1\}$ **return**;
3: AVL-fix-up-delete($\text{parent}[v]$);

We will show that the above procedure is correct. However, for the case of a delete there may be a logarithmic number of rotations.

AVL-trees: Delete

Algorithm 14 DoRotationDelete(v)

```
if balance[v] = −2 then // deletion in left sub-tree
    if balance[right[v]] ∈ {0, −1} then
        LeftRotate(v);
    else
        DoubleLeftRotate(v);
else // deletion in right sub-tree
    if balance[left[v]] = {0, 1} then
        RightRotate(v);
    else
        DoubleRightRotate(v);
```

AVL-trees: Delete

It is clear that the invariant for the fix-up routine hold as long as no rotations have been done.

We show that after doing a rotation at v:

- v fulfills the balance condition.
- All children of v still fulfill the balance condition.
- If now $\text{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

It is clear that the invariant for the fix-up routine hold as long as no rotations have been done.

We show that after doing a rotation at v:

- v fulfills the balance condition.
- All children of v still fulfill the balance condition.
- If now $\text{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

It is clear that the invariant for the fix-up routine hold as long as no rotations have been done.

We show that after doing a rotation at v:

- v fulfills the balance condition.
- All children of v still fulfill the balance condition.
- If now $\text{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

It is clear that the invariant for the fix-up routine hold as long as no rotations have been done.

We show that after doing a rotation at v:

- v fulfills the balance condition.
- All children of v still fulfill the balance condition.
- If now balance$[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

It is clear that the invariant for the fix-up routine hold as long as no rotations have been done.

We show that after doing a rotation at v:

- v fulfills the balance condition.
- All children of v still fulfill the balance condition.
- If now $\text{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

It is clear that the invariant for the fix-up routine hold as long as no rotations have been done.

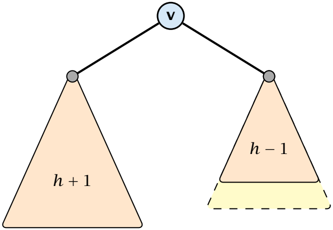
We show that after doing a rotation at v:

- v fulfills the balance condition.
- All children of v still fulfill the balance condition.
- If now $\text{balance}[v] \in \{-1, 1\}$ we can stop as the height of T_v is the same as before the deletion.

We only look at the case where the deleted node was in the right sub-tree of v. The other case is symmetric.

AVL-trees: Delete

We have the following situation:

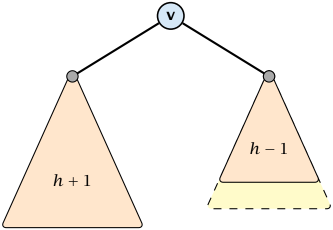


The right sub-tree of v has decreased its height which results in a balance of 2 at v.

Before the deletion the height of T_v was $h + 2$.

AVL-trees: Delete

We have the following situation:

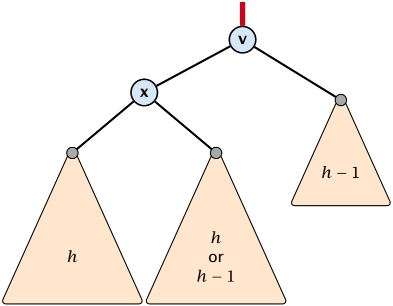


The right sub-tree of v has decreased its height which results in a balance of 2 at v.

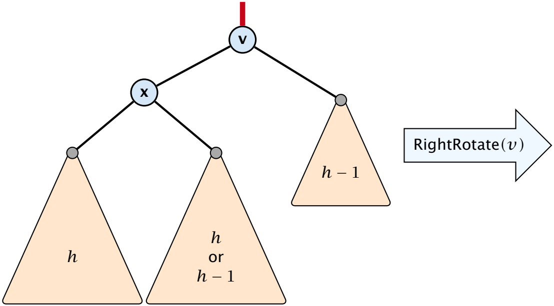
Before the deletion the height of T_v was $h + 2$.

Case 1: balance[left[v]] $\in \{0, 1\}$

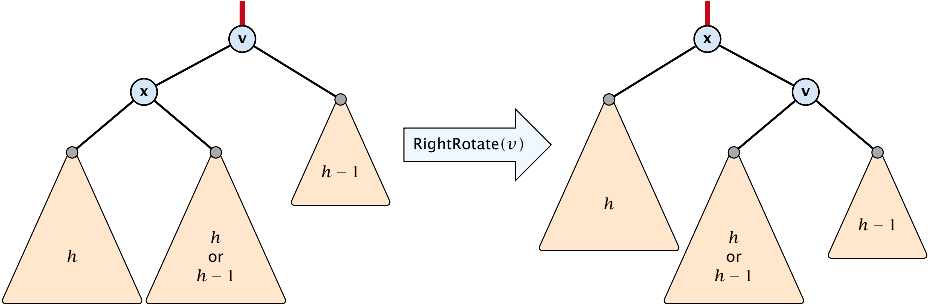
Case 1: balance[left[v]] $\in \{0, 1\}$



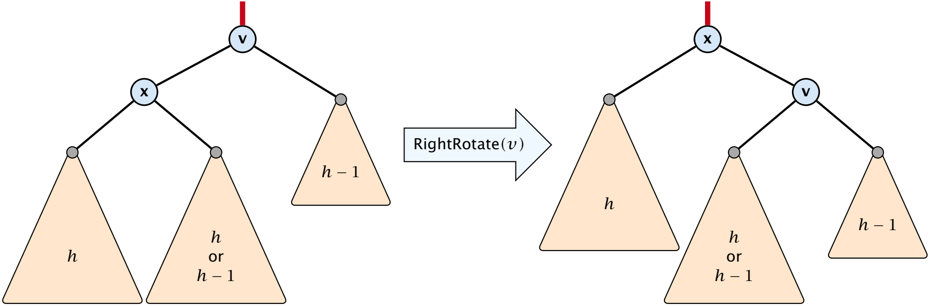
Case 1: balance[left[v]] $\in \{0, 1\}$



Case 1: balance[left[v]] $\in \{0, 1\}$

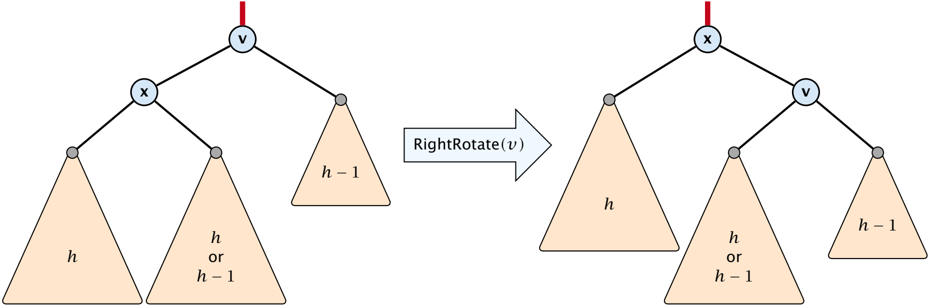


Case 1: balance[left[v]] $\in \{0, 1\}$



If the middle subtree has height h the whole tree has height $h + 2$ as before the deletion. The iteration stops as the balance at the root is non-zero.

Case 1: balance[left[v]] $\in \{0, 1\}$

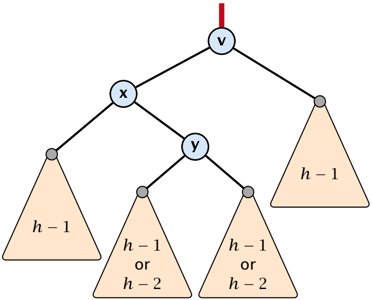


If the middle subtree has height h the whole tree has height $h + 2$ as before the deletion. The iteration stops as the balance at the root is non-zero.

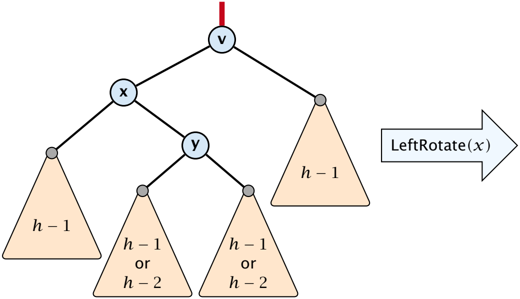
If the middle subtree has height $h - 1$ the whole tree has decreased its height from $h + 2$ to $h + 1$. We do continue the fix-up procedure as the balance at the root is zero.

Case 2: balance[left[v]] $= -1$

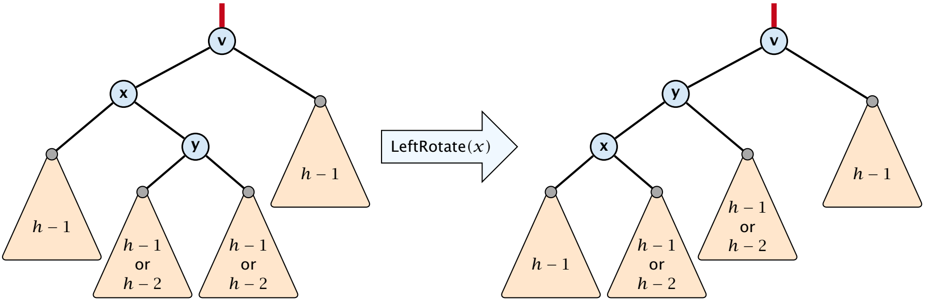
Case 2: balance[left[v]] = -1



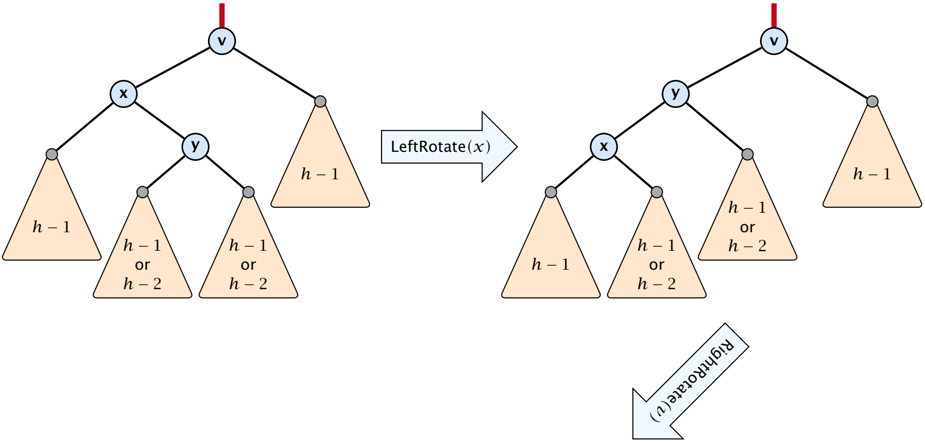
Case 2: balance[left[v]] = −1



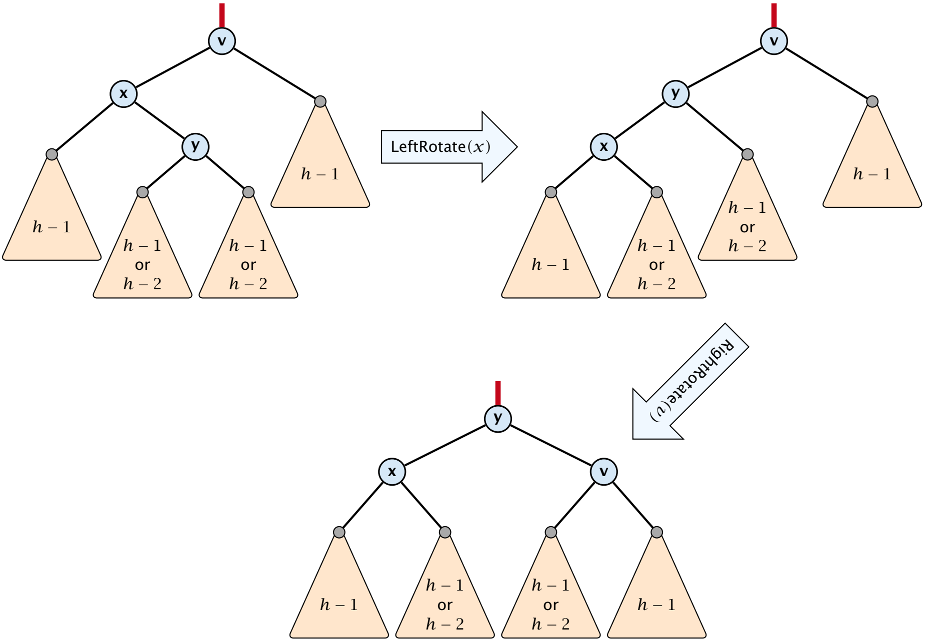
Case 2: balance[left[v]] $= -1$



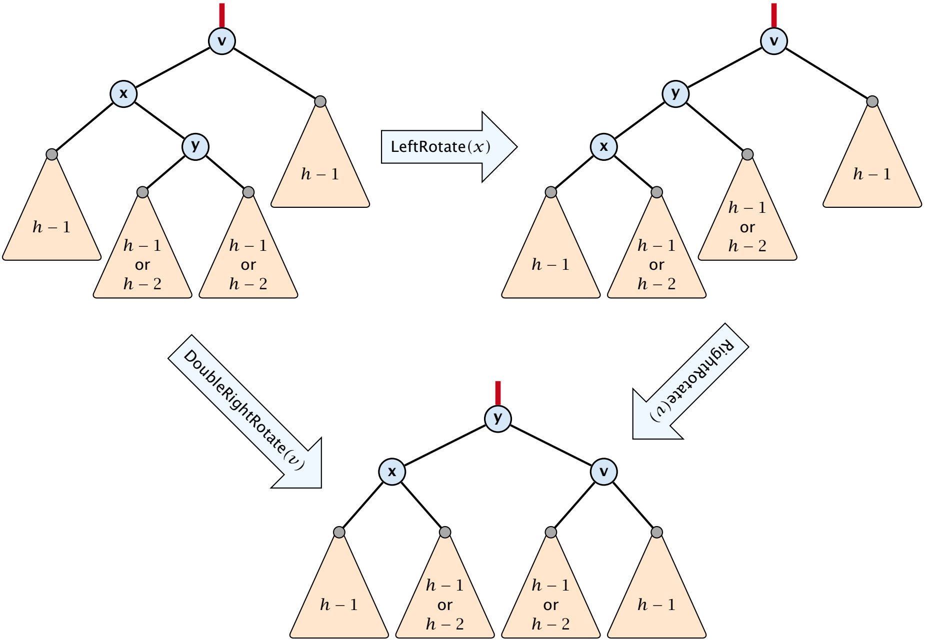
Case 2: balance[left[v]] = -1



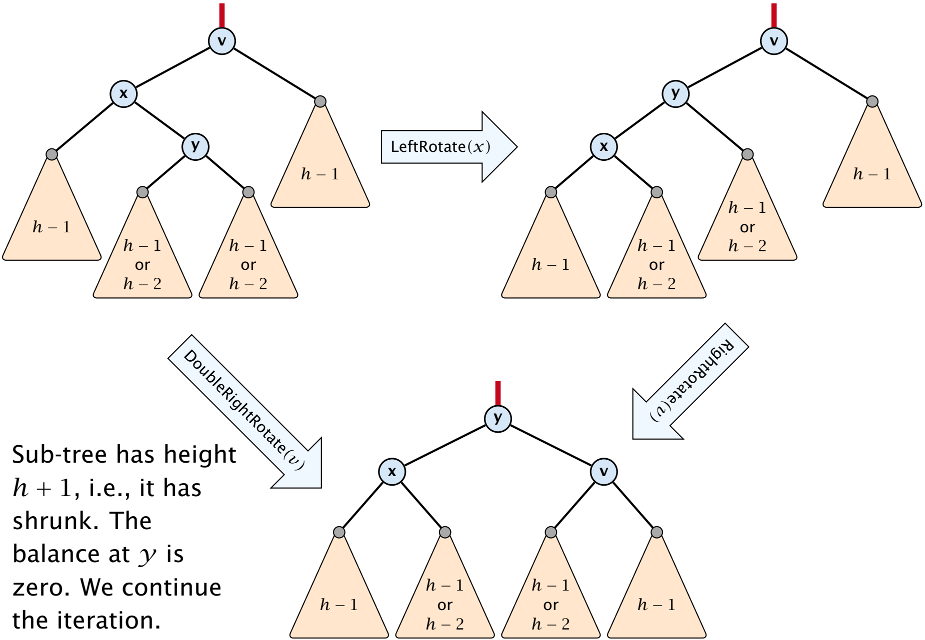
Case 2: balance[left[v]] $= -1$



Case 2: balance[left[v]] $= -1$



Case 2: balance[left[v]] = −1



Sub-tree has height $h+1$, i.e., it has shrunk. The balance at y is zero. We continue the iteration.

7.4 Augmenting Data Structures

Suppose you want to develop a data structure with:

- **Insert(x):** insert element x.
- **Search(k):** search for element with key k.
- **Delete(x):** delete element referenced by pointer x.
- **find-by-rank(ℓ):** return the ℓ-th element; return “error” if the data-structure contains less than ℓ elements.

Augment an existing data-structure instead of developing a new one.

7.4 Augmenting Data Structures

Suppose you want to develop a data structure with:

- **Insert(x):** insert element x.
- **Search(k):** search for element with key k.
- **Delete(x):** delete element referenced by pointer x.
- **find-by-rank(ℓ):** return the ℓ-th element; return “error” if the data-structure contains less than ℓ elements.

Augment an existing data-structure instead of developing a new one.

7.4 Augmenting Data Structures

How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

7.4 Augmenting Data Structures

How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

7.4 Augmenting Data Structures

How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

7.4 Augmenting Data Structures

How to augment a data-structure

1. choose an underlying data-structure
2. determine additional information to be stored in the underlying structure
3. verify/show how the additional information can be maintained for the basic modifying operations on the underlying structure.
4. develop the new operations

7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node v the size of the sub-tree rooted at v.
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...

7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node v the size of the sub-tree rooted at v.
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...

7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

1. We choose a red-black tree as the underlying data-structure.
2. We store in each node v the size of the sub-tree rooted at v.
3. We need to be able to update the size-field in each node without asymptotically affecting the running time of insert, delete, and search. We come back to this step later...

7.4 Augmenting Data Structures

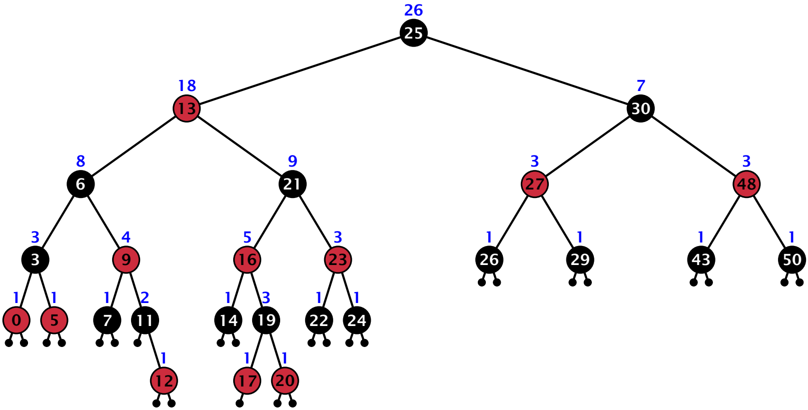
Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

4. How does find-by-rank work?
   Find-by-rank(k) ≔ Select(root, k) with

Algorithm 15 Select(x, i)

```
if x = null then return error
if left[x] ≠ null then r ← left[x].size + 1 else r ← 1
if i = r then return x
if i < r then
    return Select(left[x], i)
else
    return Select(right[x], i − r)
```

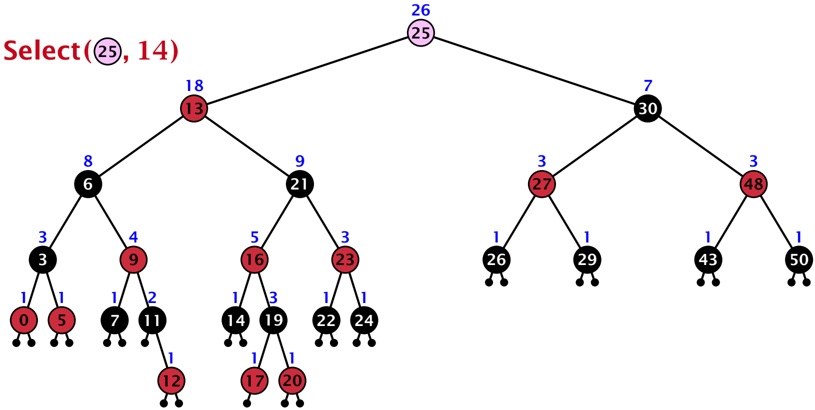
Select(*x*, *i*)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

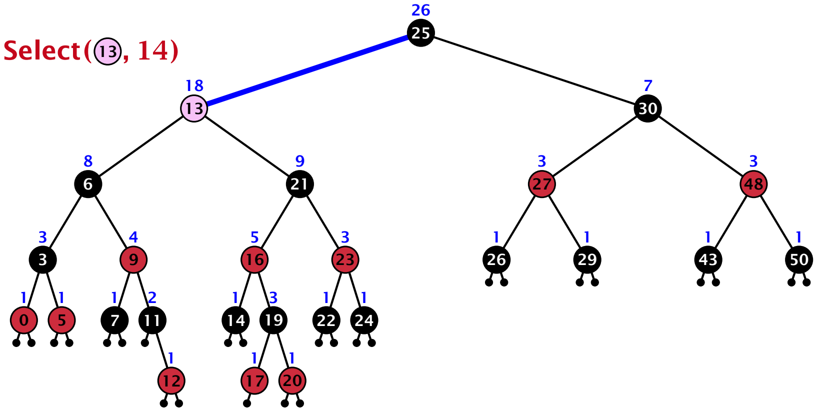
Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

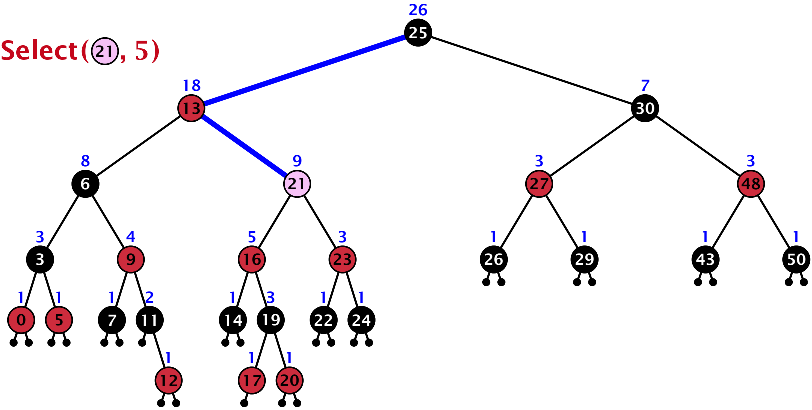
Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

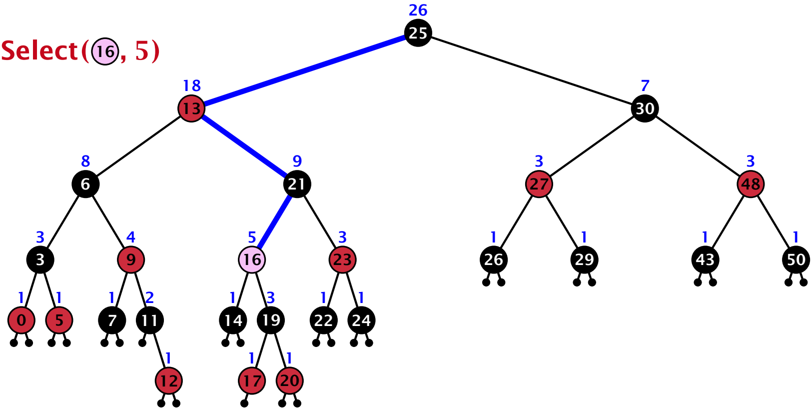
Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

Select(x, i)

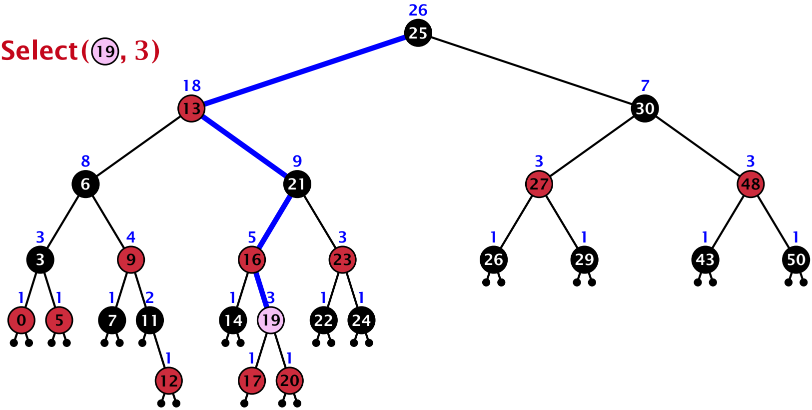


Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

Select(x, i)

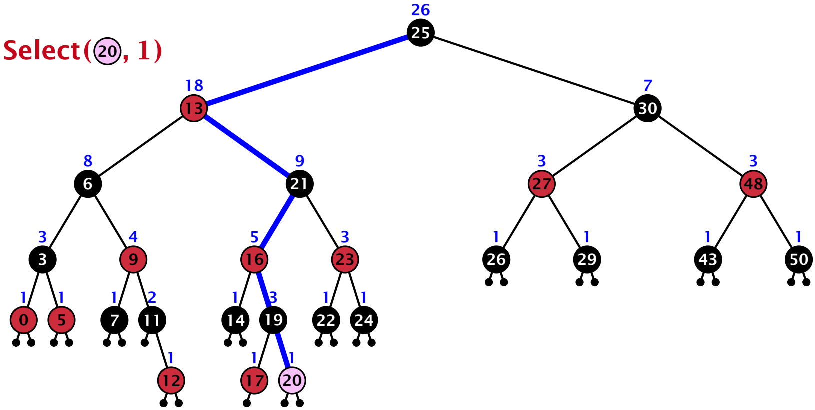
Select(19, 3)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

Select(x, i)



Find-by-rank:

- decide whether you have to proceed into the left or right sub-tree
- adjust the rank that you are searching for if you go right

7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

7.4 Augmenting Data Structures

Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

3. How do we maintain information?

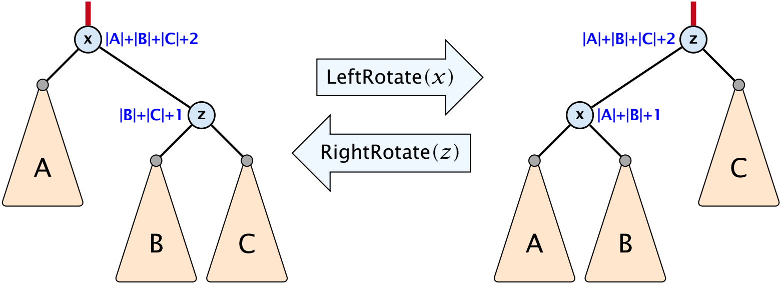
Search(k): Nothing to do.

Insert(x): When going down the search path increase the size field for each visited node. Maintain the size field during rotations.

Delete(x): Directly after splicing out a node traverse the path from the spliced out node upwards, and decrease the size counter on every node on this path. Maintain the size field during rotations.

Rotations

The only operation during the fix-up procedure that alters the tree and requires an update of the size-field:



The nodes x and z are the only nodes changing their size-fields.

The new size-fields can be computed locally from the size-fields of the children.

7.5 (a, b)-trees

Definition 8
For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.5 (a, b)-trees

Definition 8

For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.5 (a, b)-trees

Definition 8

For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.5 (a, b)-trees

Definition 8

For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.5 (a, b)-trees

Definition 8

For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.5 (a, b)-trees

Definition 8

For $b \geq 2a - 1$ an (a, b)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex v has at least a and at most b children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value ∞

7.5 (a, b)-trees

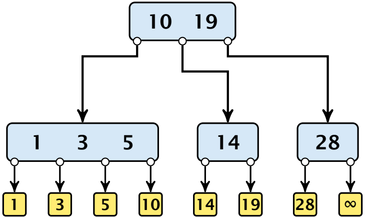
Each internal node v with $d(v)$ children stores $d - 1$ keys $k_1, \ldots, k_d - 1$. The i-th subtree of v fulfills

$$k_{i-1} < \text{ key in } i\text{-th sub-tree } \leq k_i \ ,$$

where we use $k_0 = -\infty$ and $k_d = \infty$.

7.5 (a, b)-trees

Example 9



7.5 (a, b)-trees

Variants

- The dummy leaf element may not exist; it only makes implementation more convenient.
- Variants in which $b = 2a$ are commonly referred to as B-trees.
- A B-tree usually refers to the variant in which keys and data are stored at internal nodes.
- A B^+ tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- A B^* tree requires that a node is at least $2/3$-full as opposed to $1/2$-full (the requirement of a B-tree).

7.5 (a, b)-trees

Variants

- The dummy leaf element may not exist; it only makes implementation more convenient.
- Variants in which $b = 2a$ are commonly referred to as B-trees.
- A B-tree usually refers to the variant in which keys and data are stored at internal nodes.
- A B^+ tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- A B^* tree requires that a node is at least $2/3$-full as opposed to $1/2$-full (the requirement of a B-tree).

