Part III

Data Structures

Abstract Data Type

An abstract data type (ADT) is defined by an interface of operations or methods that can be performed and that have a defined behavior.

The data types in this lecture all operate on objects that are represented by a [key, value] pair.

- The key comes from a totally ordered set, and we assume that there is an efficient comparison function.
- The value can be anything; it usually carries satellite information important for the application that uses the ADT.

Dynamic Set Operations

- $\mathbf{S.\,search(k)}$: Returns pointer to object x from S with $\text{key}[x] = k$ or null.
- $\mathbf{S.\,insert(x)}$: Inserts object x into set S. $\text{key}[x]$ must not currently exist in the data-structure.
- $\mathbf{S.\,delete(x)}$: Given pointer to object x from S, delete x from the set.
- $\mathbf{S.\,minimum()}$: Return pointer to object with smallest key-value in S.
- $\mathbf{S.\,maximum()}$: Return pointer to object with largest key-value in S.
- $\mathbf{S.\,successor(x)}$: Return pointer to the next larger element in S or null if S is maximum.
- $\mathbf{S.\,predecessor(x)}$: Return pointer to the next smaller element in S or null if S is minimum.

Dynamic Set Operations

- $\mathbf{S.\,union(S')}$: Sets $S := S \cup S'$. The set S' is destroyed.
- $\mathbf{S.\,merge(S')}$: Sets $S := S \cup S'$. Requires $S \cap S' = \emptyset$.
- $\mathbf{S.\,split(k, S')}$: $S := \{x \in S \mid \text{key}[x] \leq k\}$, $S' := \{x \in S \mid \text{key}[x] > k\}$.
- $\mathbf{S.\,concatenate(S')}$: $S := S \cup S'$. Requires $S.\text{maximum}() \leq S'.\text{minimum}()$.
- $\mathbf{S.\,decrease\text{-}key(x, k)}$: Replace $\text{key}[x]$ by $k \leq \text{key}[x]$.

Examples of ADTs

Stack:

- S.**push**(x): Insert an element.
- S.**pop**$()$: Return the element from S that was inserted most recently; delete it from S.
- S.**empty**$()$: Tell if S contains any object.

Queue:

- S.**enqueue**(x): Insert an element.
- S.**dequeue**$()$: Return the element that is longest in the structure; delete it from S.
- S.**empty**$()$: Tell if S contains any object.

Priority-Queue:

- S.**insert**(x): Insert an element.
- S.**delete-min**$()$: Return the element with lowest key-value; delete it from S.

7 Dictionary

Dictionary:

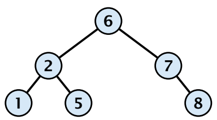
- S.**insert**(x): Insert an element x.
- S.**delete**(x): Delete the element pointed to by x.
- S.**search**(k): Return a pointer to an element e with $\text{key}[e] = k$ in S if it exists; otherwise return null.

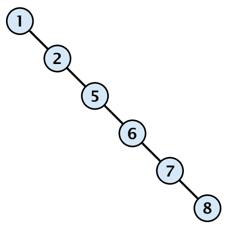
7.1 Binary Search Trees

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 $\text{key}[v]$ 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:



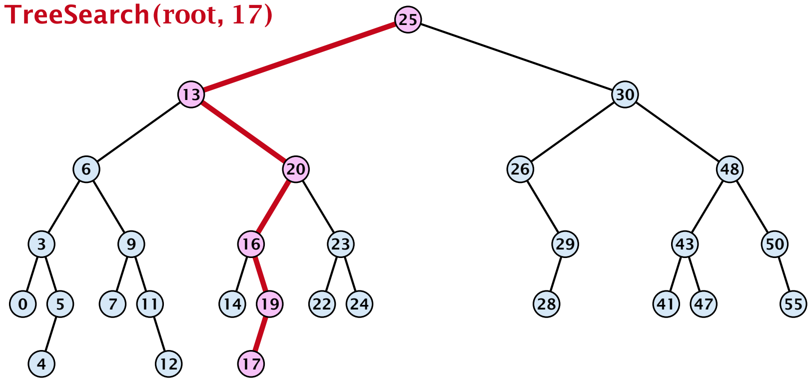


7.1 Binary Search Trees

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

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