7.5 (a, b)-trees

Variants

- The dummy leaf element may not exist; it only makes implementation more convenient.
- Variants in which $b = 2a$ are commonly referred to as B-trees.
- A B-tree usually refers to the variant in which keys and data are stored at internal nodes.
- A B^+ tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- A B^* tree requires that a node is at least $2/3$-full as opposed to $1/2$-full (the requirement of a B-tree).

7.5 (a, b)-trees

Variants

- The dummy leaf element may not exist; it only makes implementation more convenient.
- Variants in which $b = 2a$ are commonly referred to as B-trees.
- A B-tree usually refers to the variant in which keys and data are stored at internal nodes.
- A B^+ tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- A B^* tree requires that a node is at least $2/3$-full as opposed to $1/2$-full (the requirement of a B-tree).

7.5 (a, b)-trees

Variants

- The dummy leaf element may not exist; it only makes implementation more convenient.
- Variants in which $b = 2a$ are commonly referred to as B-trees.
- A B-tree usually refers to the variant in which keys and data are stored at internal nodes.
- A B^+ tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
- A B^* tree requires that a node is at least $2/3$-full as opposed to $1/2$-full (the requirement of a B-tree).

Lemma 10

Let T be an (a,b)-tree for $n > 0$ elements (i.e., $n+1$ leaf nodes) and height h (number of edges from root to a leaf vertex). Then

1. $2a^{h-1} \leq n+1 \leq b^h$
2. $\log_b(n+1) \leq h \leq 1 + \log_a(\frac{n+1}{2})$

Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least a. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most b and hence, the number of leaf nodes at most b^h.

Lemma 10

Let T be an (a, b)-tree for $n > 0$ elements (i.e., $n + 1$ leaf nodes) and height h (number of edges from root to a leaf vertex). Then

1. $2a^{h-1} \leq n + 1 \leq b^h$
2. $\log_b(n + 1) \leq h \leq 1 + \log_a(\frac{n+1}{2})$

Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least a. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most b and hence, the number of leaf nodes at most b^h.

Lemma 10

Let T be an (a, b)-tree for $n > 0$ elements (i.e., $n + 1$ leaf nodes) and height h (number of edges from root to a leaf vertex). Then

1. $2a^{h-1} \le n + 1 \le b^h$
2. $\log_b(n + 1) \le h \le 1 + \log_a(\frac{n+1}{2})$

Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least a. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most b and, hence, the number of leaf nodes at most b^h.

□

Lemma 10

Let T be an (a,b)-tree for $n > 0$ elements (i.e., $n+1$ leaf nodes) and height h (number of edges from root to a leaf vertex). Then

1. $2a^{h-1} \le n+1 \le b^h$
2. $\log_b(n+1) \le h \le 1 + \log_a(\frac{n+1}{2})$

Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least a. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most b and, hence, the number of leaf nodes at most b^h.

□

Lemma 10

Let T be an (a,b)-tree for $n > 0$ elements (i.e., $n+1$ leaf nodes) and height h (number of edges from root to a leaf vertex). Then

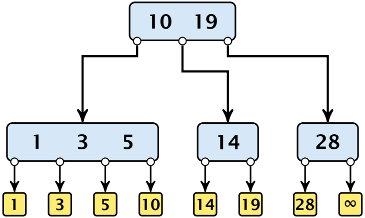
1. $2a^{h-1} \leq n+1 \leq b^h$
2. $\log_b(n+1) \leq h \leq 1 + \log_a(\frac{n+1}{2})$

Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least a. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most b and, hence, the number of leaf nodes at most b^h.

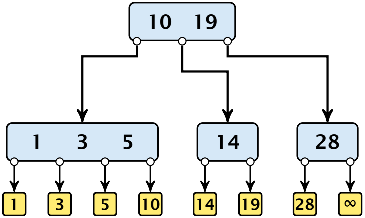
□

Search



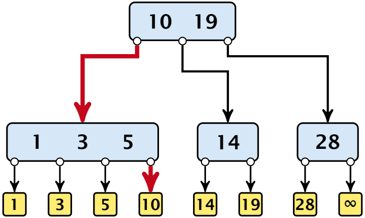
Search

Search(8)



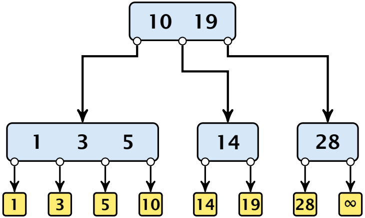
Search

Search(8)



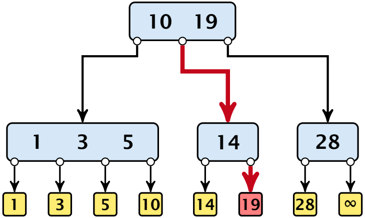
Search

Search(19)

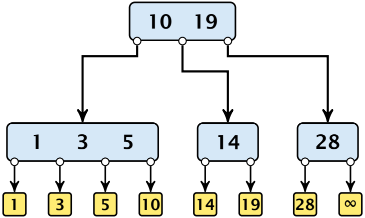


Search

Search(19)

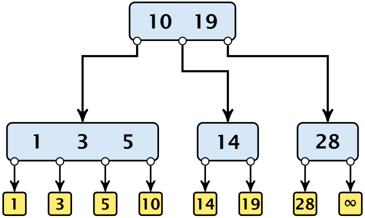


Search



The search is straightforward. It is only important that you need to go all the way to the leaf.

Search



The search is straightforward. It is only important that you need to go all the way to the leaf.

Time: $\mathcal{O}(b \cdot h) = \mathcal{O}(b \cdot \log n)$, if the individual nodes are organized as linear lists.

Insert

Insert element x:

- Follow the path as if searching for $\text{key}[x]$.
- If this search ends in leaf ℓ, insert x before this leaf.
- For this add $\text{key}[x]$ to the key-list of the last internal node v on the path.
- If after the insert v contains b nodes, do Rebalance(v).

Insert

Insert element x:

- Follow the path as if searching for $\text{key}[x]$.
- If this search ends in leaf ℓ, insert x before this leaf.
- For this add $\text{key}[x]$ to the key-list of the last internal node v on the path.
- If after the insert v contains b nodes, do Rebalance(v).

Insert

Insert element x:

- Follow the path as if searching for $\mathrm{key}[x]$.
- If this search ends in leaf ℓ, insert x before this leaf.
- For this add $\mathrm{key}[x]$ to the key-list of the last internal node v on the path.
- If after the insert v contains b nodes, do Rebalance(v).

Insert

Insert element x:

- Follow the path as if searching for $\text{key}[x]$.
- If this search ends in leaf ℓ, insert x before this leaf.
- For this add $\text{key}[x]$ to the key-list of the last internal node v on the path.
- If after the insert v contains b nodes, do Rebalance(v).

Insert

Rebalance(v):

- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

- Let k_i, $i = 1, \ldots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \ldots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \ldots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

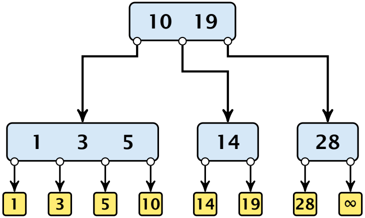
- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert

Rebalance(v):

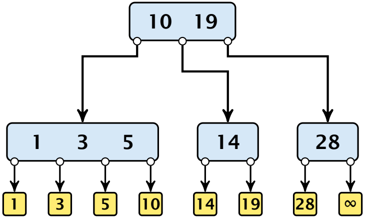
- Let k_i, $i = 1, \dots, b$ denote the keys stored in v.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes v_1, and v_2. v_1 gets all keys $k_1, \dots, k_{j-1}$ and v_2 gets keys $k_{j+1}, \dots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key k_j is promoted to the parent of v. The current pointer to v is altered to point to v_1, and a new pointer (to the right of k_j) in the parent is added to point to v_2.
- Then, re-balance the parent.

Insert



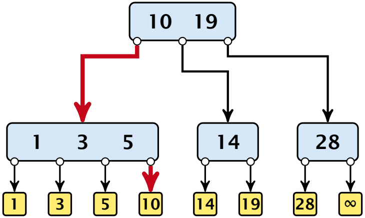
Insert

Insert(8)



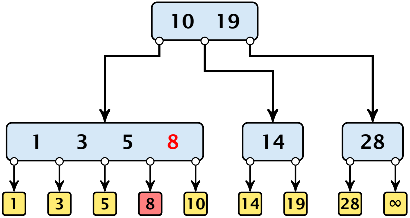
Insert

Insert(8)



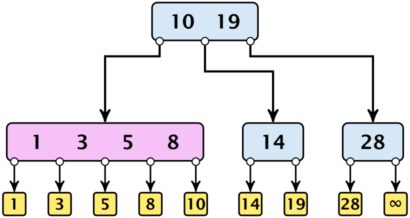
Insert

Insert(8)



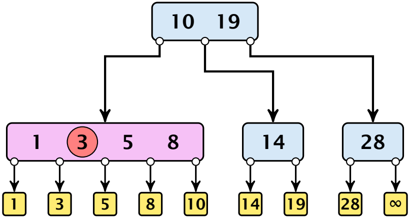
Insert

Insert(8)

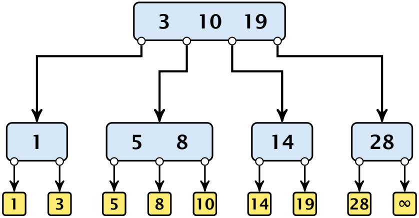


Insert

Insert(8)

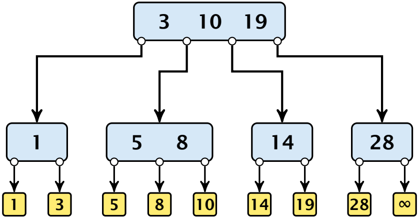


Insert



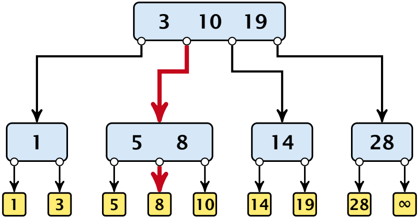
Insert

Insert(6)



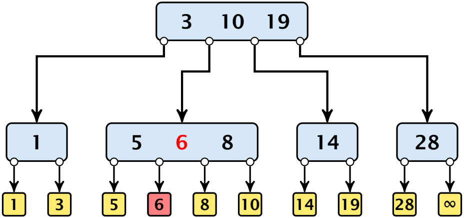
Insert

Insert(6)



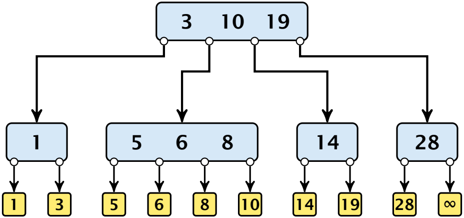
Insert

Insert(6)



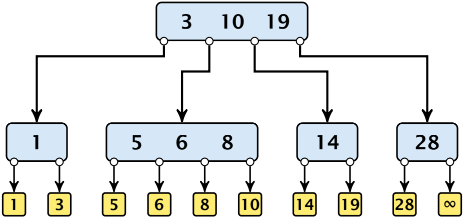
Insert

Insert(6)



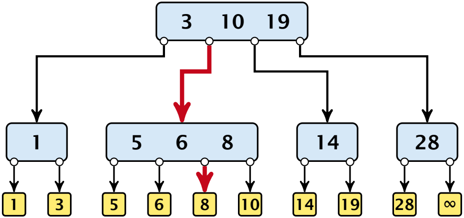
Insert

Insert(7)



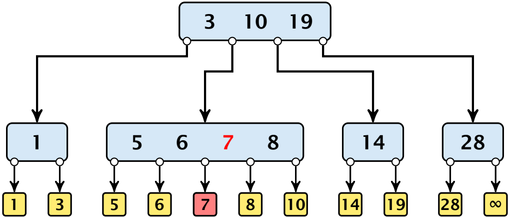
Insert

Insert(7)



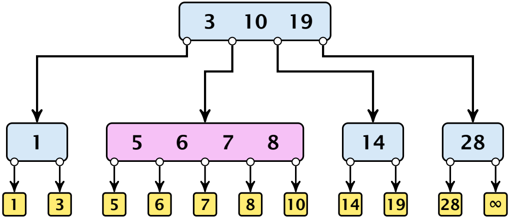
Insert

Insert(7)



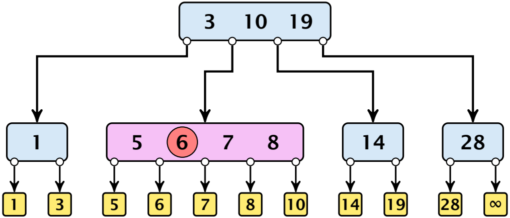
Insert

Insert(7)



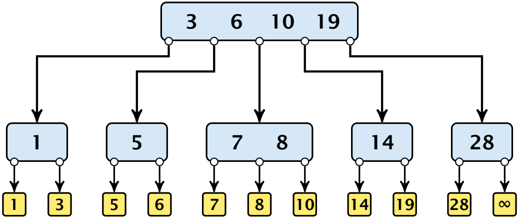
Insert

Insert(7)



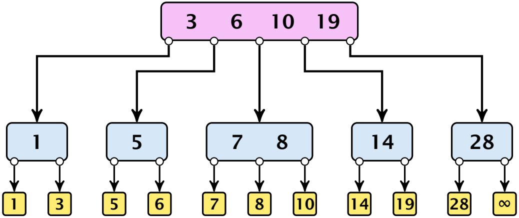
Insert

Insert(7)



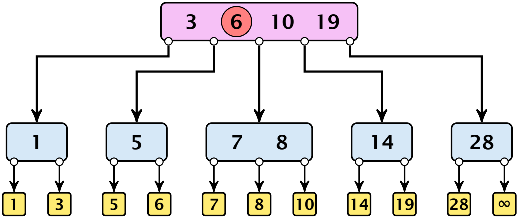
Insert

Insert(7)



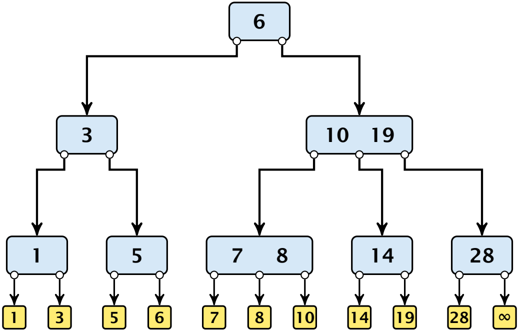
Insert

Insert(7)



Insert

Insert(7)



Delete

Delete element x (pointer to leaf vertex):

- Let v denote the parent of x. If $\mathrm{key}[x]$ is contained in v, remove the key from v, and delete the leaf vertex.
- Otherwise delete the key of the predecessor of x from v; delete the leaf vertex; and replace the occurrence of $\mathrm{key}[x]$ in internal nodes by the predecessor key. (Note that it appears in exactly one internal vertex).
- If now the number of keys in v is below $a - 1$ perform Rebalance′(v).

Delete

Delete element x (pointer to leaf vertex):

- Let v denote the parent of x. If $\mathrm{key}[x]$ is contained in v, remove the key from v, and delete the leaf vertex.
- Otherwise delete the key of the predecessor of x from v; delete the leaf vertex; and replace the occurrence of $\mathrm{key}[x]$ in internal nodes by the predecessor key. (Note that it appears in exactly one internal vertex).
- If now the number of keys in v is below $a - 1$ perform Rebalance$'(v)$.

Delete

Delete element x (pointer to leaf vertex):

- Let v denote the parent of x. If $\text{key}[x]$ is contained in v, remove the key from v, and delete the leaf vertex.
- Otherwise delete the key of the predecessor of x from v; delete the leaf vertex; and replace the occurrence of $\text{key}[x]$ in internal nodes by the predecessor key. (Note that it appears in exactly one internal vertex).
- If now the number of keys in v is below $a - 1$ perform Rebalance$'(v)$.

Delete

Rebalance$'(v)$:

- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \leq b$ successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

Delete

Rebalance$'(v)$:

- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \le b$ successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

Delete

Rebalance′(v):

- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \le b$ successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

Delete

Rebalance′(v):

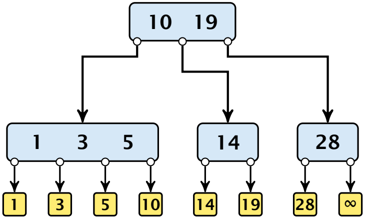
- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \le b$ successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

Delete

Rebalance′(v):

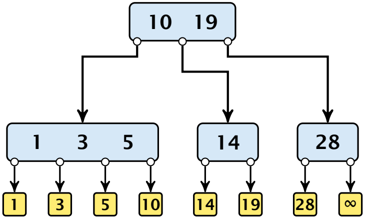
- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \le b$ successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.

Delete



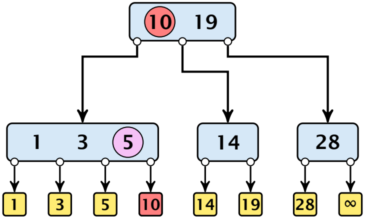
Delete

Delete(10)



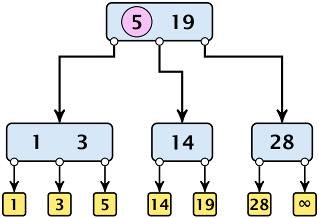
Delete

Delete(10)

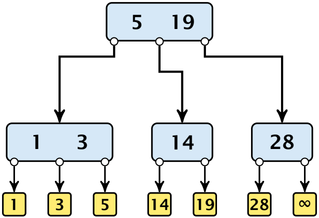


Delete

Delete(10)

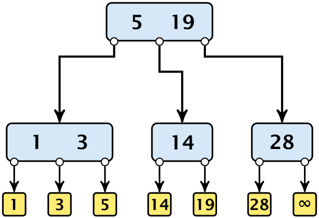


Delete



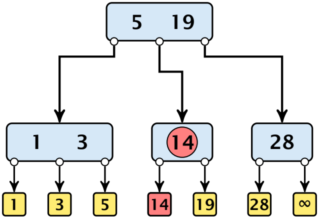
Delete

Delete(14)



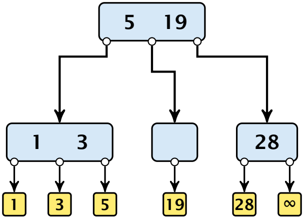
Delete

Delete(14)



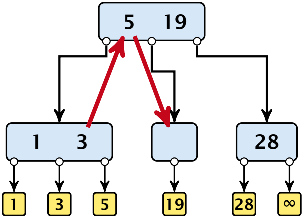
Delete

Delete(14)



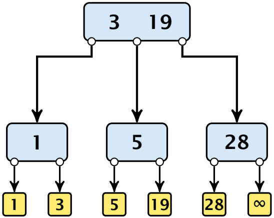
Delete

Delete(14)

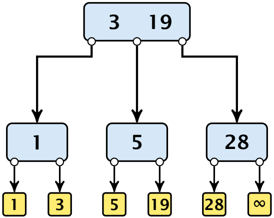


Delete

Delete(14)

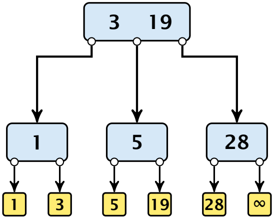


Delete



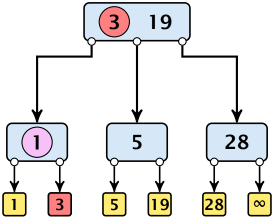
Delete

Delete(3)



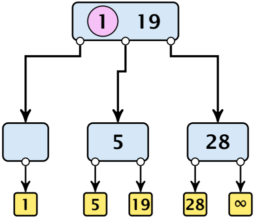
Delete

Delete(3)



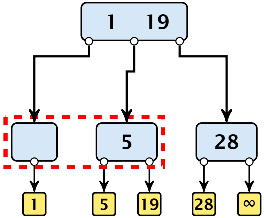
Delete

Delete(3)



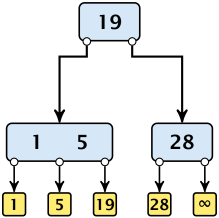
Delete

Delete(3)

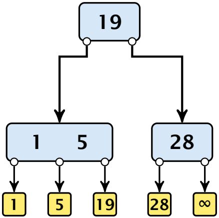


Delete

Delete(3)

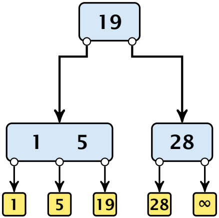


Delete



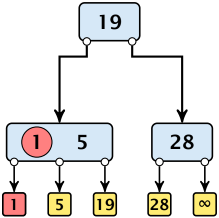
Delete

Delete(1)



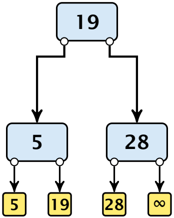
Delete

Delete(1)

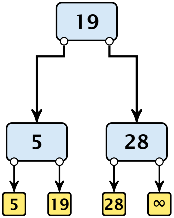


Delete

Delete(1)

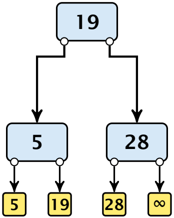


Delete



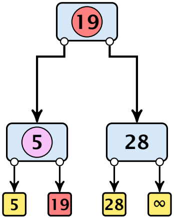
Delete

Delete(19)



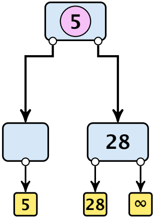
Delete

Delete(19)



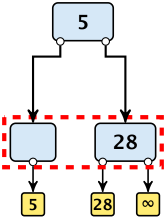
Delete

Delete(19)



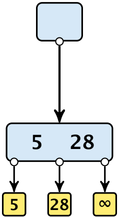
Delete

Delete(19)



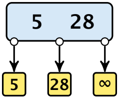
Delete

Delete(19)



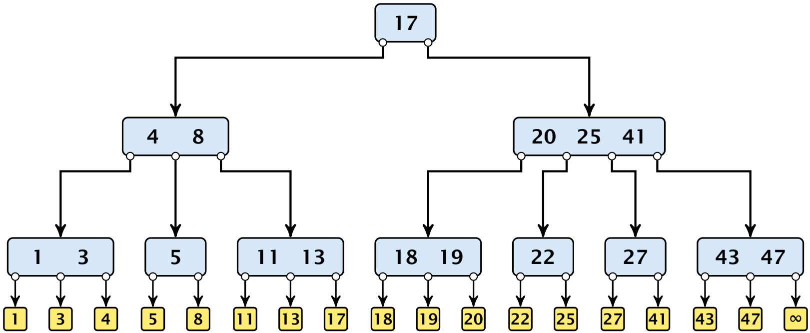
Delete

Delete(19)



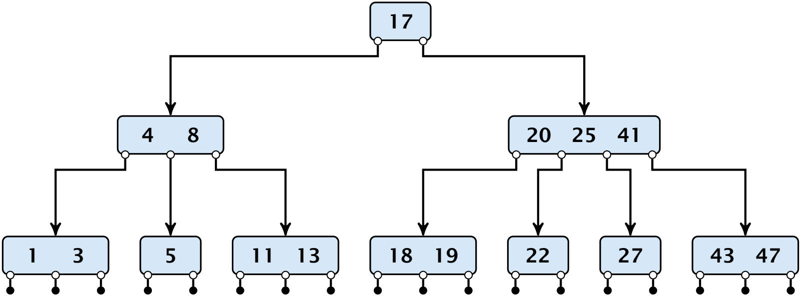
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



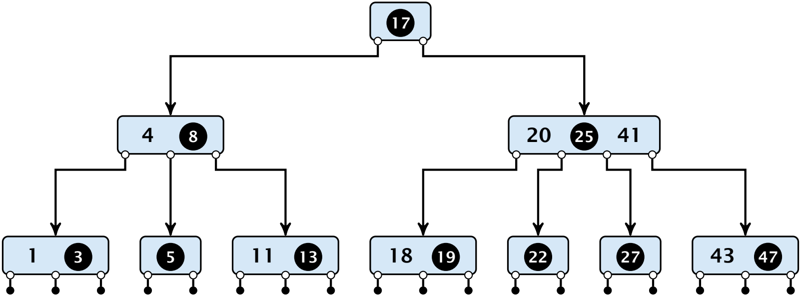
(2, 4)-trees and red black trees

There is a close relation between red-black trees and $(2, 4)$-trees:



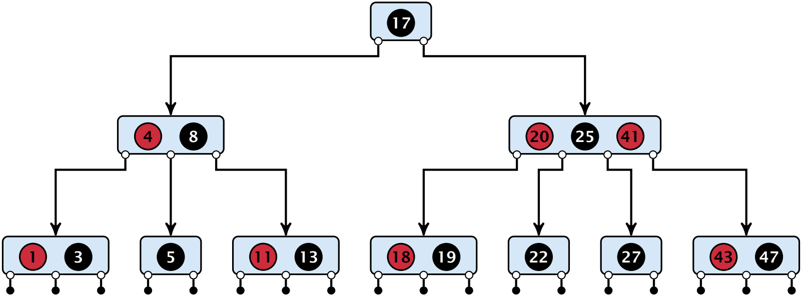
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



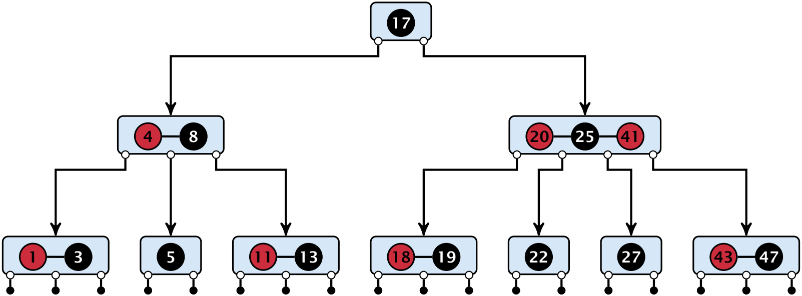
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



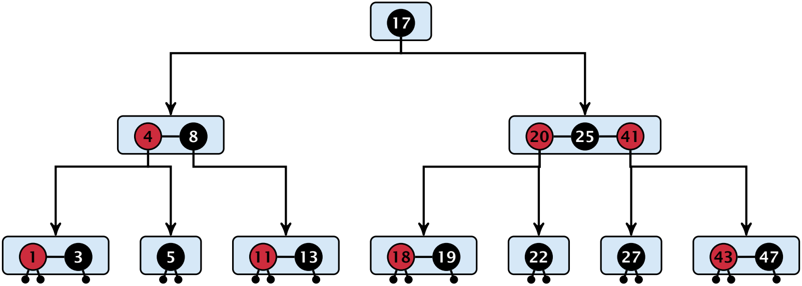
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



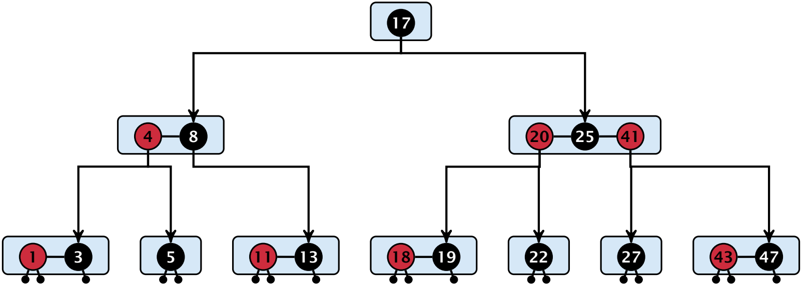
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



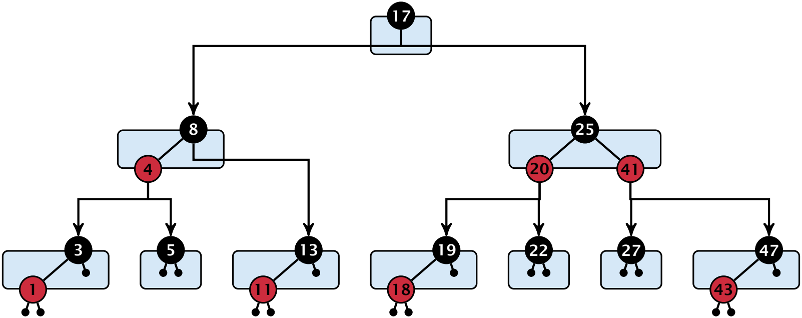
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



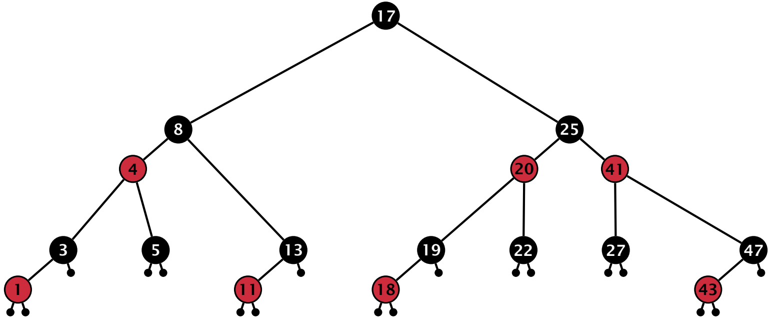
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



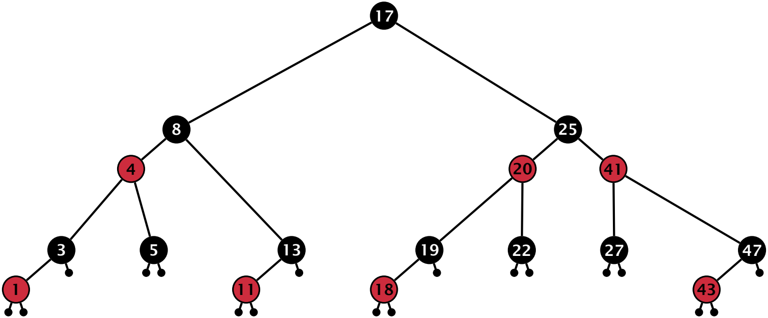
$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



$(2,4)$-trees and red black trees

There is a close relation between red-black trees and $(2,4)$-trees:



Note that this correspondence is not unique. In particular, there are different red-black trees that correspond to the same $(2,4)$-tree.

7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$

7.6 Skip Lists

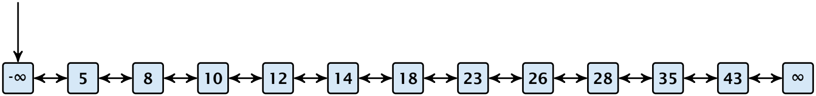
Why do we not use a list for implementing the ADT Dynamic Set?

- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$

7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

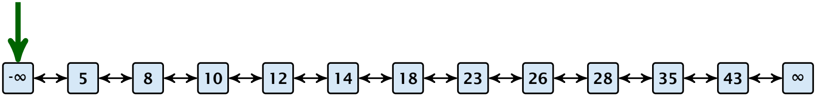
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

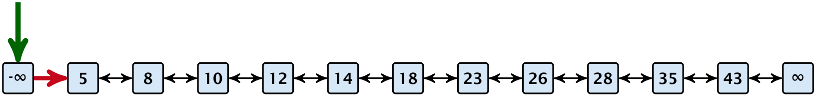
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

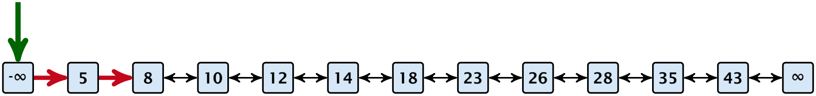
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

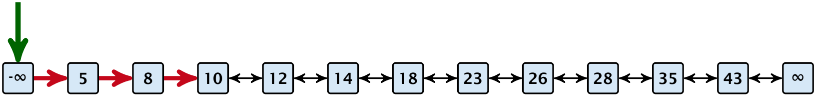
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

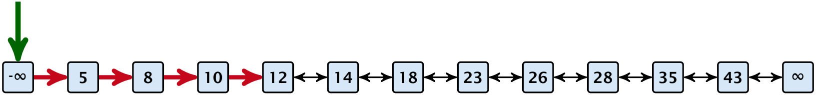
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

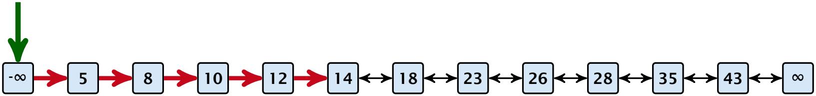
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

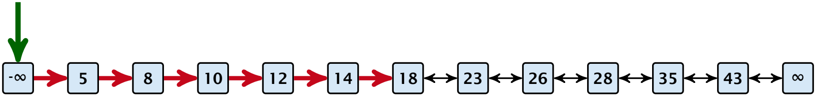
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

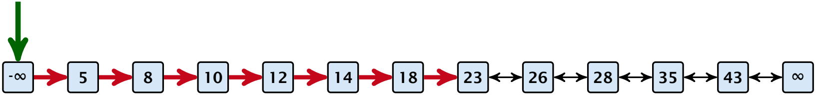
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

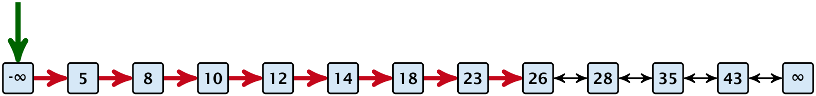
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.6 Skip Lists

Why do we not use a list for implementing the ADT Dynamic Set?

- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.6 Skip Lists

How can we improve the search-operation?

7.6 Skip Lists

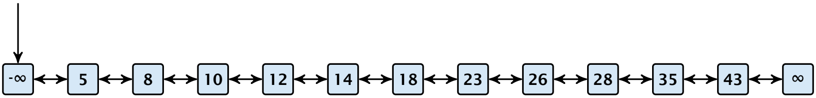
How can we improve the search-operation?

Add an express lane:

7.6 Skip Lists

How can we improve the search-operation?

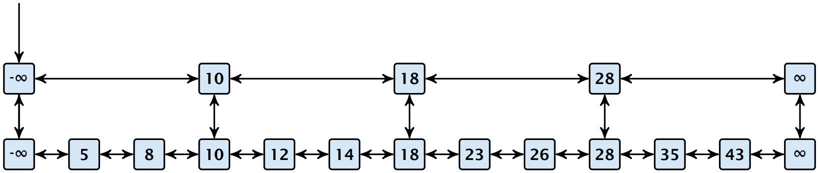
Add an express lane:



7.6 Skip Lists

How can we improve the search-operation?

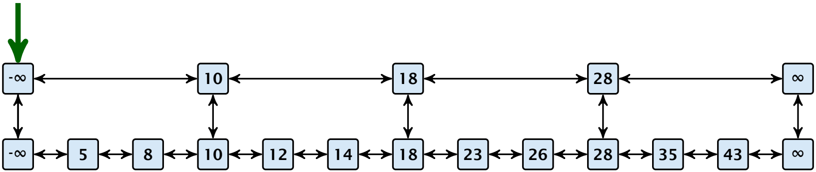
Add an express lane:



7.6 Skip Lists

How can we improve the search-operation?

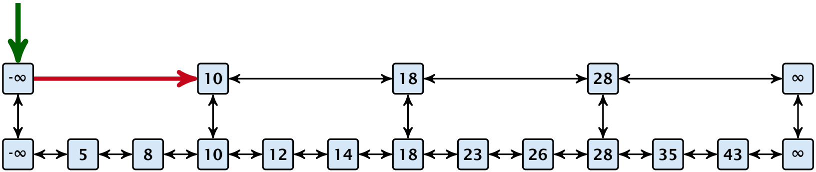
Add an express lane:



7.6 Skip Lists

How can we improve the search-operation?

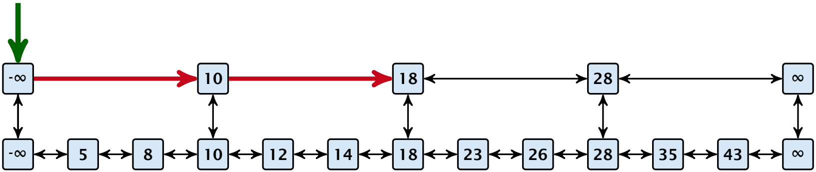
Add an express lane:



7.6 Skip Lists

How can we improve the search-operation?

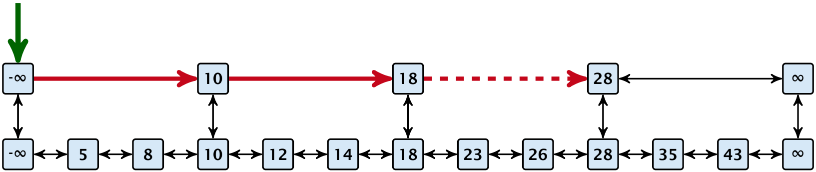
Add an express lane:



7.6 Skip Lists

How can we improve the search-operation?

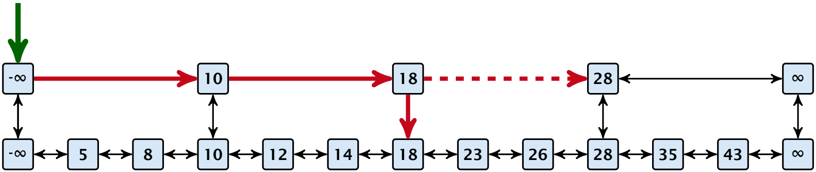
Add an express lane:



7.6 Skip Lists

How can we improve the search-operation?

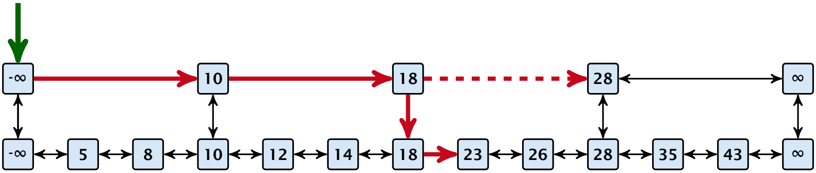
Add an express lane:



7.6 Skip Lists

How can we improve the search-operation?

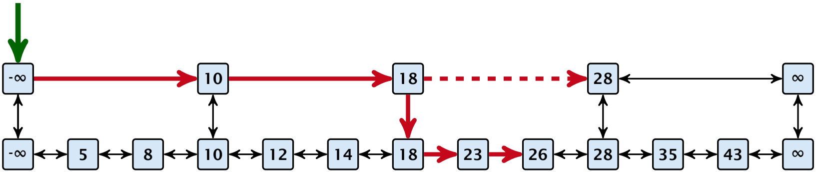
Add an express lane:



7.6 Skip Lists

How can we improve the search-operation?

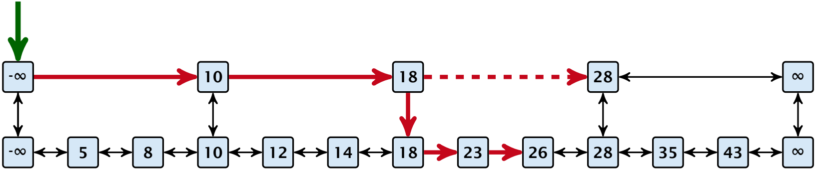
Add an express lane:



7.6 Skip Lists

How can we improve the search-operation?

Add an express lane:

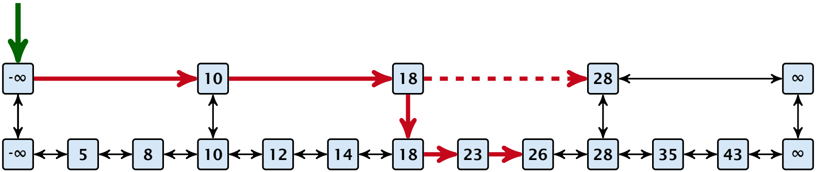


Let $|L_1|$ denote the number of elements in the "express lane", and $|L_0| = n$ the number of all elements (ignoring dummy elements).

7.6 Skip Lists

How can we improve the search-operation?

Add an express lane:



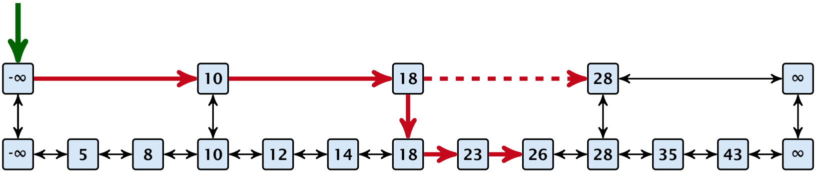
Let $|L_1|$ denote the number of elements in the “express lane”, and $|L_0| = n$ the number of all elements (ignoring dummy elements).

Worst case search time: $|L_1| + \frac{|L_0|}{|L_1|}$ (ignoring additive constants)

7.6 Skip Lists

How can we improve the search-operation?

Add an express lane:



Let $|L_1|$ denote the number of elements in the "express lane", and $|L_0| = n$ the number of all elements (ignoring dummy elements).

Worst case search time: $|L_1| + \frac{|L_0|}{|L_1|}$ (ignoring additive constants)

Choose $|L_1| = \sqrt{n}$. Then search time $\Theta(\sqrt{n})$.

7.6 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

7.6 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k + 1$ lists $L_0, \ldots, L_k$)

7.6 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \ldots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.

7.6 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \ldots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps.

7.6 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k + 1$ lists $L_0, \ldots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps.
- Find the largest item in list L_{k-2} that is smaller than x. At most $\lceil \frac{|L_{k-2}|}{|L_{k-1}|+1} \rceil + 2$ steps.

7.6 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \dots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps.
- Find the largest item in list L_{k-2} that is smaller than x. At most $\lceil \frac{|L_{k-2}|}{|L_{k-1}|+1} \rceil + 2$ steps.
- ...

7.6 Skip Lists

Add more express lanes. Lane L_i contains roughly every $\frac{L_{i-1}}{L_i}$-th item from list L_{i-1}.

Search(x) ($k+1$ lists $L_0, \ldots, L_k$)

- Find the largest item in list L_k that is smaller than x. At most $|L_k| + 2$ steps.
- Find the largest item in list L_{k-1} that is smaller than x. At most $\lceil \frac{|L_{k-1}|}{|L_k|+1} \rceil + 2$ steps.
- Find the largest item in list L_{k-2} that is smaller than x. At most $\lceil \frac{|L_{k-2}|}{|L_{k-1}|+1} \rceil + 2$ steps.
- ...
- At most $|L_k| + \sum_{i=1}^{k} \frac{L_{i-1}}{L_i} + 3(k+1)$ steps.

7.6 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

7.6 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$.

7.6 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$.
Choose $r = n^{\frac{1}{k+1}}$. Then

$$r^{-k}n + kr$$

7.6 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$.
Choose $r = n^{\frac{1}{k+1}}$. Then

$$r^{-k}n + kr = \left(n^{\frac{1}{k+1}}\right)^{-k} n + kn^{\frac{1}{k+1}}$$

7.6 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$.
Choose $r = n^{\frac{1}{k+1}}$. Then

$$\begin{aligned} r^{-k}n + kr &= \left(n^{\frac{1}{k+1}}\right)^{-k} n + kn^{\frac{1}{k+1}} \\ &= n^{1-\frac{k}{k+1}} + kn^{\frac{1}{k+1}} \end{aligned}$$

7.6 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k} n$.

Worst case running time is: $\mathcal{O}(r^{-k} n + kr)$.
Choose $r = n^{\frac{1}{k+1}}$. Then

$$\begin{aligned} r^{-k} n + kr &= \left(n^{\frac{1}{k+1}}\right)^{-k} n + kn^{\frac{1}{k+1}} \\ &= n^{1-\frac{k}{k+1}} + kn^{\frac{1}{k+1}} \\ &= (k+1) n^{\frac{1}{k+1}} \quad . \end{aligned}$$

7.6 Skip Lists

Choose ratios between list-lengths evenly, i.e., $\frac{|L_{i-1}|}{|L_i|} = r$, and, hence, $L_k \approx r^{-k}n$.

Worst case running time is: $\mathcal{O}(r^{-k}n + kr)$.
Choose $r = n^{\frac{1}{k+1}}$. Then

$$\begin{aligned} r^{-k}n + kr &= \left(n^{\frac{1}{k+1}}\right)^{-k} n + kn^{\frac{1}{k+1}} \\ &= n^{1-\frac{k}{k+1}} + kn^{\frac{1}{k+1}} \\ &= (k+1)n^{\frac{1}{k+1}} \quad . \end{aligned}$$

Choosing $k = \Theta(\log n)$ gives a logarithmic running time.

7.6 Skip Lists

How to do insert and delete?

- If we want that in L_i we always skip over roughly the same number of elements in L_{i-1} an insert or delete may require a lot of re-organisation.

Use randomization instead!

7.6 Skip Lists

How to do insert and delete?

- If we want that in L_i we always skip over roughly the same number of elements in L_{i-1} an insert or delete may require a lot of re-organisation.

Use randomization instead!

7.6 Skip Lists

How to do insert and delete?

- If we want that in L_i we always skip over roughly the same number of elements in L_{i-1} an insert or delete may require a lot of re-organisation.

Use randomization instead!

7.6 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.6 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.6 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.6 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.6 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.6 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \ldots\}$ of trials needed.
- Insert x into lists $L_0, \ldots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.6 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

7.6 Skip Lists

Insert:

- A search operation gives you the insert position for element x in every list.
- Flip a coin until it shows head, and record the number $t \in \{1, 2, \dots\}$ of trials needed.
- Insert x into lists $L_0, \dots, L_{t-1}$.

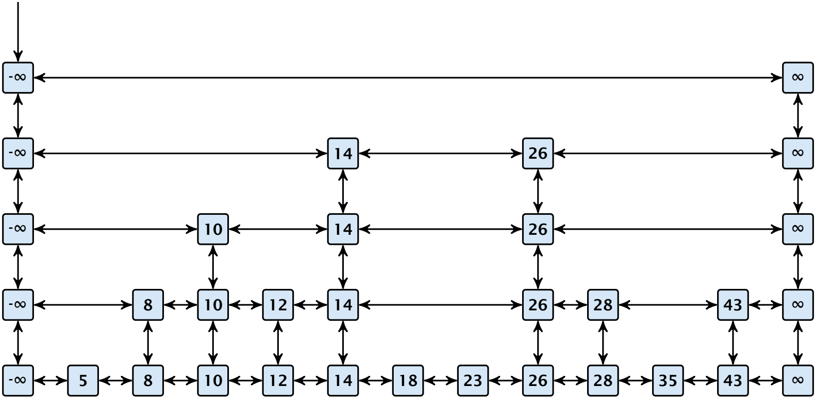
Delete:

- You get all predecessors via backward pointers.
- Delete x in all lists it actually appears in.

The time for both operations is dominated by the search time.

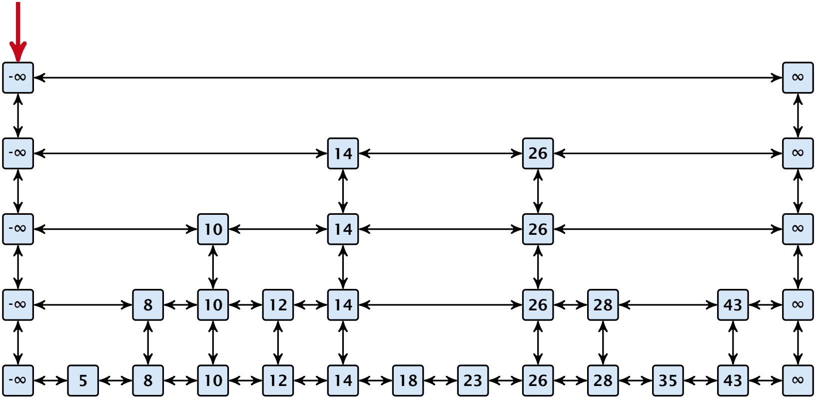
Skip Lists

Insert (35):



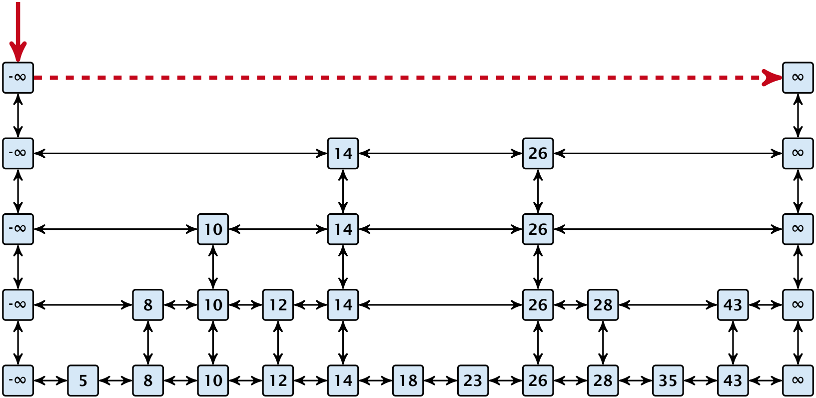
Skip Lists

Insert (35):



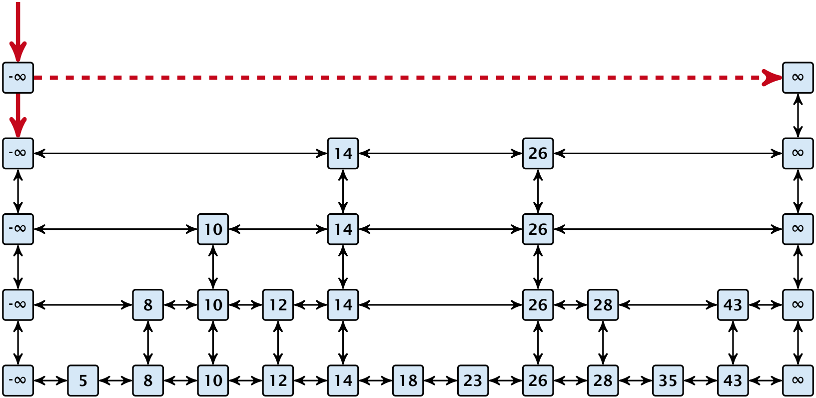
Skip Lists

Insert (35):



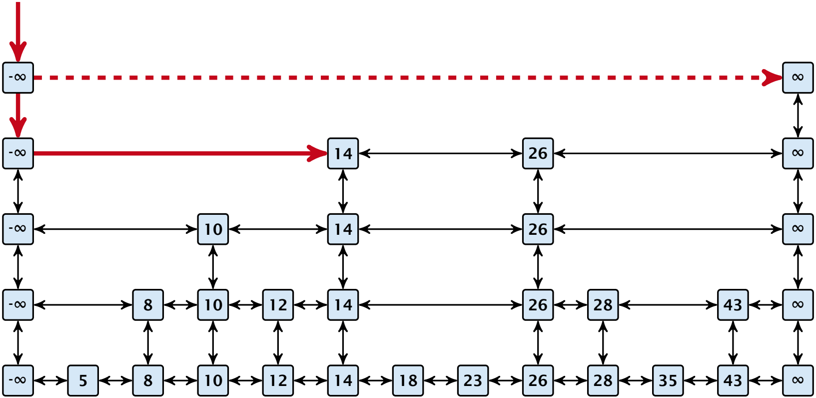
Skip Lists

Insert (35):



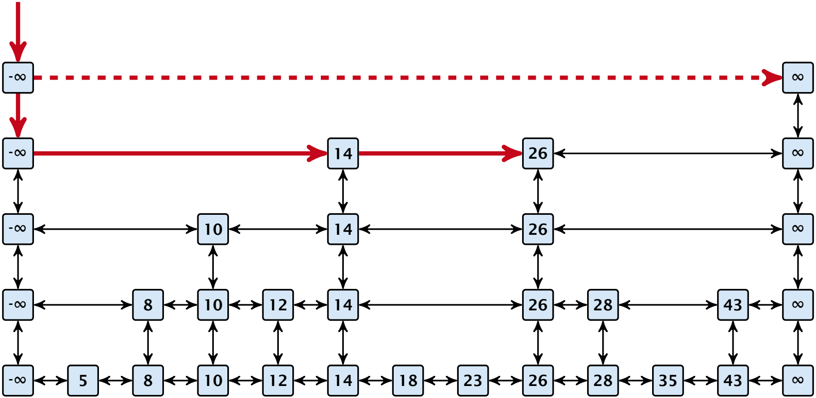
Skip Lists

Insert (35):



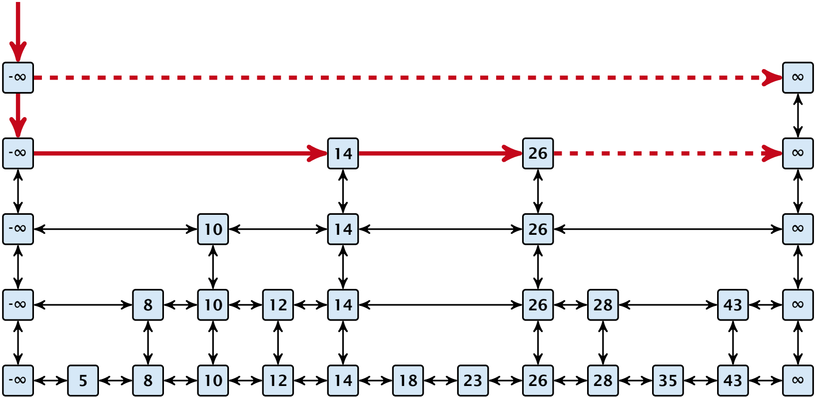
Skip Lists

Insert (35):



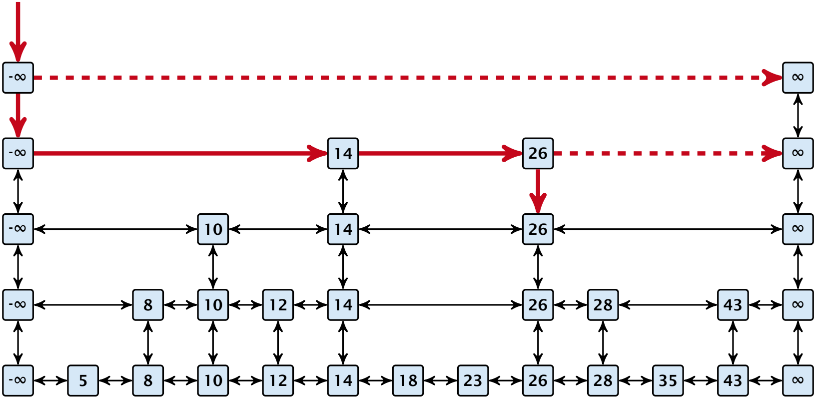
Skip Lists

Insert (35):



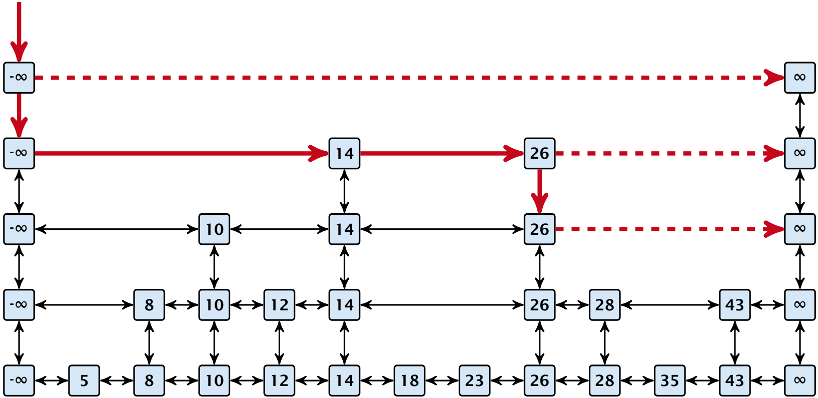
Skip Lists

Insert (35):



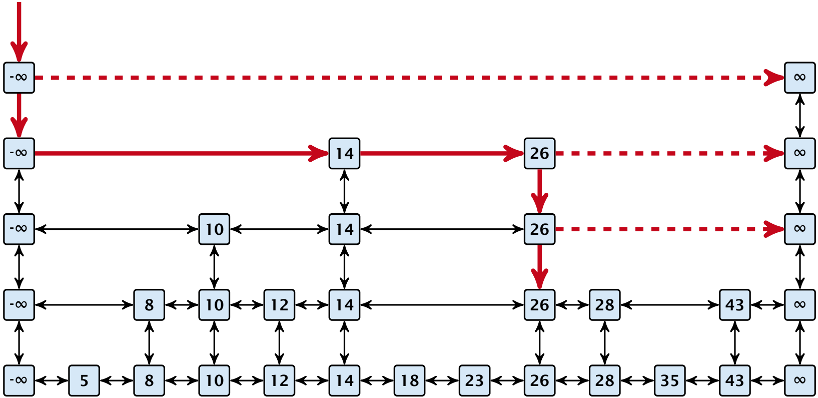
Skip Lists

Insert (35):



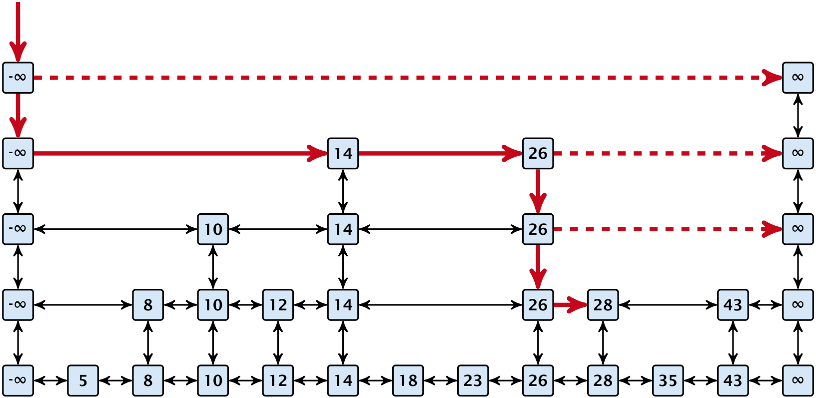
Skip Lists

Insert (35):



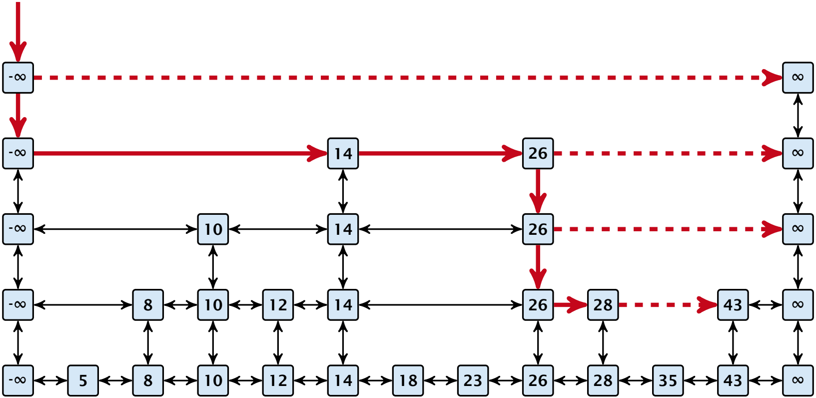
Skip Lists

Insert (35):



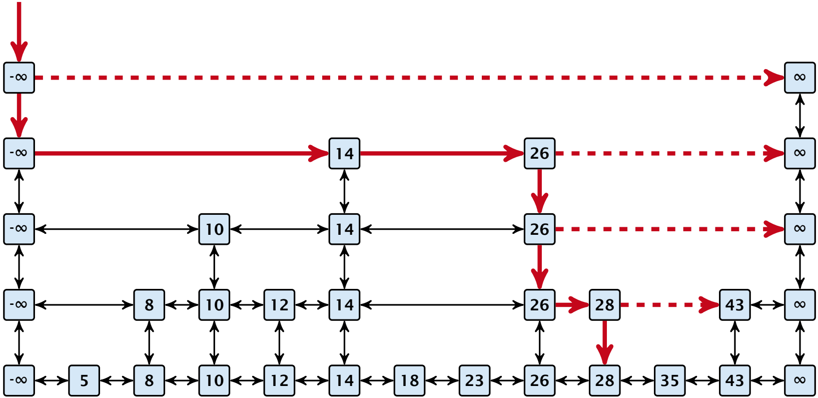
Skip Lists

Insert (35):



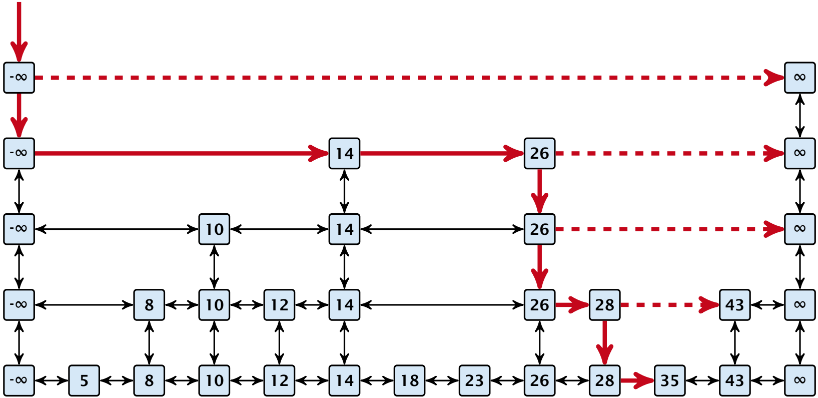
Skip Lists

Insert (35):



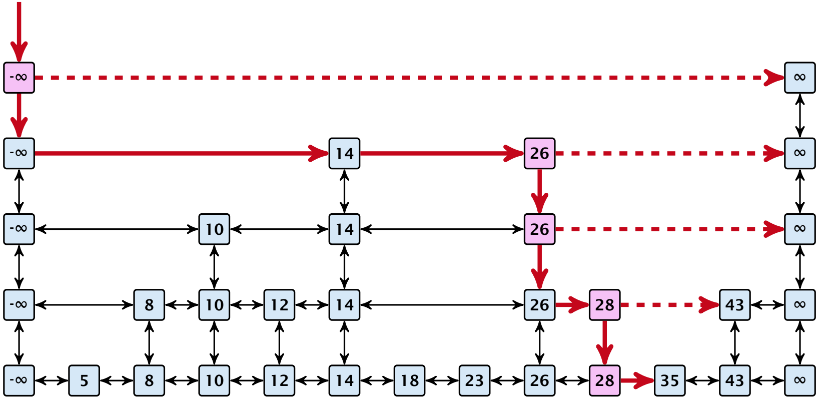
Skip Lists

Insert (35):



Skip Lists

Insert (35):



High Probability

Definition 11 (High Probability)

We say a **randomized** algorithm has running time $\mathcal{O}(\log n)$ with high probability if for any constant α the running time is at most $\mathcal{O}(\log n)$ with probability at least $1 - \frac{1}{n^\alpha}$.

Here the $\mathcal{O}$-notation hides a constant that may depend on α.

High Probability

Definition 11 (High Probability)

We say a **randomized** algorithm has running time $\mathcal{O}(\log n)$ with high probability if for any constant α the running time is at most $\mathcal{O}(\log n)$ with probability at least $1 - \frac{1}{n^\alpha}$.

Here the $\mathcal{O}$-notation hides a constant that may depend on α.

High Probability

Suppose there are a polynomially many events $E_1, E_2, \ldots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

High Probability

Suppose there are a polynomially many events $E_1, E_2, \dots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probability that all E_i hold is at least

$$\Pr[E_1 \wedge \cdots \wedge E_\ell]$$

High Probability

Suppose there are a polynomially many events $E_1, E_2, \ldots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probability that all E_i hold is at least

$$\Pr[E_1 \wedge \cdots \wedge E_\ell] = 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell]$$

High Probability

Suppose there are a polynomially many events $E_1, E_2, \ldots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probability that all E_i hold is at least

$$\begin{aligned}\Pr[E_1 \wedge \cdots \wedge E_\ell] &= 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell] \\ &\geq 1 - n^c \cdot n^{-\alpha}\end{aligned}$$

High Probability

Suppose there are a polynomially many events $E_1, E_2, \dots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probability that all E_i hold is at least

$$\begin{aligned}\Pr[E_1 \wedge \cdots \wedge E_\ell] &= 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell] \\ &\geq 1 - n^c \cdot n^{-\alpha} \\ &= 1 - n^{c-\alpha} \enspace .\end{aligned}$$

High Probability

Suppose there are a polynomially many events $E_1, E_2, \dots, E_\ell$, $\ell = n^c$ each holding with high probability (e.g. E_i may be the event that the i-th search in a skip list takes time at most $\mathcal{O}(\log n)$).

Then the probability that all E_i hold is at least

$$\begin{aligned}\Pr[E_1 \wedge \cdots \wedge E_\ell] &= 1 - \Pr[\bar{E}_1 \vee \cdots \vee \bar{E}_\ell] \\ &\geq 1 - n^c \cdot n^{-\alpha} \\ &= 1 - n^{c-\alpha} \ .\end{aligned}$$

This means $\Pr[E_1 \wedge \cdots \wedge E_\ell]$ holds with high probability.

7.6 Skip Lists

Lemma 12

A search (and, hence, also insert and delete) in a skip list with n elements takes time $\mathcal{O}(\log n)$ with high probability (w. h. p.).

Skip Lists

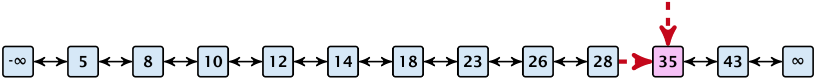
Backward analysis:

Skip Lists

Backward analysis:

Skip Lists

Backward analysis:

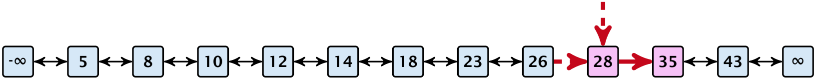


Skip Lists

Backward analysis:

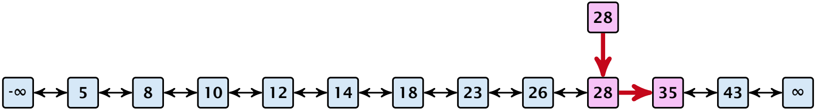
Skip Lists

Backward analysis:



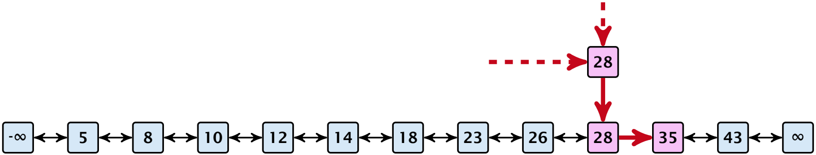
Skip Lists

Backward analysis:



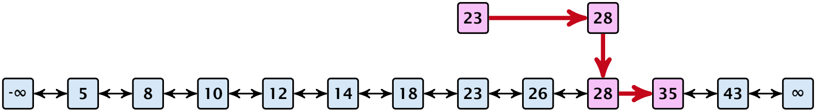
Skip Lists

Backward analysis:



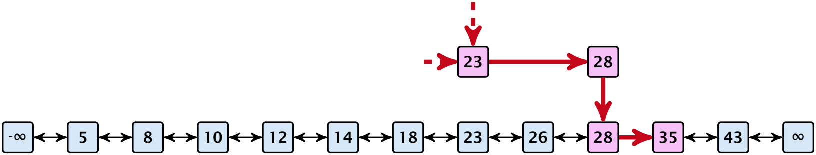
Skip Lists

Backward analysis:



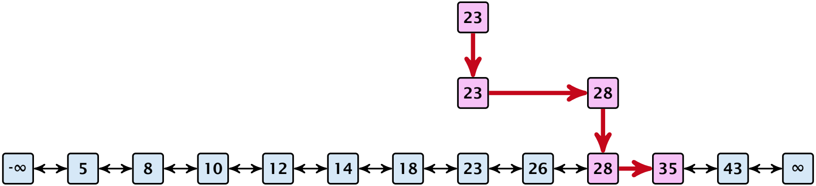
Skip Lists

Backward analysis:



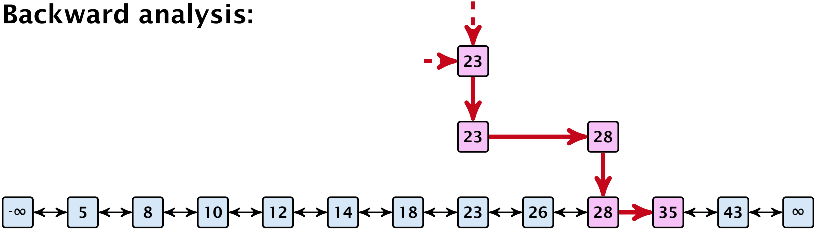
Skip Lists

Backward analysis:



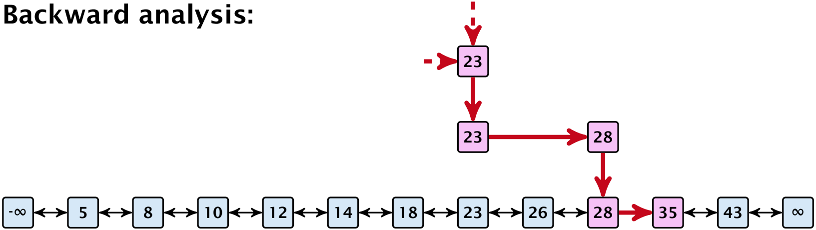
Skip Lists

Backward analysis:



Skip Lists

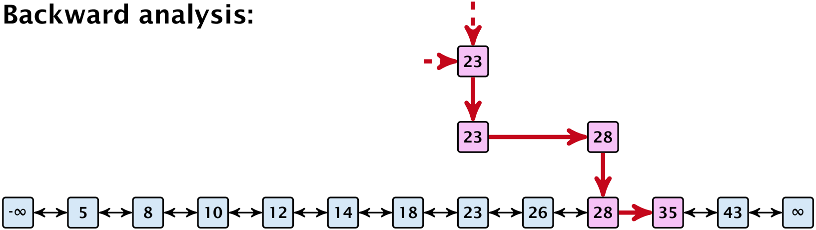
Backward analysis:



At each point the path goes up with probability $1/2$ and left with probability $1/2$.

Skip Lists

Backward analysis:



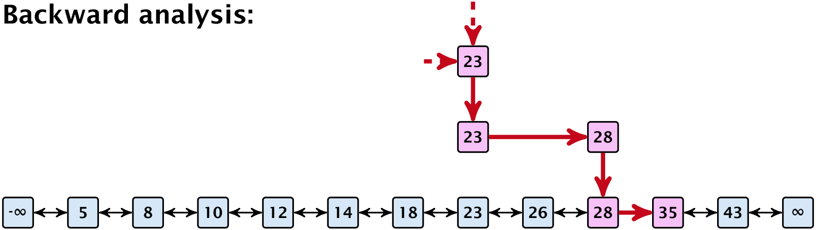
At each point the path goes up with probability $1/2$ and left with probability $1/2$.

We show that w.h.p:

- A "long" search path must also go very high.

Skip Lists

Backward analysis:



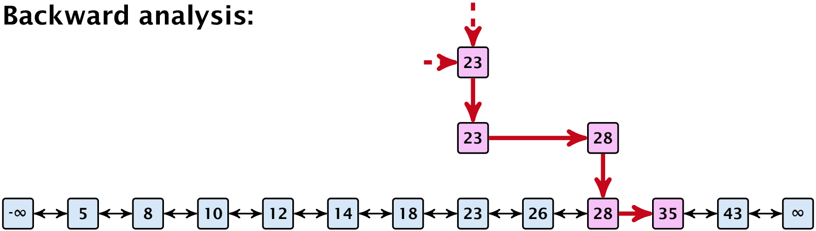
At each point the path goes up with probability $1/2$ and left with probability $1/2$.

We show that w.h.p:

- A "long" search path must also go very high.
- There are no elements in high lists.

Skip Lists

Backward analysis:



At each point the path goes up with probability $1/2$ and left with probability $1/2$.

We show that w.h.p:

- A "long" search path must also go very high.
- There are no elements in high lists.

From this it follows that w.h.p. there are no long paths.

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{en}{k}\right)^k$$

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k}$$

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot \ldots \cdot (n-k+1)}{k \cdot \ldots \cdot 1}$$

$$\left(\frac{n}{k}\right)^k \leq \binom{n}{k} \leq \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot \ldots \cdot (n-k+1)}{k \cdot \ldots \cdot 1} \geq \left(\frac{n}{k}\right)^k$$

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot \ldots \cdot (n-k+1)}{k \cdot \ldots \cdot 1} \ge \left(\frac{n}{k}\right)^k$$

$$\binom{n}{k}$$

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot \ldots \cdot (n-k+1)}{k \cdot \ldots \cdot 1} \ge \left(\frac{n}{k}\right)^k$$

$$\binom{n}{k} = \frac{n \cdot \ldots \cdot (n-k+1)}{k!}$$

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot \ldots \cdot (n-k+1)}{k \cdot \ldots \cdot 1} \ge \left(\frac{n}{k}\right)^k$$

$$\binom{n}{k} = \frac{n \cdot \ldots \cdot (n-k+1)}{k!} \le \frac{n^k}{k!}$$

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot \ldots \cdot (n-k+1)}{k \cdot \ldots \cdot 1} \ge \left(\frac{n}{k}\right)^k$$

$$\binom{n}{k} = \frac{n \cdot \ldots \cdot (n-k+1)}{k!} \le \frac{n^k}{k!} = \frac{n^k \cdot k^k}{k^k \cdot k!}$$

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot \ldots \cdot (n-k+1)}{k \cdot \ldots \cdot 1} \ge \left(\frac{n}{k}\right)^k$$

$$\binom{n}{k} = \frac{n \cdot \ldots \cdot (n-k+1)}{k!} \le \frac{n^k}{k!} = \frac{n^k \cdot k^k}{k^k \cdot k!}$$

$$= \left(\frac{n}{k}\right)^k \cdot \frac{k^k}{k!}$$

$$\left(\frac{n}{k}\right)^k \le \binom{n}{k} \le \left(\frac{en}{k}\right)^k$$

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!} = \frac{n \cdot \ldots \cdot (n-k+1)}{k \cdot \ldots \cdot 1} \ge \left(\frac{n}{k}\right)^k$$

$$\binom{n}{k} = \frac{n \cdot \ldots \cdot (n-k+1)}{k!} \le \frac{n^k}{k!} = \frac{n^k \cdot k^k}{k^k \cdot k!}$$

$$= \left(\frac{n}{k}\right)^k \cdot \frac{k^k}{k!} \le \left(\frac{en}{k}\right)^k$$

7.6 Skip Lists

7.6 Skip Lists

Let $E_{z,k}$ denote the event that a search path is of length z (number of edges) but does not visit a list above L_k.

7.6 Skip Lists

Let $E_{z,k}$ denote the event that a search path is of length z (number of edges) but does not visit a list above L_k.

In particular, this means that during the construction in the backward analysis we see at most k heads (i.e., coin flips that tell you to go up) in z trials.

7.6 Skip Lists

$$\Pr[E_{z,k}]$$

7.6 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

7.6 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)}$$

7.6 Skip Lists

$$\Pr[E_{z,k}] \le \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\le \binom{z}{k} 2^{-(z-k)} \le \left(\frac{ez}{k}\right)^k 2^{-(z-k)}$$

7.6 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \leq \left(\frac{2ez}{k}\right)^k 2^{-z}$$

7.6 Skip Lists

$$\Pr[E_{z,k}] \le \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\le \binom{z}{k} 2^{-(z-k)} \le \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \le \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \ge 1$ and $z = (\beta + \alpha)\gamma \log n$

7.6 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \leq \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \geq 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\leq \left(\frac{2ez}{k}\right)^k 2^{-\beta k} \cdot n^{-\gamma\alpha}$$

7.6 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \leq \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \geq 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\leq \left(\frac{2ez}{k}\right)^k 2^{-\beta k} \cdot n^{-\gamma\alpha} \leq \left(\frac{2ez}{2^\beta k}\right)^k \cdot n^{-\alpha}$$

7.6 Skip Lists

$$\Pr[E_{z,k}] \le \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\le \binom{z}{k} 2^{-(z-k)} \le \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \le \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \ge 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\le \left(\frac{2ez}{k}\right)^k 2^{-\beta k} \cdot n^{-\gamma\alpha} \le \left(\frac{2ez}{2^\beta k}\right)^k \cdot n^{-\alpha}$$

$$\le \left(\frac{2e(\beta+\alpha)}{2^\beta}\right)^k n^{-\alpha}$$

7.6 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \leq \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \geq 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\leq \left(\frac{2ez}{k}\right)^k 2^{-\beta k} \cdot n^{-\gamma\alpha} \leq \left(\frac{2ez}{2^\beta k}\right)^k \cdot n^{-\alpha}$$

$$\leq \left(\frac{2e(\beta + \alpha)}{2^\beta}\right)^k n^{-\alpha}$$

now choosing $\beta = 6\alpha$ gives

7.6 Skip Lists

$$\Pr[E_{z,k}] \le \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\le \binom{z}{k} 2^{-(z-k)} \le \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \le \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \ge 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\le \left(\frac{2ez}{k}\right)^k 2^{-\beta k} \cdot n^{-\gamma\alpha} \le \left(\frac{2ez}{2^\beta k}\right)^k \cdot n^{-\alpha}$$

$$\le \left(\frac{2e(\beta+\alpha)}{2^\beta}\right)^k n^{-\alpha}$$

now choosing $\beta = 6\alpha$ gives

$$\le \left(\frac{42\alpha}{64^\alpha}\right)^k n^{-\alpha}$$

7.6 Skip Lists

$$\Pr[E_{z,k}] \le \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\le \binom{z}{k} 2^{-(z-k)} \le \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \le \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \ge 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\le \left(\frac{2ez}{k}\right)^k 2^{-\beta k} \cdot n^{-\gamma\alpha} \le \left(\frac{2ez}{2^\beta k}\right)^k \cdot n^{-\alpha}$$

$$\le \left(\frac{2e(\beta+\alpha)}{2^\beta}\right)^k n^{-\alpha}$$

now choosing $\beta = 6\alpha$ gives

$$\le \left(\frac{42\alpha}{64^\alpha}\right)^k n^{-\alpha} \le n^{-\alpha}$$

7.6 Skip Lists

$$\Pr[E_{z,k}] \leq \Pr[\text{at most } k \text{ heads in } z \text{ trials}]$$

$$\leq \binom{z}{k} 2^{-(z-k)} \leq \left(\frac{ez}{k}\right)^k 2^{-(z-k)} \leq \left(\frac{2ez}{k}\right)^k 2^{-z}$$

choosing $k = \gamma \log n$ with $\gamma \geq 1$ and $z = (\beta + \alpha)\gamma \log n$

$$\leq \left(\frac{2ez}{k}\right)^k 2^{-\beta k} \cdot n^{-\gamma\alpha} \leq \left(\frac{2ez}{2^\beta k}\right)^k \cdot n^{-\alpha}$$

$$\leq \left(\frac{2e(\beta + \alpha)}{2^\beta}\right)^k n^{-\alpha}$$

now choosing $\beta = 6\alpha$ gives

$$\leq \left(\frac{42\alpha}{64^\alpha}\right)^k n^{-\alpha} \leq n^{-\alpha}$$

for $\alpha \geq 1$.

7.6 Skip Lists

7.6 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

7.6 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

7.6 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

7.6 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \enspace .$$

7.6 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \quad .$$

For the search to take at least $z = 7\alpha\gamma \log n$ steps either the event $E_{z,k}$ or the even A_{k+1} must hold.

7.6 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \ .$$

For the search to take at least $z = 7\alpha\gamma \log n$ steps either the event $E_{z,k}$ or the even A_{k+1} must hold.
Hence,

$$\Pr[\text{search requires } z \text{ steps}]$$

7.6 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \ .$$

For the search to take at least $z = 7\alpha\gamma \log n$ steps either the event $E_{z,k}$ or the even A_{k+1} must hold.
Hence,

$$\Pr[\text{search requires } z \text{ steps}] \leq \Pr[E_{z,k}] + \Pr[A_{k+1}]$$

7.6 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \ .$$

For the search to take at least $z = 7\alpha\gamma \log n$ steps either the event $E_{z,k}$ or the even A_{k+1} must hold.
Hence,

$$\begin{aligned} \Pr[\text{search requires } z \text{ steps}] &\leq \Pr[E_{z,k}] + \Pr[A_{k+1}] \\ &\leq n^{-\alpha} + n^{-(\gamma-1)} \end{aligned}$$

7.6 Skip Lists

So far we fixed $k = \gamma \log n$, $\gamma \geq 1$, and $z = 7\alpha\gamma \log n$, $\alpha \geq 1$.

This means that a search path of length $\Omega(\log n)$ visits a list on a level $\Omega(\log n)$, w.h.p.

Let A_{k+1} denote the event that the list L_{k+1} is non-empty. Then

$$\Pr[A_{k+1}] \leq n2^{-(k+1)} \leq n^{-(\gamma-1)} \; .$$

For the search to take at least $z = 7\alpha\gamma \log n$ steps either the event $E_{z,k}$ or the even A_{k+1} must hold.
Hence,

$$\begin{aligned} \Pr[\text{search requires } z \text{ steps}] &\leq \Pr[E_{z,k}] + \Pr[A_{k+1}] \\ &\leq n^{-\alpha} + n^{-(\gamma-1)} \end{aligned}$$

This means, the search requires at most z steps, w. h. p.

7 Dictionary

Dictionary:

- S**.insert**(x): Insert an element x.
- S**.delete**(x): Delete the element pointed to by x.
- S**.search**(k): Return a pointer to an element e with $\text{key}[e] = k$ in S if it exists; otherwise return null.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object x with key k is determined by successively comparing k to split-elements.

Hashing tries to directly compute the memory location from the given key. The goal is to have constant search time.

7 Dictionary

Dictionary:

- S.**insert**(x): Insert an element x.
- S.**delete**(x): Delete the element pointed to by x.
- S.**search**(k): Return a pointer to an element e with $\text{key}[e] = k$ in S if it exists; otherwise return null.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object x with key k is determined by successively comparing k to split-elements.

Hashing tries to directly compute the memory location from the given key. The goal is to have constant search time.

7 Dictionary

Dictionary:

- S.insert(x): Insert an element x.
- S.delete(x): Delete the element pointed to by x.
- S.search(k): Return a pointer to an element e with $\text{key}[e] = k$ in S if it exists; otherwise return null.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object x with key k is determined by successively comparing k to split-elements.

Hashing tries to directly compute the memory location from the given key. The goal is to have constant search time.

7 Dictionary

Dictionary:

- S.**insert**(x): Insert an element x.
- S.**delete**(x): Delete the element pointed to by x.
- S.**search**(k): Return a pointer to an element e with $\text{key}[e] = k$ in S if it exists; otherwise return null.

So far we have implemented the search for a key by carefully choosing split-elements.

Then the memory location of an object x with key k is determined by successively comparing k to split-elements.

Hashing tries to directly compute the memory location from the given key. The goal is to have constant search time.

7 Dictionary

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ hash-table.
- Hash function $h : U \to [0, \ldots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7 Dictionary

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ hash-table.
- Hash function $h : U \to [0, \ldots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7 Dictionary

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ hash-table.
- Hash function $h : U \to [0, \ldots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7 Dictionary

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq |U|$.
- Array $T[0, \dots, n-1]$ hash-table.
- Hash function $h : U \to [0, \dots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7 Dictionary

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq |U|$.
- Array $T[0, \dots, n-1]$ hash-table.
- Hash function $h : U \to [0, \dots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7 Dictionary

Definitions:

- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ hash-table.
- Hash function $h : U \to [0, \ldots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7 Dictionary

Definitions:

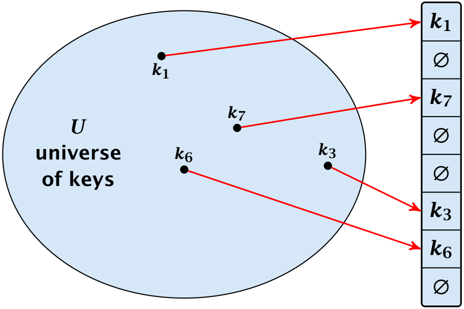
- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ hash-table.
- Hash function $h : U \to [0, \ldots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7 Dictionary

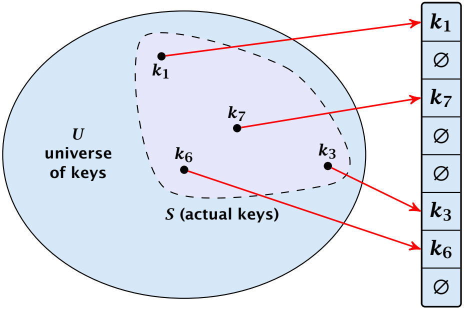
Ideally the hash function maps all keys to different memory locations.



This special case is known as Direct Addressing. It is usually very unrealistic as the universe of keys typically is quite large, and in particular larger than the available memory.

7 Dictionary

Suppose that we know the set S of actual keys (no insert/no delete). Then we may want to design a simple hash-function that maps all these keys to different memory locations.



Such a hash function h is called a perfect hash function for set S.

7 Dictionary

If we do not know the keys in advance, the best we can hope for is that the hash function distributes keys evenly across the table.

Problem: Collisions

Usually the universe U is much larger than the table-size n.

Hence, there may be two elements k_1, k_2 from the set S that map to the same memory location (i.e., $h(k_1) = h(k_2)$). This is called a collision.

7 Dictionary

If we do not know the keys in advance, the best we can hope for is that the hash function distributes keys evenly across the table.

Problem: Collisions

Usually the universe U is much larger than the table-size n.

Hence, there may be two elements k_1, k_2 from the set S that map to the same memory location (i.e., $h(k_1) = h(k_2)$). This is called a collision.

7 Dictionary

If we do not know the keys in advance, the best we can hope for is that the hash function distributes keys evenly across the table.

Problem: Collisions
Usually the universe U is much larger than the table-size n.

Hence, there may be two elements k_1, k_2 from the set S that map to the same memory location (i.e., $h(k_1) = h(k_2)$). This is called a collision.

7 Dictionary

Typically, collisions do not appear once the size of the set S of actual keys gets close to n, but already when $|S| \geq \omega(\sqrt{n})$.

Lemma 13

The probability of having a collision when hashing m elements into a table of size n under uniform hashing is at least

$$1 - e^{-\frac{m(m-1)}{2n}} \approx 1 - e^{-\frac{m^2}{2n}} .$$

Uniform hashing:

Choose a hash function uniformly at random from all functions $f : U \to [0, \ldots, n-1]$.

7 Dictionary

Typically, collisions do not appear once the size of the set S of actual keys gets close to n, but already when $|S| \geq \omega(\sqrt{n})$.

Lemma 13

The probability of having a collision when hashing m elements into a table of size n under uniform hashing is at least

$$1 - e^{-\frac{m(m-1)}{2n}} \approx 1 - e^{-\frac{m^2}{2n}} \ .$$

Uniform hashing:

Choose a hash function uniformly at random from all functions $f : U \rightarrow [0, \ldots, n-1]$.

Typically, collisions do not appear once the size of the set S of actual keys gets close to n, but already when $|S| \geq \omega(\sqrt{n})$.

Lemma 13

The probability of having a collision when hashing m elements into a table of size n under uniform hashing is at least

$$1 - e^{-\frac{m(m-1)}{2n}} \approx 1 - e^{-\frac{m^2}{2n}} \quad .$$

Uniform hashing:

Choose a hash function uniformly at random from all functions $f : U \to [0, \dots, n-1]$.

7 Dictionary

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

7 Dictionary

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}]$$

7 Dictionary

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n - \ell + 1}{n}$$

7 Dictionary

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n - \ell + 1}{n} = \prod_{j=0}^{m-1} \left(1 - \frac{j}{n}\right)$$

7 Dictionary

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \left(1 - \frac{j}{n}\right)$$

$$\leq \prod_{j=0}^{m-1} e^{-j/n}$$

7 Dictionary

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1}\Big(1-\frac{j}{n}\Big)$$

$$\leq \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1}\frac{j}{n}}$$

Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \Bigl(1 - \frac{j}{n}\Bigr)$$

$$\leq \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \quad .$$

7 Dictionary

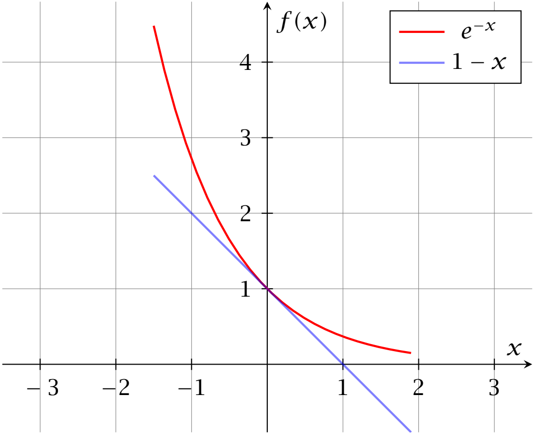
Proof.

Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1} \Big(1 - \frac{j}{n}\Big)$$

$$\le \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1} \frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \quad .$$

Here the first equality follows since the ℓ-th element that is hashed has a probability of $\frac{n-\ell+1}{n}$ to not generate a collision under the condition that the previous elements did not induce collisions. □



The inequality $1 - x \le e^{-x}$ is derived by stopping the Taylor-expansion of e^{-x} after the second term.

Resolving Collisions

The methods for dealing with collisions can be classified into the two main types

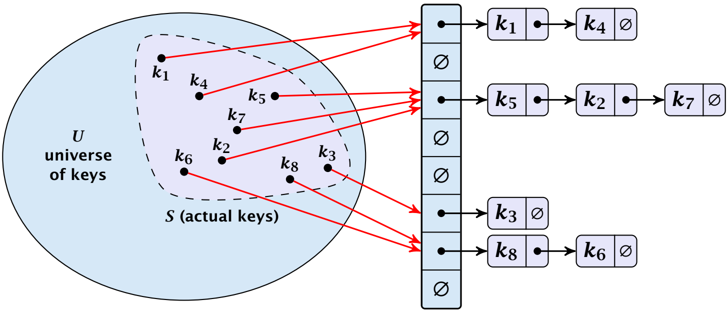
- **open addressing**, aka. closed hashing
- **hashing with chaining**, aka. closed addressing, open hashing.

There are applications e.g. computer chess where you do not resolve collisions at all.

Hashing with Chaining

Arrange elements that map to the same position in a linear list.

- Access: compute $h(x)$ and search list for $\mathrm{key}[x]$.
- Insert: insert at the front of the list.



7 Dictionary

Let A denote a strategy for resolving collisions. We use the following notation:

- A^+ denotes the average time for a **successful** search when using A;
- A^- denotes the average time for an **unsuccessful** search when using A;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

7 Dictionary

Let A denote a strategy for resolving collisions. We use the following notation:

- A^+ denotes the average time for a **successful** search when using A;
- A^- denotes the average time for an **unsuccessful** search when using A;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

7 Dictionary

Let A denote a strategy for resolving collisions. We use the following notation:

- A^+ denotes the average time for a **successful** search when using A;
- A^- denotes the average time for an **unsuccessful** search when using A;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

7 Dictionary

Let A denote a strategy for resolving collisions. We use the following notation:

- A^+ denotes the average time for a **successful** search when using A;
- A^- denotes the average time for an **unsuccessful** search when using A;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

7 Dictionary

Let A denote a strategy for resolving collisions. We use the following notation:

- A^+ denotes the average time for a **successful** search when using A;
- A^- denotes the average time for an **unsuccessful** search when using A;
- We parameterize the complexity results in terms of $\alpha := \frac{m}{n}$, the so-called fill factor of the hash-table.

We assume uniform hashing for the following analysis.

Hashing with Chaining

The time required for an unsuccessful search is 1 plus the length of the list that is examined.

Hashing with Chaining

The time required for an unsuccessful search is 1 plus the length of the list that is examined. The average length of a list is $\alpha = \frac{m}{n}$.

Hashing with Chaining

The time required for an unsuccessful search is 1 plus the length of the list that is examined. The average length of a list is $\alpha = \frac{m}{n}$. Hence, if A is the collision resolving strategy “Hashing with Chaining” we have

$$A^{-} = 1 + \alpha \ .$$

Hashing with Chaining

For a successful search observe that we do **not** choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the indicator variable for the event that k_i and k_j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)\right]$$

Hashing with Chaining

For a successful search observe that we do **not** choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the indicator variable for the event that k_i and k_j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)\right]$$

Hashing with Chaining

For a successful search observe that we do **not** choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the indicator variable for the event that k_i and k_j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)\right]$$

Hashing with Chaining

For a successful search observe that we do **not** choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the indicator variable for the event that k_i and k_j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)\right]$$

Hashing with Chaining

For a successful search observe that we do **not** choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the indicator variable for the event that k_i and k_j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)\right]$$

Hashing with Chaining

For a successful search observe that we do **not** choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the indicator variable for the event that k_i and k_j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\Bigg[\frac{1}{m}\sum_{i=1}^{m}\Bigg(1 + \underbrace{\sum_{j=i+1}^{m} X_{ij}}_{\text{keys before } k_i}\Bigg)\Bigg]$$

Hashing with Chaining

For a successful search observe that we do **not** choose a list at random, but we consider a random key k in the hash-table and ask for the search-time for k.

This is 1 plus the number of elements that lie before k in k's list.

Let k_ℓ denote the ℓ-th key inserted into the table.

Let for two keys k_i and k_j, X_{ij} denote the indicator variable for the event that k_i and k_j hash to the same position. Clearly, $\Pr[X_{ij} = 1] = 1/n$ for uniform hashing.

The expected successful search cost is

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\underbrace{\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)}_{\text{cost for key } k_i}\right]$$

Hashing with Chaining

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right]$$

Hashing with Chaining

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right]=\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)$$

Hashing with Chaining

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right]=\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)$$

$$=\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)$$

Hashing with Chaining

$$
\begin{aligned}
\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)\\
&= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)\\
&= 1+\frac{1}{mn}\sum_{i=1}^{m}(m-i)
\end{aligned}
$$

Hashing with Chaining

$$\begin{aligned}
\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)\\
&= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)\\
&= 1+\frac{1}{mn}\sum_{i=1}^{m}(m-i)\\
&= 1+\frac{1}{mn}\left(m^2-\frac{m(m+1)}{2}\right)
\end{aligned}$$

Hashing with Chaining

$$\begin{aligned}
\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right) \\
&= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right) \\
&= 1+\frac{1}{mn}\sum_{i=1}^{m}(m-i) \\
&= 1+\frac{1}{mn}\left(m^2-\frac{m(m+1)}{2}\right) \\
&= 1+\frac{m-1}{2n}
\end{aligned}$$

Hashing with Chaining

$$
\begin{aligned}
\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)\\
&= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)\\
&= 1+\frac{1}{mn}\sum_{i=1}^{m}(m-i)\\
&= 1+\frac{1}{mn}\left(m^2-\frac{m(m+1)}{2}\right)\\
&= 1+\frac{m-1}{2n} = 1+\frac{\alpha}{2}-\frac{\alpha}{2m}\ .
\end{aligned}
$$

Hashing with Chaining

$$
\begin{aligned}
\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\mathrm{E}\left[X_{ij}\right]\right)\\
&= \frac{1}{m}\sum_{i=1}^{m}\left(1+\sum_{j=i+1}^{m}\frac{1}{n}\right)\\
&= 1+\frac{1}{mn}\sum_{i=1}^{m}(m-i)\\
&= 1+\frac{1}{mn}\left(m^2-\frac{m(m+1)}{2}\right)\\
&= 1+\frac{m-1}{2n} = 1+\frac{\alpha}{2}-\frac{\alpha}{2m}\ .
\end{aligned}
$$

Hence, the expected cost for a successful search is $A^+ \leq 1+\frac{\alpha}{2}$.

Hashing with Chaining

Disadvantages:

- pointers increase memory requirements
- pointers may lead to bad cache efficiency

Advantages:

- no à priori limit on the number of elements
- deletion can be implemented efficiently
- by using balanced trees instead of linked list one can also obtain worst-case guarantees.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k, j)$ that determines the table-position to be examined in the j-th step. The values $h(k, 0)$,...,$h(k, n-1)$ must form a permutation of $0, \ldots, n-1$.

Search(k): Try position $h(k, 0)$; if it is empty your search fails; otw. continue with $h(k, 1)$, $h(k, 2)$, ...

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k, n-1)$, and this slot is non-empty then your table is full.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k, j)$ that determines the table-position to be examined in the j-th step. The values $h(k, 0)$,. . . ,$h(k, n-1)$ must form a permutation of $0, \ldots, n-1$.

Search(k): Try position $h(k, 0)$; if it is empty your search fails; otw. continue with $h(k, 1)$, $h(k, 2)$, ...

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k, n-1)$, and this slot is non-empty then your table is full.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k,j)$ that determines the table-position to be examined in the j-th step. The values $h(k,0)$,...,$h(k,n-1)$ must form a permutation of $0, \ldots, n-1$.

Search(k): Try position $h(k,0)$; if it is empty your search fails; otw. continue with $h(k,1)$, $h(k,2)$, ...

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k,n-1)$, and this slot is non-empty then your table is full.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k, j)$ that determines the table-position to be examined in the j-th step. The values $h(k, 0)$,...,$h(k, n-1)$ must form a permutation of $0, \ldots, n-1$.

Search(k): Try position $h(k, 0)$; if it is empty your search fails; otw. continue with $h(k, 1)$, $h(k, 2)$, ...

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k, n-1)$, and this slot is non-empty then your table is full.

Open Addressing

All objects are stored in the table itself.

Define a function $h(k,j)$ that determines the table-position to be examined in the j-th step. The values $h(k,0)$,. . . ,$h(k,n-1)$ must form a permutation of $0, \ldots, n-1$.

Search(k): Try position $h(k,0)$; if it is empty your search fails; otw. continue with $h(k,1)$, $h(k,2)$, ...

Insert(x): Search until you find an empty slot; insert your element there. If your search reaches $h(k,n-1)$, and this slot is non-empty then your table is full.

Open Addressing

Choices for $h(k, j)$:

- Linear probing:
  $h(k, i) = h(k) + i \mod n$
  (sometimes: $h(k, i) = h(k) + ci \mod n$).
- Quadratic probing:
  $h(k, i) = h(k) + c_1 i + c_2 i^2 \mod n$.
- Double hashing:
  $h(k, i) = h_1(k) + i h_2(k) \mod n$.

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing $h_2(k)$ must be relatively prime to n (teilerfremd); for quadratic probing c_1 and c_2 have to be chosen carefully).

Open Addressing

Choices for $h(k, j)$:

- Linear probing:
  $h(k,i) = h(k) + i \mod n$
  (sometimes: $h(k,i) = h(k) + ci \mod n$).
- Quadratic probing:
  $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$.
- Double hashing:
  $h(k,i) = h_1(k) + ih_2(k) \mod n$.

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing $h_2(k)$ must be relatively prime to n (teilerfremd); for quadratic probing c_1 and c_2 have to be chosen carefully).

Open Addressing

Choices for $h(k, j)$:

- Linear probing:
  $h(k, i) = h(k) + i \mod n$
  (sometimes: $h(k, i) = h(k) + ci \mod n$).
- Quadratic probing:
  $h(k, i) = h(k) + c_1 i + c_2 i^2 \mod n$.
- Double hashing:
  $h(k, i) = h_1(k) + i h_2(k) \mod n$.

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing $h_2(k)$ must be relatively prime to n (teilerfremd); for quadratic probing c_1 and c_2 have to be chosen carefully).

Open Addressing

Choices for $h(k,j)$:

- Linear probing:
  $h(k,i) = h(k) + i \mod n$
  (sometimes: $h(k,i) = h(k) + ci \mod n$).
- Quadratic probing:
  $h(k,i) = h(k) + c_1 i + c_2 i^2 \mod n$.
- Double hashing:
  $h(k,i) = h_1(k) + i h_2(k) \mod n$.

For quadratic probing and double hashing one has to ensure that the search covers all positions in the table (i.e., for double hashing $h_2(k)$ must be relatively prime to n (teilerfremd); for quadratic probing c_1 and c_2 have to be chosen carefully).

Linear Probing

- Advantage: Cache-efficiency. The new probe position is very likely to be in the cache.
- Disadvantage: Primary clustering. Long sequences of occupied table-positions get longer as they have a larger probability to be hit. Furthermore, they can merge forming larger sequences.

Lemma 14

Let L be the method of linear probing for resolving collisions:

$$L^{+} \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

$$L^{-} \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Linear Probing

- Advantage: Cache-efficiency. The new probe position is very likely to be in the cache.
- Disadvantage: Primary clustering. Long sequences of occupied table-positions get longer as they have a larger probability to be hit. Furthermore, they can merge forming larger sequences.

Lemma 14

Let L be the method of linear probing for resolving collisions:

$$L^+ \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

$$L^- \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Linear Probing

- Advantage: Cache-efficiency. The new probe position is very likely to be in the cache.
- Disadvantage: Primary clustering. Long sequences of occupied table-positions get longer as they have a larger probability to be hit. Furthermore, they can merge forming larger sequences.

Lemma 14

Let L be the method of linear probing for resolving collisions:

$$L^+ \approx \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

$$L^- \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

Quadratic Probing

- Not as cache-efficient as Linear Probing.
- Secondary clustering: caused by the fact that all keys mapped to the same position have the same probe sequence.

Lemma 15

Let Q be the method of quadratic probing for resolving collisions:

$$Q^+ \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$$

$$Q^- \approx \frac{1}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right) - \alpha$$

Quadratic Probing

- Not as cache-efficient as Linear Probing.
- Secondary clustering: caused by the fact that all keys mapped to the same position have the same probe sequence.

Lemma 15
Let Q be the method of quadratic probing for resolving collisions:

$$Q^{+} \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$$

$$Q^{-} \approx \frac{1}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right) - \alpha$$

Double Hashing

- Any probe into the hash-table usually creates a cache-miss.

Lemma 16

Let A be the method of double hashing for resolving collisions:

$$D^+ \approx \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

$$D^- \approx \frac{1}{1-\alpha}$$

Double Hashing

- Any probe into the hash-table usually creates a cache-miss.

Lemma 16

Let A be the method of double hashing for resolving collisions:

$$D^+ \approx \frac{1}{\alpha} \ln\left(\frac{1}{1-\alpha}\right)$$

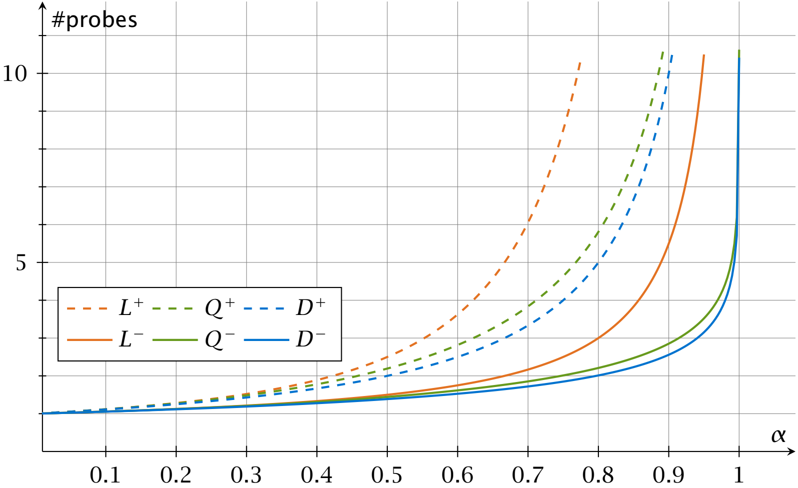
$$D^- \approx \frac{1}{1-\alpha}$$

Open Addressing

Some values:

α	*Linear Probing*		*Quadratic Probing*		*Double Hashing*	
	L^+	L^-	Q^+	Q^-	D^+	D^-
0.5	1.5	2.5	1.44	2.19	1.39	2
0.9	5.5	50.5	2.85	11.40	2.55	10
0.95	10.5	200.5	3.52	22.05	3.15	20

Open Addressing



Analysis of Idealized Open Address Hashing

We analyze the time for a search in a very idealized Open Addressing scheme.

- The probe sequence $h(k,0), h(k,1), h(k,2), \ldots$ is equally likely to be any permutation of $\langle 0, 1, \ldots, n-1 \rangle$.

Analysis of Idealized Open Address Hashing

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i-1}]$$

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \dots \cap A_{i-1}]& \\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \dots \cdot \Pr[A_{i-1} \mid A_1 \cap \dots \cap A_{i-2}]\end{aligned}$$

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i-1}]$$
$$= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot$$
$$\ldots \cdot \Pr[A_{i-1} \mid A_1 \cap \cdots \cap A_{i-2}]$$

$$\Pr[X \geq i]$$

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i-1}] \\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \ldots \cdot \Pr[A_{i-1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

$$\Pr[X \geq i] = \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \ldots \cdot \frac{m-i+2}{n-i+2}$$

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i-1}]&\\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \ldots \cdot \Pr[A_{i-1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

$$\begin{aligned}\Pr[X \geq i] &= \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \ldots \cdot \frac{m-i+2}{n-i+2} \\ &\leq \left(\frac{m}{n}\right)^{i-1}\end{aligned}$$

Analysis of Idealized Open Address Hashing

Let X denote a random variable describing the number of probes in an unsuccessful search.

Let A_i denote the event that the i-th probe occurs and is to a non-empty slot.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i-1}]& \\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \ldots \cdot \Pr[A_{i-1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

$$\begin{aligned}\Pr[X \geq i] &= \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \ldots \cdot \frac{m-i+2}{n-i+2} \\ &\leq \left(\frac{m}{n}\right)^{i-1} = \alpha^{i-1} \ .\end{aligned}$$

Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X]$$

Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i]$$

Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1}$$

Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i}$$

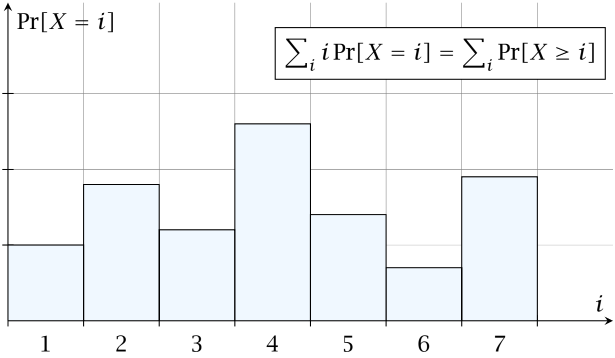
Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha} \; .$$

Analysis of Idealized Open Address Hashing

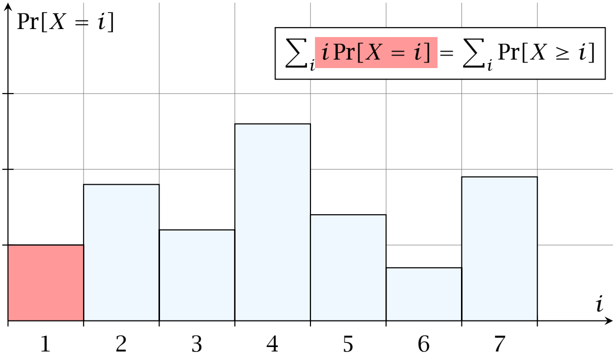
$$E[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha} .$$

$$\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \dots$$

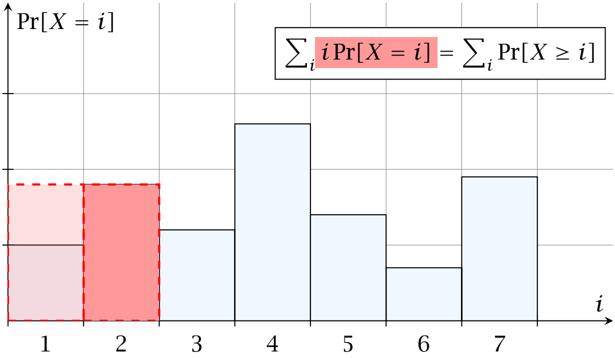


Pr[X = i]
∑_i i Pr[X = i] = ∑_i Pr[X ≥ i]
i
1
2
3
4
5
6
7

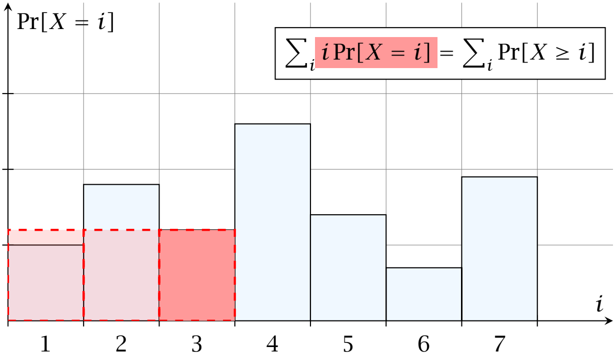
$i = 1$



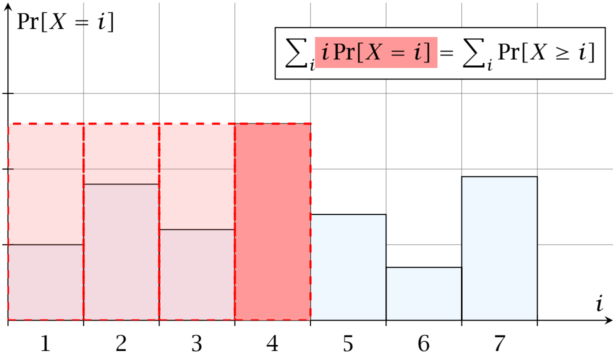
$i = 2$



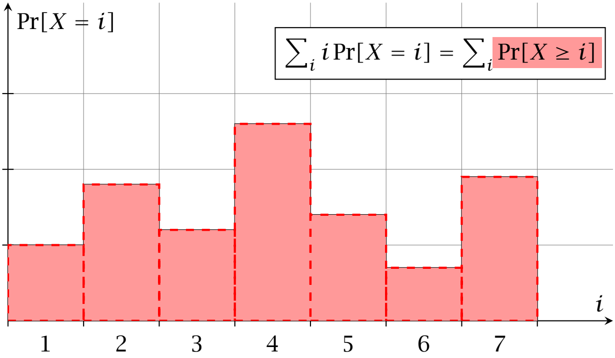
$i = 3$



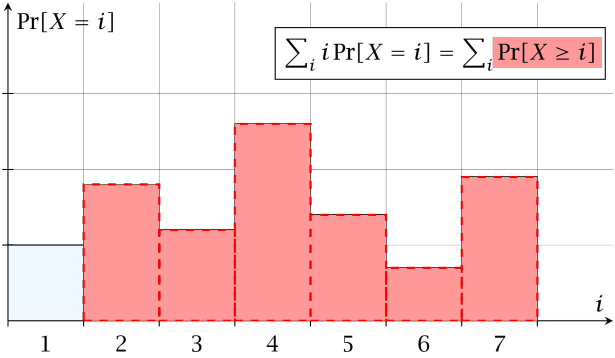
$i = 4$



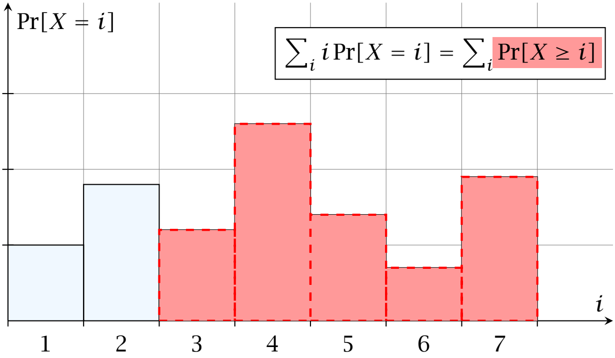
$i = 1$



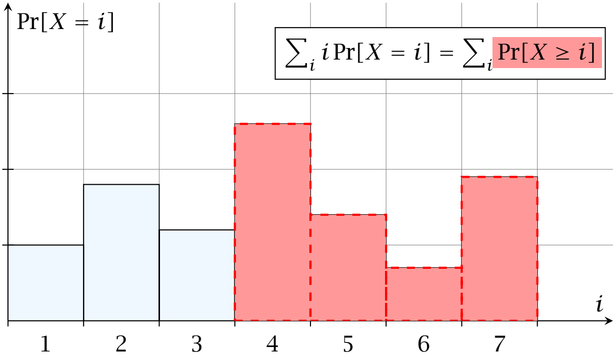
$i = 2$

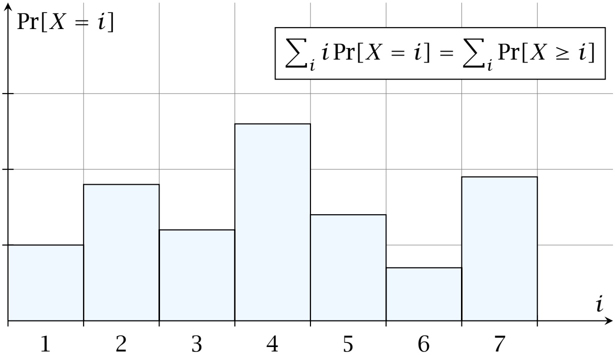


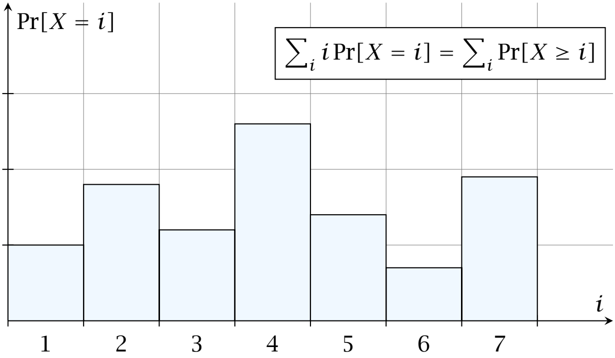
$i = 3$



$i = 4$




$\Pr[X = i]$
$\sum_i i \Pr[X = i] = \sum_i \Pr[X \geq i]$
i
1
2
3
4
5
6
7



The j-th rectangle appears in both sums j times. (j times in the first due to multiplication with j; and j times in the second for summands $i = 1, 2, \ldots, j$)

Analysis of Idealized Open Address Hashing

Analysis of Idealized Open Address Hashing

The number of probes in a successful search for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Analysis of Idealized Open Address Hashing

The number of probes in a successful search for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i + 1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

Analysis of Idealized Open Address Hashing

The number of probes in a successful search for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i}$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful search for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i}$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful search for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful search for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

$$\leq \frac{1}{\alpha}\int_{n-m}^{n}\frac{1}{x}\mathrm{d}x$$

Analysis of Idealized Open Address Hashing

The number of probes in a successful search for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

$$\leq \frac{1}{\alpha}\int_{n-m}^{n}\frac{1}{x}\mathrm{d}x = \frac{1}{\alpha}\ln\frac{n}{n-m}$$

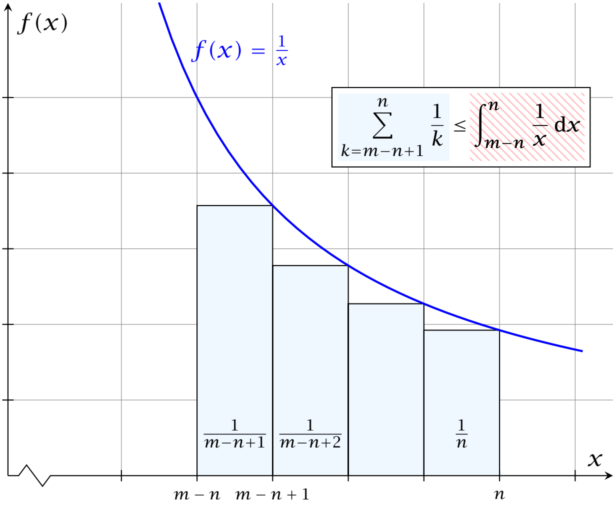
Analysis of Idealized Open Address Hashing

The number of probes in a successful search for k is equal to the number of probes made in an unsuccessful search for k at the time that k is inserted.

Let k be the $i+1$-st element. The expected time for a search for k is at most $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

$$\leq \frac{1}{\alpha}\int_{n-m}^{n}\frac{1}{x}\mathrm{d}x = \frac{1}{\alpha}\ln\frac{n}{n-m} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha} \;.$$



$f(x)$
$f(x) = \frac{1}{x}$
$\sum_{k=m-n+1}^{n} \frac{1}{k} \le \int_{m-n}^{n} \frac{1}{x}\, dx$
$\frac{1}{m-n+1}$
$\frac{1}{m-n+2}$
$\frac{1}{n}$
x
$m-n$
$m-n+1$
n

Deletions

How do we delete in a hash-table?

- For hashing with chaining this is not a problem. Simply search for the key, and delete the item in the corresponding list.
- For open addressing this is difficult.

Deletions

How do we delete in a hash-table?

- For hashing with chaining this is not a problem. Simply search for the key, and delete the item in the corresponding list.
- For open addressing this is difficult.

Deletions

- Simply removing a key might interrupt the probe sequence of other keys which then cannot be found anymore.
- One can delete an element by replacing it with a deleted-marker.
  - During an insertion if a deleted-marker is encountered an element can be inserted there.
  - During a search a deleted-marker must not be used to terminate the probe sequence.
- The table could fill up with deleted-markers leading to bad performance.
- If a table contains many deleted-markers (linear fraction of the keys) one can rehash the whole table and amortize the cost for this rehash against the cost for the deletions.

Deletions

- Simply removing a key might interrupt the probe sequence of other keys which then cannot be found anymore.
- One can delete an element by replacing it with a deleted-marker.
  - During an insertion if a deleted-marker is encountered an element can be inserted there.
  - During a search a deleted-marker must not be used to terminate the probe sequence.
- The table could fill up with deleted-markers leading to bad performance.
- If a table contains many deleted-markers (linear fraction of the keys) one can rehash the whole table and amortize the cost for this rehash against the cost for the deletions.

Deletions

- Simply removing a key might interrupt the probe sequence of other keys which then cannot be found anymore.
- One can delete an element by replacing it with a deleted-marker.
  - During an insertion if a deleted-marker is encountered an element can be inserted there.
  - During a search a deleted-marker must not be used to terminate the probe sequence.
- The table could fill up with deleted-markers leading to bad performance.
- If a table contains many deleted-markers (linear fraction of the keys) one can rehash the whole table and amortize the cost for this rehash against the cost for the deletions.

Deletions

- Simply removing a key might interrupt the probe sequence of other keys which then cannot be found anymore.
- One can delete an element by replacing it with a deleted-marker.
  - During an insertion if a deleted-marker is encountered an element can be inserted there.
  - During a search a deleted-marker must not be used to terminate the probe sequence.
- The table could fill up with deleted-markers leading to bad performance.
- If a table contains many deleted-markers (linear fraction of the keys) one can rehash the whole table and amortize the cost for this rehash against the cost for the deletions.

Deletions

- Simply removing a key might interrupt the probe sequence of other keys which then cannot be found anymore.
- One can delete an element by replacing it with a deleted-marker.
  - During an insertion if a deleted-marker is encountered an element can be inserted there.
  - During a search a deleted-marker must not be used to terminate the probe sequence.
- The table could fill up with deleted-markers leading to bad performance.
- If a table contains many deleted-markers (linear fraction of the keys) one can rehash the whole table and amortize the cost for this rehash against the cost for the deletions.

Deletions

- Simply removing a key might interrupt the probe sequence of other keys which then cannot be found anymore.
- One can delete an element by replacing it with a deleted-marker.
  - During an insertion if a deleted-marker is encountered an element can be inserted there.
  - During a search a deleted-marker must not be used to terminate the probe sequence.
- The table could fill up with deleted-markers leading to bad performance.
- If a table contains many deleted-markers (linear fraction of the keys) one can rehash the whole table and amortize the cost for this rehash against the cost for the deletions.

Deletions for Linear Probing

- For Linear Probing one can delete elements without using deletion-markers.
- Upon a deletion elements that are further down in the probe-sequence may be moved to guarantee that they are still found during a search.

Deletions for Linear Probing

- For Linear Probing one can delete elements without using deletion-markers.
- Upon a deletion elements that are further down in the probe-sequence may be moved to guarantee that they are still found during a search.

Deletions for Linear Probing

Algorithm 16 delete(p)

$T[p] \leftarrow$ null
$p \leftarrow \text{succ}(p)$
while $T[p] \neq$ null **do**
    $y \leftarrow T[p]$
    $T[p] \leftarrow$ null
    $p \leftarrow \text{succ}(p)$
    insert(y)

p is the index into the table-cell that contains the object to be deleted.

Pointers into the hash-table become invalid.

Deletions for Linear Probing

Algorithm 16 delete(p)

```
T[p] ← null
p ← succ(p)
while T[p] ≠ null do
    y ← T[p]
    T[p] ← null
    p ← succ(p)
    insert(y)
```

p is the index into the table-cell that contains the object to be deleted.

Pointers into the hash-table become invalid.

Universal Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \to [0, \dots, n-1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set $\mathcal{H}$ of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from $\mathcal{H}$.

Universal Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \to [0, \dots, n-1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set $\mathcal{H}$ of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from $\mathcal{H}$.

Universal Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \to [0, \dots, n-1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set $\mathcal{H}$ of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from $\mathcal{H}$.

Universal Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \to [0, \dots, n-1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set $\mathcal{H}$ of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from $\mathcal{H}$.

Universal Hashing

Regardless, of the choice of hash-function there is always an input (a set of keys) that has a very poor worst-case behaviour.

Therefore, so far we assumed that the hash-function is random so that regardless of the input the average case behaviour is good.

However, the assumption of uniform hashing that h is chosen randomly from all functions $f : U \to [0, \dots, n-1]$ is clearly unrealistic as there are $n^{|U|}$ such functions. Even writing down such a function would take $|U| \log n$ bits.

Universal hashing tries to define a set $\mathcal{H}$ of functions that is much smaller but still leads to good average case behaviour when selecting a hash-function uniformly at random from $\mathcal{H}$.

Universal Hashing

Definition 17
A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0,\ldots,n-1\}$ is called universal if for all $u_1, u_2 \in U$ with $u_1 \neq u_2$

$$\Pr[h(u_1) = h(u_2)] \leq \frac{1}{n} \ ,$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

Note that this means that the probability of a collision is at most $\frac{1}{n}$.

Universal Hashing

Definition 17

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0,\ldots,n-1\}$ is called universal if for all $u_1, u_2 \in U$ with $u_1 \neq u_2$

$$\Pr[h(u_1) = h(u_2)] \leq \frac{1}{n} \ ,$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

Note that this means that the probability of a collision is at most $\frac{1}{n}$.

Universal Hashing

Definition 18
A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \ldots, n-1\}$ is called 2-independent (pairwise independent) if the following two conditions hold

- For any key $u \in U$, and $t \in \{0, \ldots, n-1\}$ $\Pr[h(u) = t] = \frac{1}{n}$, i.e., a key is distributed uniformly within the hash-table.
- For all $u_1, u_2 \in U$ with $u_1 \neq u_2$, and for any two hash-positions t_1, t_2:

$$\Pr[h(u_1) = t_1 \wedge h(u_2) = t_2] \leq \frac{1}{n^2} \ .$$

This requirement clearly implies a universal hash-function.

Universal Hashing

Definition 18

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \ldots, n-1\}$ is called 2-independent (pairwise independent) if the following two conditions hold

- For any key $u \in U$, and $t \in \{0, \ldots, n-1\}$ $\Pr[h(u) = t] = \frac{1}{n}$, i.e., a key is distributed uniformly within the hash-table.
- For all $u_1, u_2 \in U$ with $u_1 \neq u_2$, and for any two hash-positions t_1, t_2:

$$\Pr[h(u_1) = t_1 \wedge h(u_2) = t_2] \leq \frac{1}{n^2} .$$

This requirement clearly implies a universal hash-function.

Universal Hashing

Definition 19

A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \ldots, n-1\}$ is called *k-independent* if for any choice of $\ell \leq k$ distinct keys $u_1, \ldots, u_\ell \in U$, and for any set of ℓ not necessarily distinct hash-positions $t_1, \ldots, t_\ell$:

$$\Pr[h(u_1) = t_1 \wedge \cdots \wedge h(u_\ell) = t_\ell] \leq \frac{1}{n^\ell} \enspace ,$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

Universal Hashing

Definition 20
A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0, \dots, n-1\}$ is called (μ, k)-independent if for any choice of $\ell \leq k$ distinct keys $u_1, \dots, u_\ell \in U$, and for any set of ℓ not necessarily distinct hash-positions $t_1, \dots, t_\ell$:

$$\Pr[h(u_1) = t_1 \wedge \cdots \wedge h(u_\ell) = t_\ell] \leq \frac{\mu}{n^\ell} \ ,$$

where the probability is w. r. t. the choice of a random hash-function from set $\mathcal{H}$.

Universal Hashing

Let $U := \{0, \ldots, p-1\}$ for a prime p. Let $\mathbb{Z}_p := \{0, \ldots, p-1\}$, and let $\mathbb{Z}_p^* := \{1, \ldots, p-1\}$ denote the set of invertible elements in $\mathbb{Z}_p$.

Define

$$h_{a,b}(x) := (ax + b \bmod p) \bmod n$$

Lemma 21

The class

$$\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is a universal class of hash-functions from U to $\{0, \ldots, n-1\}$.

Universal Hashing

Let $U := \{0, \ldots, p-1\}$ for a prime p. Let $\mathbb{Z}_p := \{0, \ldots, p-1\}$, and let $\mathbb{Z}_p^* := \{1, \ldots, p-1\}$ denote the set of invertible elements in $\mathbb{Z}_p$.

Define

$$h_{a,b}(x) := (ax + b \bmod p) \bmod n$$

Lemma 21

The class

$$\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is a universal class of hash-functions from U to $\{0, \ldots, n-1\}$.

Universal Hashing

Let $U := \{0, \ldots, p-1\}$ for a prime p. Let $\mathbb{Z}_p := \{0, \ldots, p-1\}$, and let $\mathbb{Z}_p^* := \{1, \ldots, p-1\}$ denote the set of invertible elements in $\mathbb{Z}_p$.

Define

$$h_{a,b}(x) := (ax + b \bmod p) \bmod n$$

Lemma 21

The class

$$\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is a universal class of hash-functions from U to $\{0, \ldots, n-1\}$.

Universal Hashing

Let $U := \{0, \ldots, p-1\}$ for a prime p. Let $\mathbb{Z}_p := \{0, \ldots, p-1\}$, and let $\mathbb{Z}_p^* := \{1, \ldots, p-1\}$ denote the set of invertible elements in $\mathbb{Z}_p$.

Define

$$h_{a,b}(x) := (ax + b \bmod p) \bmod n$$

Lemma 21

The class

$$\mathcal{H} = \{h_{a,b} \mid a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p\}$$

is a universal class of hash-functions from U to $\{0, \ldots, n-1\}$.

Universal Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $ax + b \not\equiv ay + b \pmod{p}$

Universal Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $ax + b \not\equiv ay + b \pmod{p}$

  If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

  Multiplying with $a \not\equiv 0 \pmod{p}$ gives

  $$a(x - y) \not\equiv 0 \pmod{p}$$

  where we use that $\mathbb{Z}_p$ is a field (Körper) and, hence, has no zero divisors (nullteilerfrei).

Universal Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $ax + b \not\equiv ay + b \pmod{p}$

If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

Multiplying with $a \not\equiv 0 \pmod{p}$ gives

$$a(x - y) \not\equiv 0 \pmod{p}$$

where we use that $\mathbb{Z}_p$ is a field (Körper) and, hence, has no zero divisors (nullteilerfrei).

Universal Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $\boldsymbol{ax + b \not\equiv ay + b \pmod{p}}$

  If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

  Multiplying with $a \not\equiv 0 \pmod{p}$ gives

  $$a(x - y) \not\equiv 0 \pmod{p}$$

  where we use that $\mathbb{Z}_p$ is a field (Körper) and, hence, has no zero divisors (nullteilerfrei).

Universal Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $\boldsymbol{ax + b \not\equiv ay + b \pmod{p}}$

  If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

  Multiplying with $a \not\equiv 0 \pmod{p}$ gives

  $$a(x - y) \not\equiv 0 \pmod{p}$$

  where we use that $\mathbb{Z}_p$ is a field (Körper) and, hence, has no zero divisors (nullteilerfrei).

Universal Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $\boldsymbol{ax + b \not\equiv ay + b \pmod{p}}$

  If $x \neq y$ then $(x - y) \not\equiv 0 \pmod{p}$.

  Multiplying with $a \not\equiv 0 \pmod{p}$ gives

  $$a(x - y) \not\equiv 0 \pmod{p}$$

  where we use that $\mathbb{Z}_p$ is a field (Körper) and, hence, has no zero divisors (nullteilerfrei).

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to a different pair (t_x, t_y) with $t_x := ax + b$ and $t_y := ay + b$.

- The hash-function does not generate collisions before the $(\bmod\ n)$-operation. Furthermore, every choice (a, b) is mapped to a different pair (t_x, t_y) with $t_x := ax + b$ and $t_y := ay + b$.

  This holds because we can compute a and b when given t_x and t_y:

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to a different pair (t_x, t_y) with $t_x := ax + b$ and $t_y := ay + b$.

  This holds because we can compute a and b when given t_x and t_y:

$$t_x \equiv ax + b \pmod{p}$$

$$t_y \equiv ay + b \pmod{p}$$

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to a different pair (t_x, t_y) with $t_x := ax + b$ and $t_y := ay + b$.

  This holds because we can compute a and b when given t_x and t_y:

$$t_x \equiv ax + b \pmod{p}$$
$$t_y \equiv ay + b \pmod{p}$$

$$t_x - t_y \equiv a(x - y) \pmod{p}$$
$$t_y \equiv ay + b \pmod{p}$$

- The hash-function does not generate collisions before the $\pmod{n}$-operation. Furthermore, every choice (a, b) is mapped to a different pair (t_x, t_y) with $t_x := ax + b$ and $t_y := ay + b$.

  This holds because we can compute a and b when given t_x and t_y:

$$t_x \equiv ax + b \pmod{p}$$
$$t_y \equiv ay + b \pmod{p}$$

$$t_x - t_y \equiv a(x - y) \pmod{p}$$
$$t_y \equiv ay + b \pmod{p}$$

$$a \equiv (t_x - t_y)(x - y)^{-1} \pmod{p}$$
$$b \equiv t_y - ay \pmod{p}$$

Universal Hashing

There is a one-to-one correspondence between hash-functions (pairs (a, b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $\mathrm{mod}\, n$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $\mathrm{mod}\, n$ operation?

Fix a value t_x. There are $p - 1$ possible values for choosing t_y.

From the range $0, \ldots, p-1$ the values $t_x, t_x + n, t_x + 2n, \ldots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

Universal Hashing

There is a one-to-one correspondence between hash-functions (pairs (a, b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $\bmod n$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $\bmod n$ operation?

Fix a value t_x. There are $p - 1$ possible values for choosing t_y.

From the range $0, \ldots, p - 1$ the values $t_x, t_x + n, t_x + 2n, \ldots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

Universal Hashing

There is a one-to-one correspondence between hash-functions (pairs (a, b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $\bmod n$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $\bmod n$ operation?

Fix a value t_x. There are $p - 1$ possible values for choosing t_y.

From the range $0, \dots, p-1$ the values $t_x, t_x + n, t_x + 2n, \dots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

Universal Hashing

There is a one-to-one correspondence between hash-functions (pairs (a, b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $\bmod n$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $\bmod n$ operation?

Fix a value t_x. There are $p - 1$ possible values for choosing t_y.

From the range $0, \ldots, p - 1$ the values $t_x, t_x + n, t_x + 2n, \ldots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

Universal Hashing

There is a one-to-one correspondence between hash-functions (pairs (a, b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $\bmod n$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $\bmod n$ operation?

Fix a value t_x. There are $p - 1$ possible values for choosing t_y.

From the range $0, \dots, p-1$ the values $t_x, t_x + n, t_x + 2n, \dots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

Universal Hashing

There is a one-to-one correspondence between hash-functions (pairs (a, b), $a \neq 0$) and pairs (t_x, t_y), $t_x \neq t_y$.

Therefore, we can view the first step (before the $\bmod n$-operation) as choosing a pair (t_x, t_y), $t_x \neq t_y$ uniformly at random.

What happens when we do the $\bmod n$ operation?

Fix a value t_x. There are $p - 1$ possible values for choosing t_y.

From the range $0, \ldots, p - 1$ the values $t_x, t_x + n, t_x + 2n, \ldots$ map to t_x after the modulo-operation. These are at most $\lceil p/n \rceil$ values.

Universal Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

Universal Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

Universal Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

Universal Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

Universal Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

Universal Hashing

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \leq \frac{p}{n} + \frac{n-1}{n} - 1 \leq \frac{p-1}{n}$$

possibilities for choosing t_y such that the final hash-value creates a collision.

This happens with probability at most $\frac{1}{n}$.

Universal Hashing

Universal Hashing

It is also possible to show that $\mathcal{H}$ is an (almost) pairwise independent class of hash-functions.

$$\Pr_{t_x \neq t_y \in \mathbb{Z}_p^2}\left[\begin{array}{c} t_x \bmod n = h_1 \\ \wedge \\ t_y \bmod n = h_2 \end{array}\right]$$

Universal Hashing

It is also possible to show that $\mathcal{H}$ is an (almost) pairwise independent class of hash-functions.

$$\frac{\left\lfloor \frac{p}{n} \right\rfloor^2}{p(p-1)} \le \Pr_{t_x \ne t_y \in \mathbb{Z}_p^2}\left[\begin{array}{c} t_x \bmod n = h_1 \\ \wedge \\ t_y \bmod n = h_2 \end{array} \right] \le \frac{\left\lceil \frac{p}{n} \right\rceil^2}{p(p-1)}$$

Universal Hashing

It is also possible to show that $\mathcal{H}$ is an (almost) pairwise independent class of hash-functions.

$$\frac{\left\lfloor \frac{p}{n} \right\rfloor^2}{p(p-1)} \le \Pr_{t_x \neq t_y \in \mathbb{Z}_p^2}\left[\begin{array}{c} t_x \bmod n = h_1 \\ \wedge \\ t_y \bmod n = h_2 \end{array}\right] \le \frac{\left\lceil \frac{p}{n} \right\rceil^2}{p(p-1)}$$

Note that the middle is the probability that $h(x) = h_1$ and $h(y) = h_2$. The total number of choices for (t_x, t_y) is $p(p-1)$. The number of choices for t_x (t_y) such that $t_x \bmod n = h_1$ ($t_y \bmod n = h_2$) lies between $\lfloor \frac{p}{n} \rfloor$ and $\lceil \frac{p}{n} \rceil$.

Definition 22

Let $d \in \mathbb{N}$; $q \geq (d+1)n$ be a prime; and let $\vec{a} \in \{0, \dots, q-1\}^{d+1}$. Define for $x \in \{0, \dots, q\}$

$$h_{\vec{a}}(x) := \Big(\sum_{i=0}^{d} a_i x^i \bmod q \Big) \bmod n \ .$$

Let $\mathcal{H}_n^d := \{h_{\vec{a}} \mid \vec{a} \in \{0, \dots, q\}^{d+1}\}$. The class $\mathcal{H}_n^d$ is $(e, d+1)$-independent.

Note that in the previous case we had $d = 1$ and chose $a_d \neq 0$.

For the coefficients $\bar{a} \in \{0, \dots, q-1\}^{d+1}$ let $f_{\bar{a}}$ denote the polynomial

$$f_{\bar{a}}(x) = \Big(\sum_{i=0}^{d} a_i x^i\Big) \bmod q$$

The polynomial is defined by $d+1$ distinct points.

For the coefficients $\bar{a} \in \{0, \dots, q-1\}^{d+1}$ let $f_{\bar{a}}$ denote the polynomial

$$f_{\bar{a}}(x) = \Big(\sum_{i=0}^{d} a_i x^i \Big) \bmod q$$

The polynomial is defined by $d+1$ distinct points.

For the coefficients $\bar{a} \in \{0, \dots, q-1\}^{d+1}$ let $f_{\bar{a}}$ denote the polynomial

$$f_{\bar{a}}(x) = \Big(\sum_{i=0}^{d} a_i x^i\Big) \bmod q$$

The polynomial is defined by $d+1$ distinct points.

Fix $\ell \leq d+1$; let $x_1,\dots,x_\ell \in \{0,\dots,q-1\}$ be keys, and let $t_1,\dots,t_\ell$ denote the corresponding hash-function values.

Let $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i \text{ for all } i \in \{1,\dots,\ell\}\}$
Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \underbrace{\{t_i + \alpha \cdot n \mid \alpha \in \{0,\dots,\lceil \tfrac{q}{n} \rceil - 1\}\}}_{=:B_i}$$

In order to obtain the cardinality of A^ℓ we choose our polynomial by fixing $d+1$ points.

We first fix the values for inputs $x_1,\dots,x_\ell$.
We have

$$|B_1| \cdot \ldots \cdot |B_\ell|$$

possibilities to do this (so that $h_{\bar{a}}(x_i) = t_i$).

Fix $\ell \leq d+1$; let $x_1, \ldots, x_\ell \in \{0, \ldots, q-1\}$ be keys, and let $t_1, \ldots, t_\ell$ denote the corresponding hash-function values.

Let $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i$ for all $i \in \{1, \ldots, \ell\}\}$
Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \underbrace{\{t_i + \alpha \cdot n \mid \alpha \in \{0, \ldots, \lceil \tfrac{q}{n} \rceil - 1\}\}}_{=: B_i}$$

In order to obtain the cardinality of A^ℓ we choose our polynomial by fixing $d+1$ points.

We first fix the values for inputs $x_1, \ldots, x_\ell$.
We have

$$|B_1| \cdot \ldots \cdot |B_\ell|$$

possibilities to do this (so that $h_{\bar{a}}(x_i) = t_i$).

Fix $\ell \leq d+1$; let $x_1, \ldots, x_\ell \in \{0, \ldots, q-1\}$ be keys, and let $t_1, \ldots, t_\ell$ denote the corresponding hash-function values.

Let $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i \text{ for all } i \in \{1, \ldots, \ell\}\}$

Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \underbrace{\{t_i + \alpha \cdot n \mid \alpha \in \{0, \ldots, \lceil \tfrac{q}{n} \rceil - 1\}\}}_{=:B_i}$$

In order to obtain the cardinality of A^ℓ we choose our polynomial by fixing $d+1$ points.

We first fix the values for inputs $x_1, \ldots, x_\ell$.

We have

$$|B_1| \cdot \ldots \cdot |B_\ell|$$

possibilities to do this (so that $h_{\bar{a}}(x_i) = t_i$).

Fix $\ell \leq d+1$; let $x_1, \dots, x_\ell \in \{0, \dots, q-1\}$ be keys, and let $t_1, \dots, t_\ell$ denote the corresponding hash-function values.

Let $A^\ell = \{h_{\bar{a}} \in \mathcal{H} \mid h_{\bar{a}}(x_i) = t_i \text{ for all } i \in \{1, \dots, \ell\}\}$
Then

$$h_{\bar{a}} \in A^\ell \Leftrightarrow h_{\bar{a}} = f_{\bar{a}} \bmod n \text{ and}$$

$$f_{\bar{a}}(x_i) \in \underbrace{\{t_i + \alpha \cdot n \mid \alpha \in \{0, \dots, \lceil \tfrac{q}{n} \rceil - 1\}\}}_{=: B_i}$$

In order to obtain the cardinality of A^ℓ we choose our polynomial by fixing $d+1$ points.

We first fix the values for inputs $x_1, \dots, x_\ell$.
We have

$$|B_1| \cdot \ldots \cdot |B_\ell|$$

possibilities to do this (so that $h_{\bar{a}}(x_i) = t_i$).

Now, we choose $d-\ell+1$ other inputs and choose their value arbitrarily. We have $q^{d-\ell+1}$ possibilities to do this.

Therefore we have

$$|B_1|\cdot\ldots\cdot|B_\ell|\cdot q^{d-\ell+1}\le\lceil\frac{q}{n}\rceil^\ell\cdot q^{d-\ell+1}$$

possibilities to choose $\bar{a}$ such that $h_{\bar{a}}\in A_\ell$.

Now, we choose $d - \ell + 1$ other inputs and choose their value arbitrarily. We have $q^{d-\ell+1}$ possibilities to do this.

Therefore we have

$$|B_1| \cdot \ldots \cdot |B_\ell| \cdot q^{d-\ell+1} \le \lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}$$

possibilities to choose $\bar{a}$ such that $h_{\bar{a}} \in A_\ell$.

Therefore the probability of choosing $h_{\bar{a}}$ from A_ℓ is only

$$\frac{\lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}}{q^{d+1}}$$

This shows that the $\mathcal{H}$ is $(e, d+1)$-universal.

Therefore the probability of choosing $h_{\bar{a}}$ from A_ℓ is only

$$\frac{\lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}}{q^{d+1}} \leq \frac{(\frac{q+n}{n})^\ell}{q^\ell}$$

This shows that the $\mathcal{H}$ is $(e, d+1)$-universal.

Therefore the probability of choosing $h_{\bar{a}}$ from A_ℓ is only

$$\frac{\lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}}{q^{d+1}} \le \frac{(\frac{q+n}{n})^\ell}{q^\ell} \le \left(\frac{q+n}{q}\right)^\ell \cdot \frac{1}{n^\ell}$$

This shows that the $\mathcal{H}$ is $(e, d+1)$-universal.

Therefore the probability of choosing $h_{\bar{a}}$ from A_ℓ is only

$$\frac{\lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}}{q^{d+1}} \le \frac{(\frac{q+n}{n})^\ell}{q^\ell} \le \Big(\frac{q+n}{q}\Big)^\ell \cdot \frac{1}{n^\ell}$$
$$\le \Big(1 + \frac{1}{\ell}\Big)^\ell \cdot \frac{1}{n^\ell}$$

This shows that the $\mathcal{H}$ is $(e, d+1)$-universal.

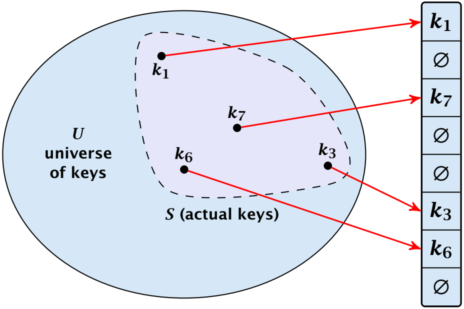
Therefore the probability of choosing $h_{\bar{a}}$ from A_ℓ is only

$$\frac{\lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}}{q^{d+1}} \le \frac{(\frac{q+n}{n})^\ell}{q^\ell} \le \left(\frac{q+n}{q}\right)^\ell \cdot \frac{1}{n^\ell}$$
$$\le \left(1+\frac{1}{\ell}\right)^\ell \cdot \frac{1}{n^\ell} \le \frac{e}{n^\ell} \quad .$$

This shows that the $\mathcal{H}$ is $(e, d+1)$-universal.

Perfect Hashing

Suppose that we **know** the set S of actual keys (no insert/no delete). Then we may want to design a **simple** hash-function that maps all these keys to different memory locations.



Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} \ .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} \enspace .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} \ .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} \ .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

Let $m = |S|$. We could simply choose the hash-table size very large so that we don't get any collisions.

Using a universal hash-function the expected number of collisions is

$$\mathrm{E}[\#\text{Collisions}] = \binom{m}{2} \cdot \frac{1}{n} \enspace .$$

If we choose $n = m^2$ the expected number of collisions is strictly less than $\frac{1}{2}$.

Can we get an upper bound on the probability of having collisions?

The probability of having 1 or more collisions can be at most $\frac{1}{2}$ as otherwise the expectation would be larger than $\frac{1}{2}$.

Perfect Hashing

We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j-th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2. The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing

We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j-th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2. The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing

We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j-th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2. The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing

We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j-th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2. The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing

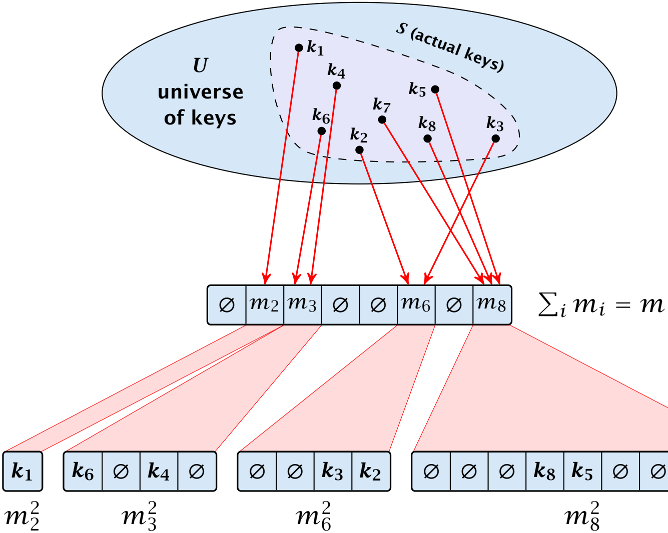
We can find such a hash-function by a few trials.

However, a hash-table size of $n = m^2$ is very very high.

We construct a two-level scheme. We first use a hash-function that maps elements from S to m buckets.

Let m_j denote the number of items that are hashed to the j-th bucket. For each bucket we choose a second hash-function that maps the elements of the bucket into a table of size m_j^2. The second function can be chosen such that all elements are mapped to different locations.

Perfect Hashing



Perfect Hashing

Perfect Hashing

The total memory that is required by all hash-tables is $\mathcal{O}(\sum_j m_j^2)$. Note that m_j is a random variable.

$$\mathrm{E}\left[\sum_j m_j^2\right]$$

Perfect Hashing

The total memory that is required by all hash-tables is $\mathcal{O}(\sum_j m_j^2)$. Note that m_j is a random variable.

$$\mathrm{E}\Big[\sum_j m_j^2\Big] = \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big]$$

Perfect Hashing

The total memory that is required by all hash-tables is $\mathcal{O}(\sum_j m_j^2)$. Note that m_j is a random variable.

$$\begin{aligned}\mathrm{E}\Big[\sum_j m_j^2\Big] &= \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big] \\ &= 2\,\mathrm{E}\Big[\sum_j \binom{m_j}{2}\Big] + \mathrm{E}\Big[\sum_j m_j\Big]\end{aligned}$$

Perfect Hashing

The total memory that is required by all hash-tables is $\mathcal{O}(\sum_j m_j^2)$. Note that m_j is a random variable.

$$
\begin{aligned}
\mathrm{E}\Big[\sum_j m_j^2\Big] &= \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big] \\
&= 2\,\mathrm{E}\Big[\sum_j \binom{m_j}{2}\Big] + \mathrm{E}\Big[\sum_j m_j\Big]
\end{aligned}
$$

The first expectation is simply the expected number of collisions, for the first level. Since we use universal hashing we have

Perfect Hashing

The total memory that is required by all hash-tables is $\mathcal{O}(\sum_j m_j^2)$. Note that m_j is a random variable.

$$\mathrm{E}\Big[\sum_j m_j^2\Big] = \mathrm{E}\Big[2\sum_j \binom{m_j}{2} + \sum_j m_j\Big]$$
$$= 2\,\mathrm{E}\Big[\sum_j \binom{m_j}{2}\Big] + \mathrm{E}\Big[\sum_j m_j\Big]$$

The first expectation is simply the expected number of collisions, for the first level. Since we use universal hashing we have

$$= 2\binom{m}{2}\frac{1}{m} + m = 2m - 1\ .$$

Perfect Hashing

We need only $\mathcal{O}(m)$ time to construct a hash-function h with $\sum_j m_j^2 = \mathcal{O}(4m)$, because with probability at least $1/2$ a random function from a universal family will have this property.

Then we construct a hash-table h_j for every bucket. This takes expected time $\mathcal{O}(m_j)$ for every bucket. A random function h_j is collision-free with probability at least $1/2$. We need $\mathcal{O}(m_j)$ to test this.

We only need that the hash-functions are chosen from a universal family!!!

Cuckoo Hashing

Goal:
Try to generate a hash-table with constant worst-case search time in a dynamic scenario.

- Two hash-tables $T_1[0, \dots, n-1]$ and $T_2[0, \dots, n-1]$, with hash-functions h_1, and h_2.
- An object x is either stored at location $T_1[h_1(x)]$ or $T_2[h_2(x)]$.
- A search clearly takes constant time if the above constraint is met.

Cuckoo Hashing

Goal:
Try to generate a hash-table with constant worst-case search time in a dynamic scenario.

- Two hash-tables $T_1[0,\ldots,n-1]$ and $T_2[0,\ldots,n-1]$, with hash-functions h_1, and h_2.
- An object x is either stored at location $T_1[h_1(x)]$ or $T_2[h_2(x)]$.
- A search clearly takes constant time if the above constraint is met.

Cuckoo Hashing

Goal:

Try to generate a hash-table with constant worst-case search time in a dynamic scenario.

- Two hash-tables $T_1[0, \dots, n-1]$ and $T_2[0, \dots, n-1]$, with hash-functions h_1, and h_2.
- An object x is either stored at location $T_1[h_1(x)]$ or $T_2[h_2(x)]$.
- A search clearly takes constant time if the above constraint is met.

Cuckoo Hashing

Goal:
Try to generate a hash-table with constant worst-case search time in a dynamic scenario.

- Two hash-tables $T_1[0,\dots,n-1]$ and $T_2[0,\dots,n-1]$, with hash-functions h_1, and h_2.
- An object x is either stored at location $T_1[h_1(x)]$ or $T_2[h_2(x)]$.
- A search clearly takes constant time if the above constraint is met.

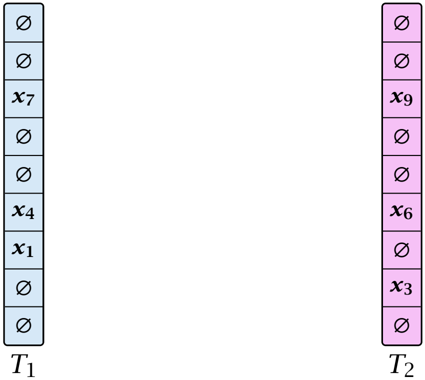
Cuckoo Hashing

Goal:
Try to generate a hash-table with constant worst-case search time in a dynamic scenario.

- Two hash-tables $T_1[0,\ldots,n-1]$ and $T_2[0,\ldots,n-1]$, with hash-functions h_1, and h_2.
- An object x is either stored at location $T_1[h_1(x)]$ or $T_2[h_2(x)]$.
- A search clearly takes constant time if the above constraint is met.

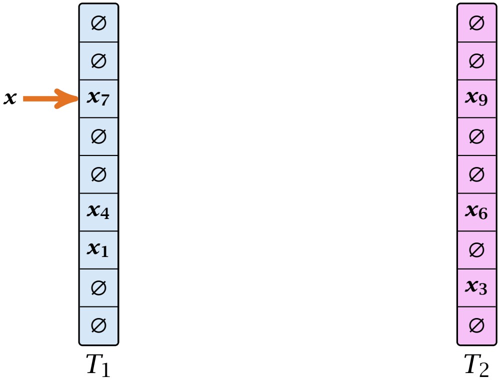
Cuckoo Hashing

Insert:



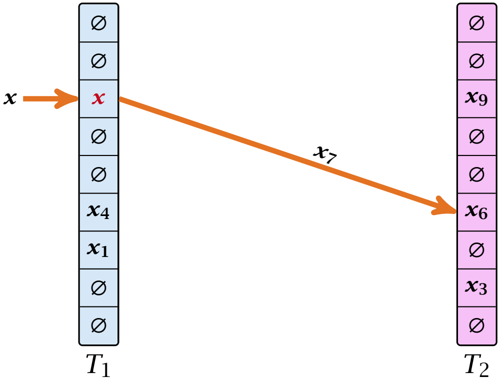
Cuckoo Hashing

Insert:



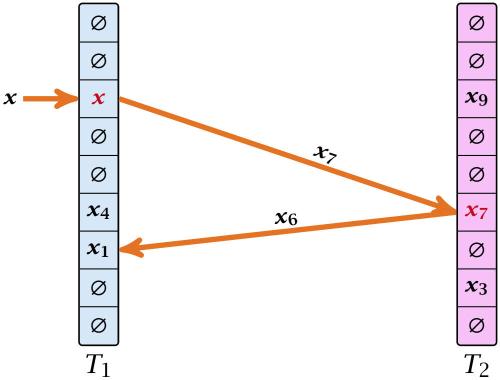
Cuckoo Hashing

Insert:



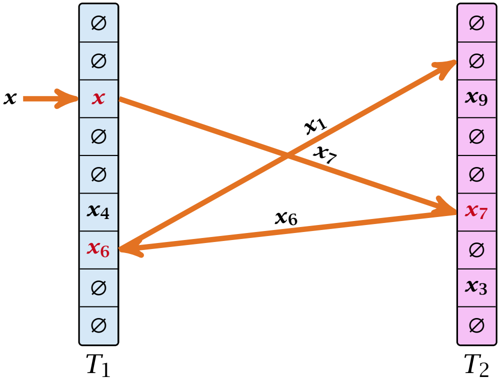
Cuckoo Hashing

Insert:



Cuckoo Hashing

Insert:



Cuckoo Hashing

Algorithm 17 Cuckoo-Insert(x)

```
if T_1[h_1(x)] = x ∨ T_2[h_2(x)] = x then return
steps ← 1
while steps ≤ maxsteps do
    exchange x and T_1[h_1(x)]
    if x = null then return
    exchange x and T_2[h_2(x)]
    if x = null then return
    steps ← steps +1
rehash() // change hash-functions; rehash everything
Cuckoo-Insert(x)
```

Cuckoo Hashing

- We call one iteration through the while-loop a step of the algorithm.
- We call a sequence of iterations through the while-loop without the termination condition becoming true a phase of the algorithm.
- We say a phase is successful if it is not terminated by the maxstep-condition, but the while loop is left because $x = \text{null}$.

Cuckoo Hashing

- We call one iteration through the while-loop a step of the algorithm.
- We call a sequence of iterations through the while-loop without the termination condition becoming true a phase of the algorithm.
- We say a phase is successful if it is not terminated by the maxstep-condition, but the while loop is left because $x = \text{null}$.

Cuckoo Hashing

- We call one iteration through the while-loop a step of the algorithm.
- We call a sequence of iterations through the while-loop without the termination condition becoming true a phase of the algorithm.
- We say a phase is successful if it is not terminated by the $\mathrm{maxstep}$-condition, but the while loop is left because $x = \mathrm{null}$.

Cuckoo Hashing

What is the expected time for an insert-operation?

We first analyze the probability that we end-up in an infinite loop (that is then terminated after $\mathrm{maxsteps}$ steps).

Formally what is the probability to enter an infinite loop that touches s different keys?

Cuckoo Hashing

What is the expected time for an insert-operation?

We first analyze the probability that we end-up in an infinite loop (that is then terminated after $\mathrm{maxsteps}$ steps).

Formally what is the probability to enter an infinite loop that touches s different keys?

Cuckoo Hashing

What is the expected time for an insert-operation?

We first analyze the probability that we end-up in an infinite loop (that is then terminated after maxsteps steps).

Formally what is the probability to enter an infinite loop that touches s different keys?

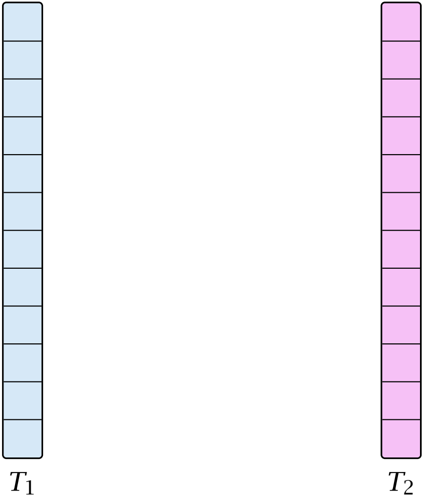
Cuckoo Hashing

What is the expected time for an insert-operation?

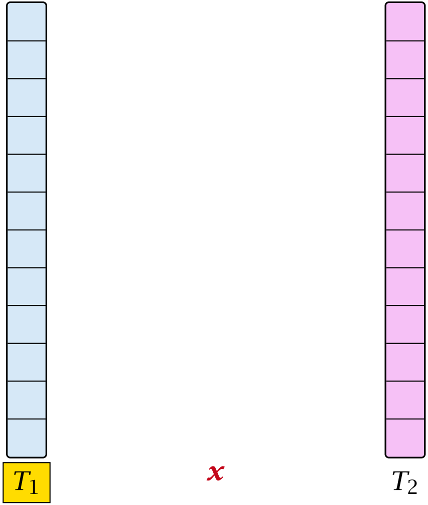
We first analyze the probability that we end-up in an infinite loop (that is then terminated after maxsteps steps).

Formally what is the probability to enter an infinite loop that touches s different keys?

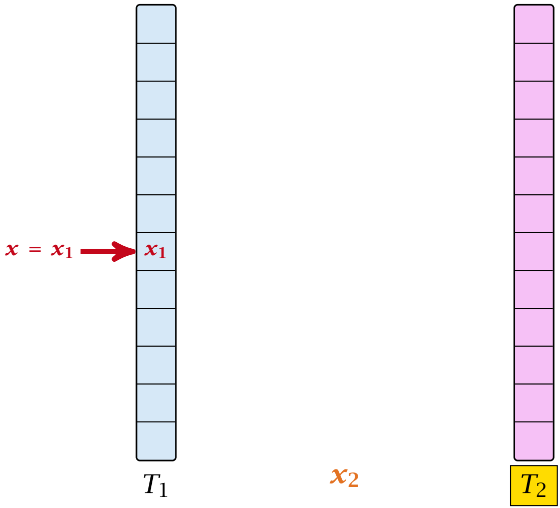
Cuckoo Hashing: Insert



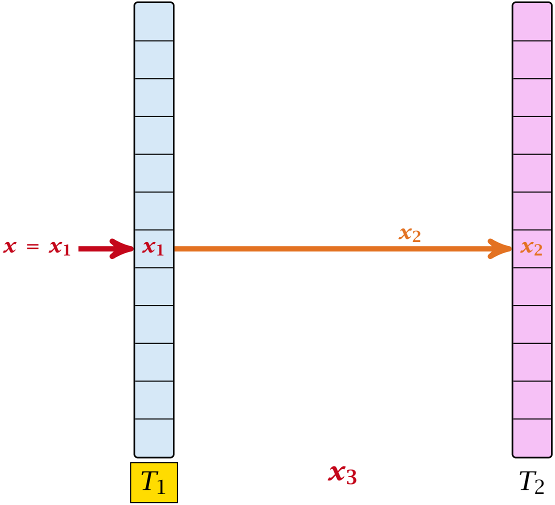
Cuckoo Hashing: Insert



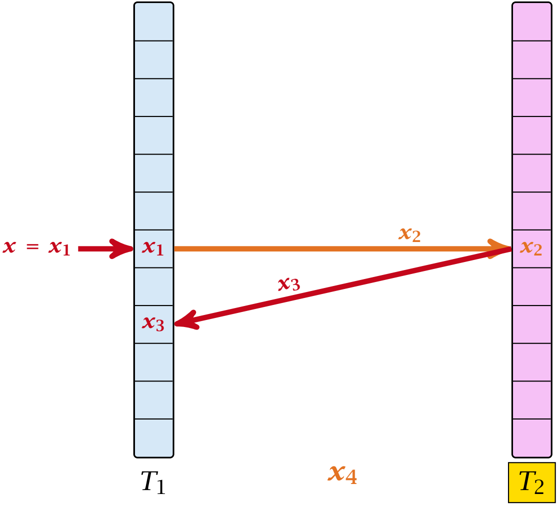
Cuckoo Hashing: Insert



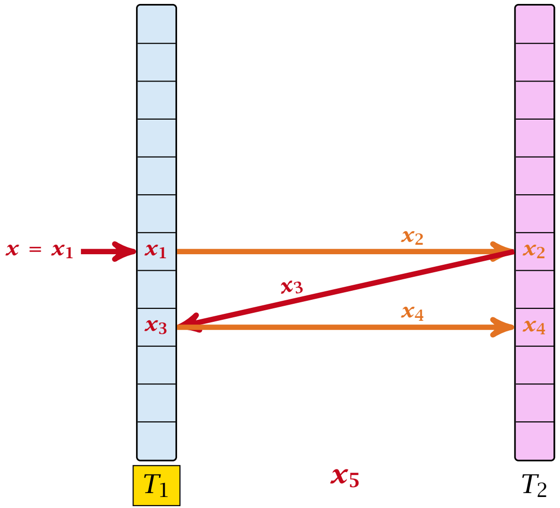
Cuckoo Hashing: Insert



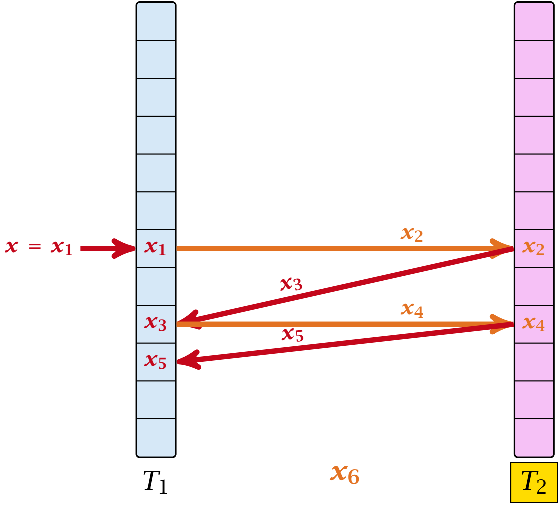
Cuckoo Hashing: Insert



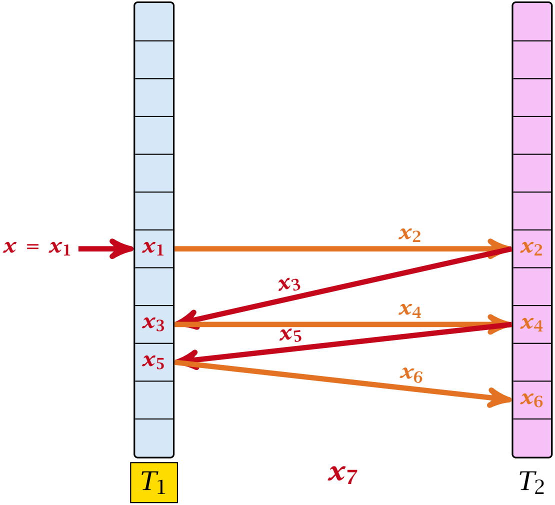
Cuckoo Hashing: Insert



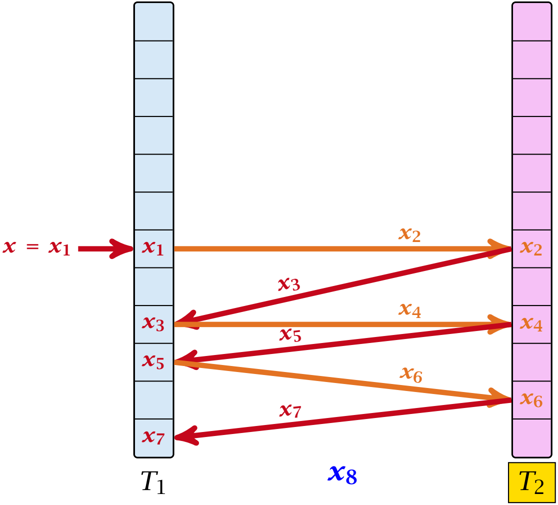
Cuckoo Hashing: Insert



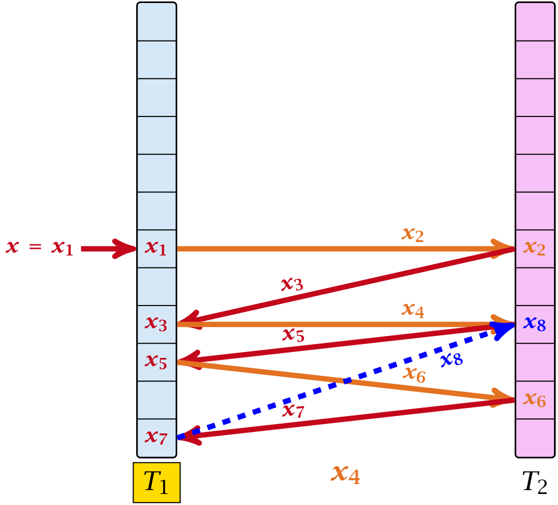
Cuckoo Hashing: Insert



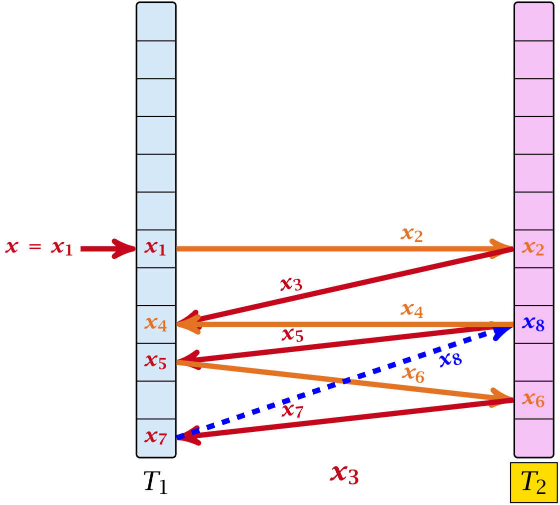
Cuckoo Hashing: Insert



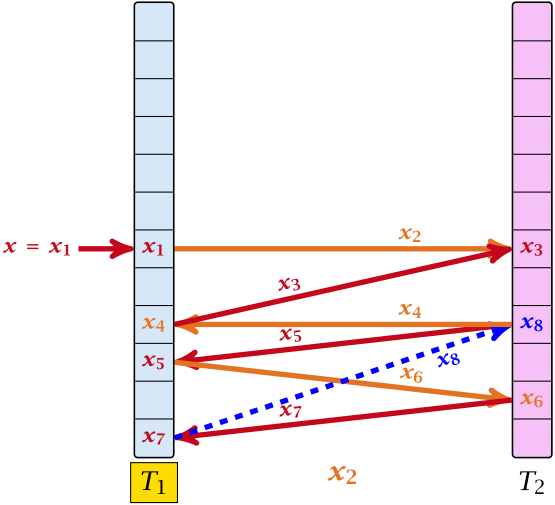
Cuckoo Hashing: Insert



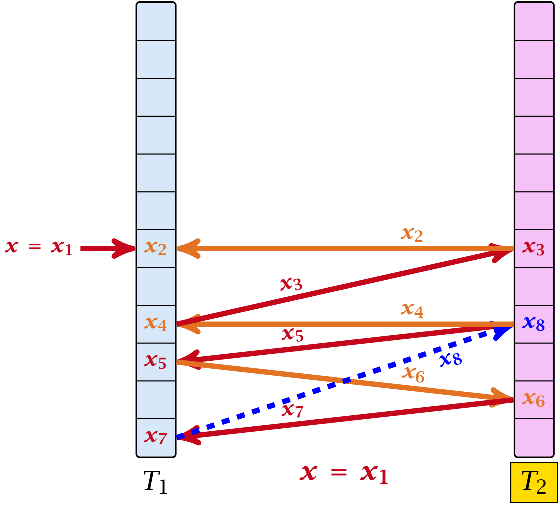
Cuckoo Hashing: Insert



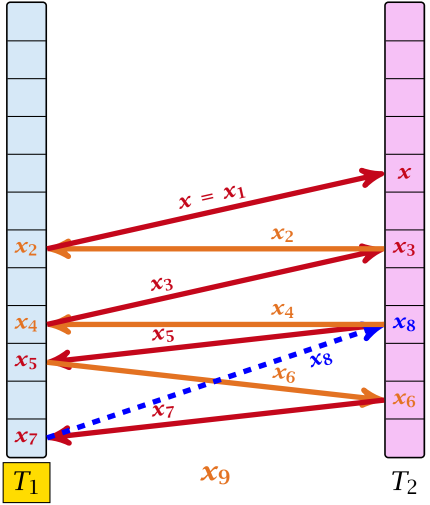
Cuckoo Hashing: Insert



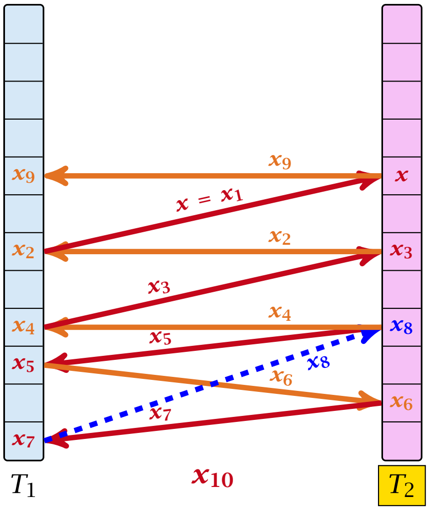
Cuckoo Hashing: Insert



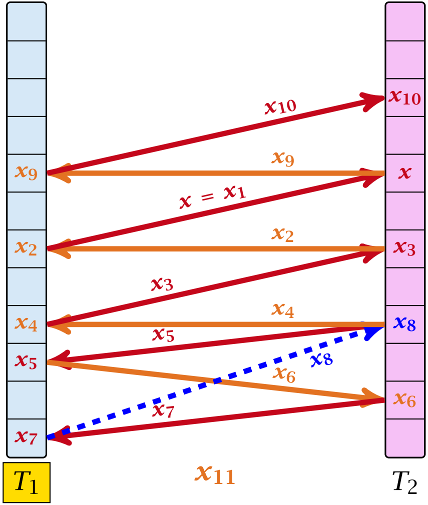
Cuckoo Hashing: Insert



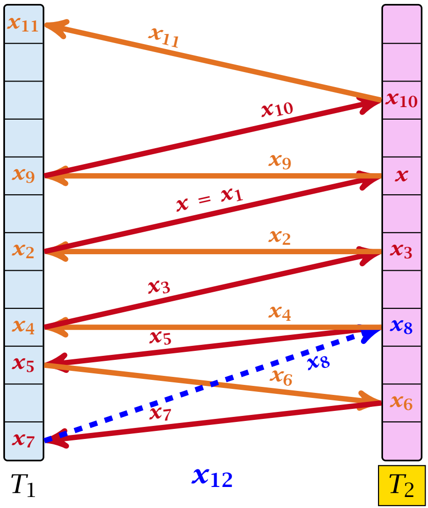
Cuckoo Hashing: Insert



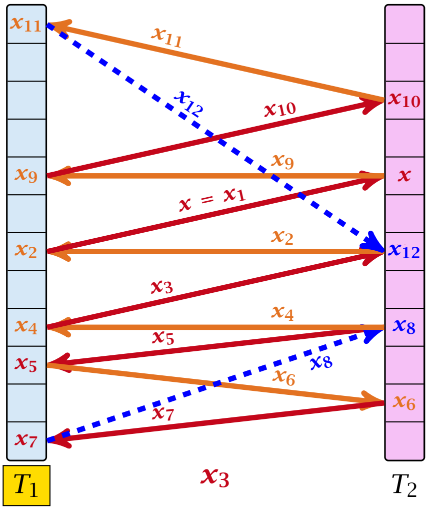
Cuckoo Hashing: Insert



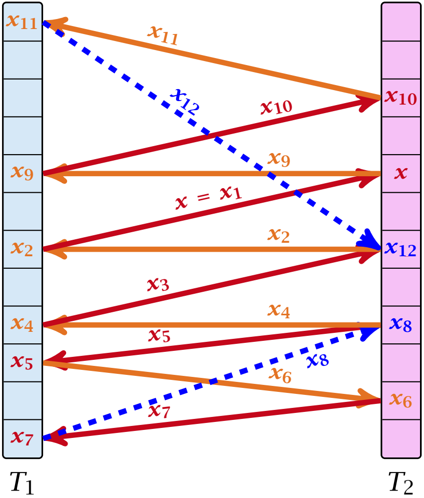
Cuckoo Hashing: Insert



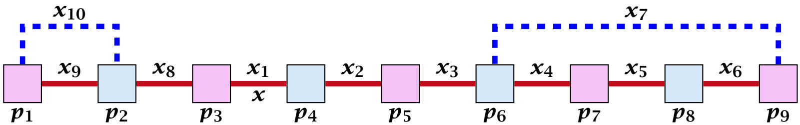
Cuckoo Hashing: Insert



Cuckoo Hashing: Insert



Cuckoo Hashing



Cuckoo Hashing

A cycle-structure is active if for every key x_ℓ (linking a cell p_i from T_1 and a cell p_j from T_2) we have

$$h_1(x_\ell) = p_i \qquad \text{and} \qquad h_2(x_\ell) = p_j$$

Observation:

If during a phase the insert-procedure runs into a cycle there must exist an active cycle structure of size $s \geq 3$.

Cuckoo Hashing

A cycle-structure is active if for every key x_ℓ (linking a cell p_i from T_1 and a cell p_j from T_2) we have

$$h_1(x_\ell) = p_i \qquad \text{and} \qquad h_2(x_\ell) = p_j$$

Observation:

If during a phase the insert-procedure runs into a cycle there must exist an active cycle structure of size $s \geq 3$.

Cuckoo Hashing

What is the probability that all keys in a cycle-structure of size s correctly map into their T_1-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_1 is a (μ, s)-independent hash-function.

What is the probability that all keys in the cycle-structure of size s correctly map into their T_2-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_2 is a (μ, s)-independent hash-function.

These events are independet.

Cuckoo Hashing

What is the probability that all keys in a cycle-structure of size s correctly map into their T_1-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_1 is a (μ, s)-independent hash-function.

What is the probability that all keys in the cycle-structure of size s correctly map into their T_2-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_2 is a (μ, s)-independent hash-function.

These events are independept.

Cuckoo Hashing

What is the probability that all keys in a cycle-structure of size s correctly map into their T_1-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_1 is a (μ, s)-independent hash-function.

What is the probability that all keys in the cycle-structure of size s correctly map into their T_2-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_2 is a (μ, s)-independent hash-function.

These events are independet.

Cuckoo Hashing

What is the probability that all keys in a cycle-structure of size s correctly map into their T_1-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_1 is a (μ, s)-independent hash-function.

What is the probability that all keys in the cycle-structure of size s correctly map into their T_2-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_2 is a (μ, s)-independent hash-function.

These events are independept.

Cuckoo Hashing

What is the probability that all keys in a cycle-structure of size s correctly map into their T_1-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_1 is a (μ, s)-independent hash-function.

What is the probability that all keys in the cycle-structure of size s correctly map into their T_2-cell?

This probability is at most $\frac{\mu}{n^s}$ since h_2 is a (μ, s)-independent hash-function.

These events are independet.

Cuckoo Hashing

The probability that a given cycle-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

What is the probability that there exists an active cycle structure of size s?

Cuckoo Hashing

The probability that a given cycle-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

What is the probability that there exists an active cycle structure of size s?

Cuckoo Hashing

The number of cycle-structures of size s is at most

$$s^3 \cdot n^{s-1} \cdot m^{s-1} \ .$$

- There are at most s^2 possibilities where to attach the forward and backward links.
- There are at most s possibilities to choose where to place key x.
- There are m^{s-1} possibilities to choose the keys apart from x.
- There are n^{s-1} possibilities to choose the cells.

Cuckoo Hashing

The number of cycle-structures of size s is at most

$$s^3 \cdot n^{s-1} \cdot m^{s-1} \ .$$

- There are at most s^2 possibilities where to attach the forward and backward links.
- There are at most s possibilities to choose where to place key x.
- There are m^{s-1} possibilities to choose the keys apart from x.
- There are n^{s-1} possibilities to choose the cells.

Cuckoo Hashing

The number of cycle-structures of size s is at most

$$s^3 \cdot n^{s-1} \cdot m^{s-1} \; .$$

- There are at most s^2 possibilities where to attach the forward and backward links.
- There are at most s possibilities to choose where to place key x.
- There are m^{s-1} possibilities to choose the keys apart from x.
- There are n^{s-1} possibilities to choose the cells.

Cuckoo Hashing

The number of cycle-structures of size s is at most

$$s^3 \cdot n^{s-1} \cdot m^{s-1} \ .$$

- There are at most s^2 possibilities where to attach the forward and backward links.
- There are at most s possibilities to choose where to place key x.
- There are m^{s-1} possibilities to choose the keys apart from x.
- There are n^{s-1} possibilities to choose the cells.

Cuckoo Hashing

The number of cycle-structures of size s is at most

$$s^3 \cdot n^{s-1} \cdot m^{s-1} \ .$$

- There are at most s^2 possibilities where to attach the forward and backward links.
- There are at most s possibilities to choose where to place key x.
- There are m^{s-1} possibilities to choose the keys apart from x.
- There are n^{s-1} possibilities to choose the cells.

Cuckoo Hashing

The probability that there exists an active cycle-structure is therefore at most

$$\sum_{s=3}^{\infty} s^3 \cdot n^{s-1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}}$$

Cuckoo Hashing

The probability that there exists an active cycle-structure is therefore at most

$$\sum_{s=3}^{\infty} s^3 \cdot n^{s-1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} = \frac{\mu^2}{nm} \sum_{s=3}^{\infty} s^3 \left(\frac{m}{n}\right)^s$$

Cuckoo Hashing

The probability that there exists an active cycle-structure is therefore at most

$$\sum_{s=3}^{\infty} s^3 \cdot n^{s-1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} = \frac{\mu^2}{nm} \sum_{s=3}^{\infty} s^3 \left(\frac{m}{n}\right)^s$$

$$\leq \frac{\mu^2}{m^2} \sum_{s=3}^{\infty} s^3 \left(\frac{1}{1+\epsilon}\right)^s$$

Cuckoo Hashing

The probability that there exists an active cycle-structure is therefore at most

$$\sum_{s=3}^{\infty} s^3 \cdot n^{s-1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} = \frac{\mu^2}{nm} \sum_{s=3}^{\infty} s^3 \left(\frac{m}{n}\right)^s$$

$$\leq \frac{\mu^2}{m^2} \sum_{s=3}^{\infty} s^3 \left(\frac{1}{1+\epsilon}\right)^s \leq \mathcal{O}\left(\frac{1}{m^2}\right) \quad .$$

Cuckoo Hashing

The probability that there exists an active cycle-structure is therefore at most

$$\sum_{s=3}^{\infty} s^3 \cdot n^{s-1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} = \frac{\mu^2}{nm} \sum_{s=3}^{\infty} s^3 \left(\frac{m}{n}\right)^s$$

$$\leq \frac{\mu^2}{m^2} \sum_{s=3}^{\infty} s^3 \left(\frac{1}{1+\epsilon}\right)^s \leq \mathcal{O}\left(\frac{1}{m^2}\right) \quad .$$

Here we used the fact that $(1+\epsilon)m \leq n$.

Cuckoo Hashing

The probability that there exists an active cycle-structure is therefore at most

$$\sum_{s=3}^{\infty} s^3 \cdot n^{s-1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} = \frac{\mu^2}{nm} \sum_{s=3}^{\infty} s^3 \left(\frac{m}{n}\right)^s$$
$$\le \frac{\mu^2}{m^2} \sum_{s=3}^{\infty} s^3 \left(\frac{1}{1+\epsilon}\right)^s \le \mathcal{O}\left(\frac{1}{m^2}\right) \quad .$$

Here we used the fact that $(1+\epsilon)m \le n$.

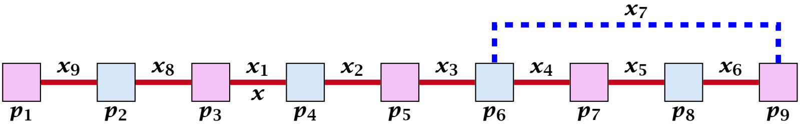
Hence,

$$\Pr[\text{cycle}] = \mathcal{O}\left(\frac{1}{m^2}\right) \quad .$$

Cuckoo Hashing

Now, we analyze the probability that a phase is not successful without running into a closed cycle.

Cuckoo Hashing



Sequence of visited keys:

$x = x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_3, x_2, x_1 = x, x_8, x_9, \ldots$

Cuckoo Hashing

Consider the sequence of not necessarily distinct keys starting with x in the order that they are visited during the phase.

Lemma 23

If the sequence is of length p then there exists a sub-sequence of at least $p/3$ keys starting with x of distinct keys.

Proof.

x is contained at most twice in the sequence.

Either the sub-sequence starting from x until right before the first repeated key, or the sub-sequence starting from the repetition of x until the end must contain at least $p/3$ distinct keys. □

Cuckoo Hashing

Consider the sequence of not necessarily distinct keys starting with x in the order that they are visited during the phase.

Lemma 23

If the sequence is of length p then there exists a sub-sequence of at least $p/3$ keys starting with x of distinct keys.

Proof.

x is contained at most twice in the sequence.

Either the sub-sequence starting from x until right before the first repeated key, or the sub-sequence starting from the repetition of x until the end must contain at least $p/3$ distinct keys. □

Cuckoo Hashing

Consider the sequence of not necessarily distinct keys starting with x in the order that they are visited during the phase.

Lemma 23

If the sequence is of length p then there exists a sub-sequence of at least $p/3$ keys starting with x of distinct keys.

Proof.

x is contained at most twice in the sequence.

Either the sub-sequence starting from x until right before the first repeated key, or the sub-sequence starting from the repetition of x until the end must contain at least $p/3$ distinct keys. □

Cuckoo Hashing

Consider the sequence of not necessarily distinct keys starting with x in the order that they are visited during the phase.

Lemma 23

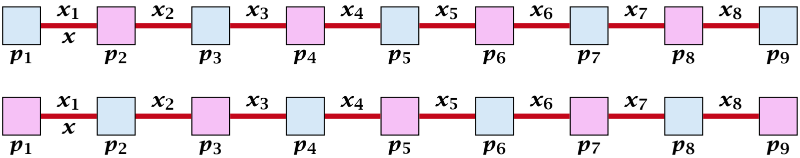
If the sequence is of length p then there exists a sub-sequence of at least $p/3$ keys starting with x of distinct keys.

Proof.

x is contained at most twice in the sequence.

Either the sub-sequence starting from x until right before the first repeated key, or the sub-sequence starting from the repetition of x until the end must contain at least $p/3$ distinct keys. □

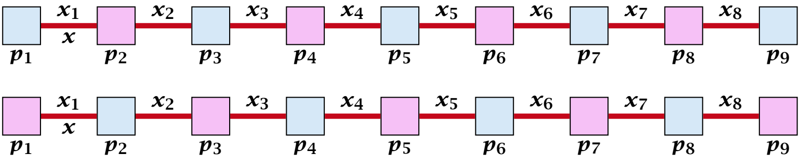
Cuckoo Hashing



A path-structure of size s is defined by

- $s+1$ different cells (alternating btw. cells from T_1 and T_2).
- s distinct keys $x = x_1, x_2, \ldots, x_s$, linking the cells.
- The leftmost cell is either from T_1 or T_2.

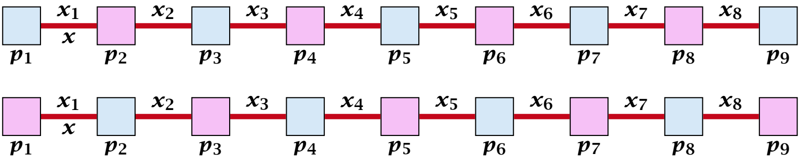
Cuckoo Hashing



A path-structure of size s is defined by

- $s+1$ different cells (alternating btw. cells from T_1 and T_2).
- s distinct keys $x = x_1, x_2, \ldots, x_s$, linking the cells.
- The leftmost cell is either from T_1 or T_2.

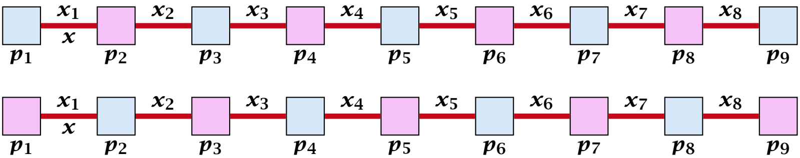
Cuckoo Hashing



A path-structure of size s is defined by

- $s+1$ different cells (alternating btw. cells from T_1 and T_2).
- s distinct keys $x = x_1, x_2, \ldots, x_s$, linking the cells.
- The leftmost cell is either from T_1 or T_2.

Cuckoo Hashing



A path-structure of size s is defined by

- $s + 1$ different cells (alternating btw. cells from T_1 and T_2).
- s distinct keys $x = x_1, x_2, \ldots, x_s$, linking the cells.
- The leftmost cell is either from T_1 or T_2.

Cuckoo Hashing

A path-structure is active if for every key x_ℓ (linking a cell p_i from T_1 and a cell p_j from T_2) we have

$$h_1(x_\ell) = p_i \qquad \text{and} \qquad h_2(x_\ell) = p_j$$

Observation:

If a phase takes at least t steps without running into a cycle there must exist an active path-structure of size $(2t-1)/3$.

Cuckoo Hashing

The probability that a given path-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

Cuckoo Hashing

The probability that a given path-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

The probability that there exists an active path-structure of size s is at most

$$2 \cdot n^{s+1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}}$$

Cuckoo Hashing

The probability that a given path-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

The probability that there exists an active path-structure of size s is at most

$$2 \cdot n^{s+1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} \leq 2\mu^2 \left(\frac{m}{n}\right)^{s-1}$$

Cuckoo Hashing

The probability that a given path-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

The probability that there exists an active path-structure of size s is at most

$$2 \cdot n^{s+1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} \leq 2\mu^2 \left(\frac{m}{n}\right)^{s-1} \leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{s-1}$$

Cuckoo Hashing

The probability that a given path-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

The probability that there exists an active path-structure of size s is at most

$$
\begin{aligned}
2 \cdot n^{s+1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} & \\
&\leq 2\mu^2 \left(\frac{m}{n}\right)^{s-1} \leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{s-1} \\
&\leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{(2t-1)/3-1}
\end{aligned}
$$

Cuckoo Hashing

The probability that a given path-structure of size s is active is at most $\frac{\mu^2}{n^{2s}}$.

The probability that there exists an active path-structure of size s is at most

$$
\begin{aligned}
2 \cdot n^{s+1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} & \\
&\leq 2\mu^2 \left(\frac{m}{n}\right)^{s-1} \leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{s-1} \\
&\leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{(2t-1)/3-1} = 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{2(t-2)/3} .
\end{aligned}
$$

Cuckoo Hashing

We choose maxsteps $\geq \frac{3\ell}{2} + 1$.

Cuckoo Hashing

We choose $\text{maxsteps} \geq \frac{3\ell}{2} + 1$. Then the probability that a phase is terminated unsuccessfully without running into a cycle is at most

$$\Pr[\text{unsuccessful} \mid \text{no cycle}]$$

Cuckoo Hashing

We choose $\text{maxsteps} \geq \frac{3\ell}{2} + 1$. Then the probability that a phase is terminated unsuccessfully without running into a cycle is at most

$$\begin{aligned} &\Pr[\text{unsuccessful} \mid \text{no cycle}] \\ &\qquad \leq \Pr[\exists \text{ active path-structure of size at least } \tfrac{2(\text{maxsteps}+1)-1}{3}] \end{aligned}$$

Cuckoo Hashing

We choose $\text{maxsteps} \geq \frac{3\ell}{2} + 1$. Then the probability that a phase is terminated unsuccessfully without running into a cycle is at most

$$\begin{aligned} &\Pr[\text{unsuccessful} \mid \text{no cycle}] \\ &\qquad \leq \Pr[\exists \text{ active path-structure of size at least } \tfrac{2(\text{maxsteps}+1)-1}{3}] \\ &\qquad \leq \Pr[\exists \text{ active path-structure of size at least } \ell + 1] \end{aligned}$$

Cuckoo Hashing

We choose $\text{maxsteps} \geq \frac{3\ell}{2} + 1$. Then the probability that a phase is terminated unsuccessfully without running into a cycle is at most

$$
\begin{aligned}
&\Pr[\text{unsuccessful} \mid \text{no cycle}] \\
&\quad \leq \Pr[\exists \text{ active path-structure of size at least } \tfrac{2(\text{maxsteps}+1)-1}{3}] \\
&\quad \leq \Pr[\exists \text{ active path-structure of size at least } \ell + 1] \\
&\quad \leq 2\mu^2\Big(\frac{1}{1+\epsilon}\Big)^{\ell}
\end{aligned}
$$

Cuckoo Hashing

We choose $\text{maxsteps} \geq \frac{3\ell}{2} + 1$. Then the probability that a phase is terminated unsuccessfully without running into a cycle is at most

$$\begin{aligned}
&\Pr[\text{unsuccessful} \mid \text{no cycle}] \\
&\qquad \leq \Pr[\exists \text{ active path-structure of size at least } \tfrac{2(\text{maxsteps}+1)-1}{3}] \\
&\qquad \leq \Pr[\exists \text{ active path-structure of size at least } \ell + 1] \\
&\qquad \leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{\ell} \leq \frac{1}{m^2}
\end{aligned}$$

Cuckoo Hashing

We choose $\text{maxsteps} \geq \frac{3\ell}{2} + 1$. Then the probability that a phase is terminated unsuccessfully without running into a cycle is at most

$$\begin{aligned}
&\Pr[\text{unsuccessful} \mid \text{no cycle}] \\
&\qquad \leq \Pr[\exists \text{ active path-structure of size at least } \tfrac{2(\text{maxsteps}+1)-1}{3}] \\
&\qquad \leq \Pr[\exists \text{ active path-structure of size at least } \ell + 1] \\
&\qquad \leq 2\mu^2 \Big(\frac{1}{1+\epsilon}\Big)^{\ell} \leq \frac{1}{m^2}
\end{aligned}$$

by choosing $\ell \geq \log\left(\frac{1}{2\mu^2 m^2}\right) / \log\left(\frac{1}{1+\epsilon}\right) = \log\left(2\mu^2 m^2\right) / \log\left(1+\epsilon\right)$

Cuckoo Hashing

We choose $\text{maxsteps} \geq \frac{3\ell}{2} + 1$. Then the probability that a phase is terminated unsuccessfully without running into a cycle is at most

$$\begin{aligned}
&\Pr[\text{unsuccessful} \mid \text{no cycle}] \\
&\qquad \leq \Pr[\exists \text{ active path-structure of size at least } \tfrac{2(\text{maxsteps}+1)-1}{3}] \\
&\qquad \leq \Pr[\exists \text{ active path-structure of size at least } \ell + 1] \\
&\qquad \leq 2\mu^2\Big(\frac{1}{1+\epsilon}\Big)^{\ell} \leq \frac{1}{m^2}
\end{aligned}$$

by choosing $\ell \geq \log\big(\frac{1}{2\mu^2 m^2}\big)/\log\big(\frac{1}{1+\epsilon}\big) = \log\big(2\mu^2 m^2\big)/\log\big(1+\epsilon\big)$

Note that this gives $\text{maxsteps} = \Theta(\log m)$.

Cuckoo Hashing

The expected number of steps in the successful phase of an insert operation is:

$$\mathrm{E}[\text{number of steps} \mid \text{phase successful}]$$

We have

$$\begin{aligned}
&\Pr[\text{search at least } t \text{ steps} \mid \text{successful}] \\
&\quad = \Pr[\text{search at least } t \text{ steps} \wedge \text{successful}] / \Pr[\text{successful}] \\
&\quad \leq \frac{1}{c} \Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] \ ,
\end{aligned}$$

where we use the fact that for a suitable **constant** $c \geq 0$

$$\begin{aligned}
\Pr[\text{successful}] &= \Pr[\text{no cycle}] - \Pr[\text{unsuccessful} \mid \text{no cycle}] \\
&\geq c \cdot \Pr[\text{no cycle}]
\end{aligned}$$

Cuckoo Hashing

The expected number of steps in the successful phase of an insert operation is:

$$\mathrm{E}[\text{number of steps} \mid \text{phase successful}] = \sum_{t\geq 1} \Pr[\text{search takes at least } t \text{ steps} \mid \text{phase successful}]$$

We have

$$\begin{aligned}
&\Pr[\text{search at least } t \text{ steps} \mid \text{successful}] \\
&\quad = \Pr[\text{search at least } t \text{ steps} \land \text{successful}] / \Pr[\text{successful}] \\
&\quad \leq \frac{1}{c} \Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] \ ,
\end{aligned}$$

where we use the fact that for a suitable **constant** $c \geq 0$

$$\begin{aligned}
\Pr[\text{successful}] &= \Pr[\text{no cycle}] - \Pr[\text{unsuccessful} \mid \text{no cycle}] \\
&\geq c \cdot \Pr[\text{no cycle}]
\end{aligned}$$

Hence,

$$
\begin{aligned}
\mathrm{E}[\text{number of steps} \mid \text{phase successful}]
&= \frac{1}{c}\sum_{t\geq 1}\Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] \\
&\leq \frac{1}{c}\Big[1 + \sum_{t\geq 2} 2\mu^2\Big(\frac{1}{1+\epsilon}\Big)^{2(t-2)/3}\Big] \\
&= \frac{1}{c} + \frac{2\mu^2}{c}\sum_{t\geq 0}\Big(\frac{1}{(1+\epsilon)^{2/3}}\Big)^{t} = \mathcal{O}(1) \enspace .
\end{aligned}
$$

Hence,

$$\begin{aligned}
&\mathrm{E}[\text{number of steps} \mid \text{phase successful}] \\
&\quad = \frac{1}{c} \sum_{t \geq 1} \Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] \\
&\quad \leq \frac{1}{c} \left[1 + \sum_{t \geq 2} 2\mu^2 \left(\frac{1}{1+\epsilon} \right)^{2(t-2)/3} \right] \\
&\quad = \frac{1}{c} + \frac{2\mu^2}{c} \sum_{t \geq 0} \left(\frac{1}{(1+\epsilon)^{2/3}} \right)^t = \mathcal{O}(1) \ .
\end{aligned}$$

Hence,

$$
\begin{aligned}
&\mathrm{E}[\text{number of steps} \mid \text{phase successful}] \\
&\quad = \frac{1}{c}\sum_{t\geq 1}\Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] \\
&\quad \leq \frac{1}{c}\left[1 + \sum_{t\geq 2} 2\mu^2\Big(\frac{1}{1+\epsilon}\Big)^{2(t-2)/3}\right] \\
&\quad = \frac{1}{c} + \frac{2\mu^2}{c}\sum_{t\geq 0}\Big(\frac{1}{(1+\epsilon)^{2/3}}\Big)^{t} = \mathcal{O}(1)\ .
\end{aligned}
$$

Hence,

$$
\begin{aligned}
&\mathrm{E}[\text{number of steps} \mid \text{phase successful}] \\
&\quad = \frac{1}{c}\sum_{t\geq 1}\Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] \\
&\quad \leq \frac{1}{c}\left[1 + \sum_{t\geq 2} 2\mu^2\Big(\frac{1}{1+\epsilon}\Big)^{2(t-2)/3}\right] \\
&\quad = \frac{1}{c} + \frac{2\mu^2}{c}\sum_{t\geq 0}\Big(\frac{1}{(1+\epsilon)^{2/3}}\Big)^{t} = \mathcal{O}(1)\enspace.
\end{aligned}
$$

Hence,

$$\begin{aligned}
&\mathrm{E}[\text{number of steps} \mid \text{phase successful}] \\
&\quad = \frac{1}{c} \sum_{t\geq 1} \Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}] \\
&\quad \leq \frac{1}{c} \left[1 + \sum_{t\geq 2} 2\mu^2 \Big(\frac{1}{1+\epsilon}\Big)^{2(t-2)/3} \right] \\
&\quad = \frac{1}{c} + \frac{2\mu^2}{c} \sum_{t\geq 0} \Big(\frac{1}{(1+\epsilon)^{2/3}}\Big)^t = \mathcal{O}(1) \ .
\end{aligned}$$

Cuckoo Hashing

A phase that is not successful induces cost $\mathcal{O}(m)$ for doing a complete rehash (this dominates the cost for the steps in the phase).

The probability that a phase is not successful is $p = \mathcal{O}(1/m^2)$ (probability $\mathcal{O}(1/m^2)$ of running into a cycle and probability $\mathcal{O}(1/m^2)$ of reaching $\mathrm{maxsteps}$ without running into a cycle).

The expected number of unsuccessful phases is $\sum_{i\geq 1} p^i = \frac{1}{1-p} - 1 = \frac{p}{1-p} = \mathcal{O}(p)$.

Therefore the expected cost for re-hashes is $\mathcal{O}(m) \cdot \mathcal{O}(p) = \mathcal{O}(1/m)$.

Cuckoo Hashing

A phase that is not successful induces cost $\mathcal{O}(m)$ for doing a complete rehash (this dominates the cost for the steps in the phase).

The probability that a phase is not successful is $p = \mathcal{O}(1/m^2)$ (probability $\mathcal{O}(1/m^2)$ of running into a cycle and probability $\mathcal{O}(1/m^2)$ of reaching maxsteps without running into a cycle).

The expected number of unsuccessful phases is $\sum_{i\geq 1} p^i = \frac{1}{1-p} - 1 = \frac{p}{1-p} = \mathcal{O}(p)$.

Therefore the expected cost for re-hashes is $\mathcal{O}(m) \cdot \mathcal{O}(p) = \mathcal{O}(1/m)$.

Cuckoo Hashing

A phase that is not successful induces cost $\mathcal{O}(m)$ for doing a complete rehash (this dominates the cost for the steps in the phase).

The probability that a phase is not successful is $p = \mathcal{O}(1/m^2)$ (probability $\mathcal{O}(1/m^2)$ of running into a cycle and probability $\mathcal{O}(1/m^2)$ of reaching $\mathrm{maxsteps}$ without running into a cycle).

The expected number of unsuccessful phases is $\sum_{i\geq 1} p^i = \frac{1}{1-p} - 1 = \frac{p}{1-p} = \mathcal{O}(p)$.

Therefore the expected cost for re-hashes is $\mathcal{O}(m) \cdot \mathcal{O}(p) = \mathcal{O}(1/m)$.

Cuckoo Hashing

A phase that is not successful induces cost $\mathcal{O}(m)$ for doing a complete rehash (this dominates the cost for the steps in the phase).

The probability that a phase is not successful is $p = \mathcal{O}(1/m^2)$ (probability $\mathcal{O}(1/m^2)$ of running into a cycle and probability $\mathcal{O}(1/m^2)$ of reaching maxsteps without running into a cycle).

The expected number of unsuccessful phases is
$\sum_{i\geq 1} p^i = \frac{1}{1-p} - 1 = \frac{p}{1-p} = \mathcal{O}(p)$.

Therefore the expected cost for re-hashes is
$\mathcal{O}(m) \cdot \mathcal{O}(p) = \mathcal{O}(1/m)$.

Cuckoo Hashing

A phase that is not successful induces cost $\mathcal{O}(m)$ for doing a complete rehash (this dominates the cost for the steps in the phase).

The probability that a phase is not successful is $p = \mathcal{O}(1/m^2)$ (probability $\mathcal{O}(1/m^2)$ of running into a cycle and probability $\mathcal{O}(1/m^2)$ of reaching $\mathrm{maxsteps}$ without running into a cycle).

The expected number of unsuccessful phases is
$\sum_{i\geq 1} p^i = \frac{1}{1-p} - 1 = \frac{p}{1-p} = \mathcal{O}(p)$.

Therefore the expected cost for re-hashes is
$\mathcal{O}(m) \cdot \mathcal{O}(p) = \mathcal{O}(1/m)$.

Cuckoo Hashing

What kind of hash-functions do we need?

Since maxsteps is $\Theta(\log m)$ the largest size of a path-structure or cycle-structure contains just $\Theta(\log m)$ different keys.

Therefore, it is sufficient to have $(\mu, \Theta(\log m))$-independent hash-functions.

Cuckoo Hashing

What kind of hash-functions do we need?

Since maxsteps is $\Theta(\log m)$ the largest size of a path-structure or cycle-structure contains just $\Theta(\log m)$ different keys.

Therefore, it is sufficient to have $(\mu, \Theta(\log m))$-independent hash-functions.

Cuckoo Hashing

What kind of hash-functions do we need?

Since maxsteps is $\Theta(\log m)$ the largest size of a path-structure or cycle-structure contains just $\Theta(\log m)$ different keys.

Therefore, it is sufficient to have $(\mu, \Theta(\log m))$-independent hash-functions.

Cuckoo Hashing

How do we make sure that $n \geq (1+\epsilon)m$?

- Let $\alpha := 1/(1+\epsilon)$.
- Keep track of the number of elements in the table. When $m \geq \alpha n$ we double n and do a complete re-hash (table-expand).
- Whenever m drops below $\alpha n/4$ we divide n by 2 and do a rehash (table-shrink).
- Note that right after a change in table-size we have $m = \alpha n/2$. In order for a table-expand to occur at least $\alpha n/2$ insertions are required. Similar, for a table-shrink at least $\alpha n/4$ deletions must occur.
- Therefore we can amortize the rehash cost after a change in table-size against the cost for insertions and deletions.

Cuckoo Hashing

How do we make sure that $n \geq (1+\epsilon)m$?

- Let $\alpha := 1/(1+\epsilon)$.
- Keep track of the number of elements in the table. When $m \geq \alpha n$ we double n and do a complete re-hash (table-expand).
- Whenever m drops below $\alpha n/4$ we divide n by 2 and do a rehash (table-shrink).
- Note that right after a change in table-size we have $m = \alpha n/2$. In order for a table-expand to occur at least $\alpha n/2$ insertions are required. Similar, for a table-shrink at least $\alpha n/4$ deletions must occur.
- Therefore we can amortize the rehash cost after a change in table-size against the cost for insertions and deletions.

Cuckoo Hashing

How do we make sure that $n \geq (1+\epsilon)m$?

- Let $\alpha := 1/(1+\epsilon)$.
- Keep track of the number of elements in the table. When $m \geq \alpha n$ we double n and do a complete re-hash (table-expand).
- Whenever m drops below $\alpha n/4$ we divide n by 2 and do a rehash (table-shrink).
- Note that right after a change in table-size we have $m = \alpha n/2$. In order for a table-expand to occur at least $\alpha n/2$ insertions are required. Similar, for a table-shrink at least $\alpha n/4$ deletions must occur.
- Therefore we can amortize the rehash cost after a change in table-size against the cost for insertions and deletions.

Cuckoo Hashing

How do we make sure that $n \geq (1+\epsilon)m$?

- Let $\alpha := 1/(1+\epsilon)$.
- Keep track of the number of elements in the table. When $m \geq \alpha n$ we double n and do a complete re-hash (table-expand).
- Whenever m drops below $\alpha n/4$ we divide n by 2 and do a rehash (table-shrink).
- Note that right after a change in table-size we have $m = \alpha n/2$. In order for a table-expand to occur at least $\alpha n/2$ insertions are required. Similar, for a table-shrink at least $\alpha n/4$ deletions must occur.
- Therefore we can amortize the rehash cost after a change in table-size against the cost for insertions and deletions.

Cuckoo Hashing

How do we make sure that $n \geq (1+\epsilon)m$?

- Let $\alpha := 1/(1+\epsilon)$.
- Keep track of the number of elements in the table. When $m \geq \alpha n$ we double n and do a complete re-hash (table-expand).
- Whenever m drops below $\alpha n/4$ we divide n by 2 and do a rehash (table-shrink).
- Note that right after a change in table-size we have $m = \alpha n/2$. In order for a table-expand to occur at least $\alpha n/2$ insertions are required. Similar, for a table-shrink at least $\alpha n/4$ deletions must occur.
- Therefore we can amortize the rehash cost after a change in table-size against the cost for insertions and deletions.

Cuckoo Hashing

Lemma 24

Cuckoo Hashing has an expected constant insert-time and a worst-case constant search-time.

Note that the above lemma only holds if the fill-factor (number of keys/total number of hash-table slots) is at most $\frac{1}{2(1+\epsilon)}$.

Cuckoo Hashing

Lemma 24
Cuckoo Hashing has an expected constant insert-time and a worst-case constant search-time.

Note that the above lemma only holds if the fill-factor (number of keys/total number of hash-table slots) is at most $\frac{1}{2(1+\epsilon)}$.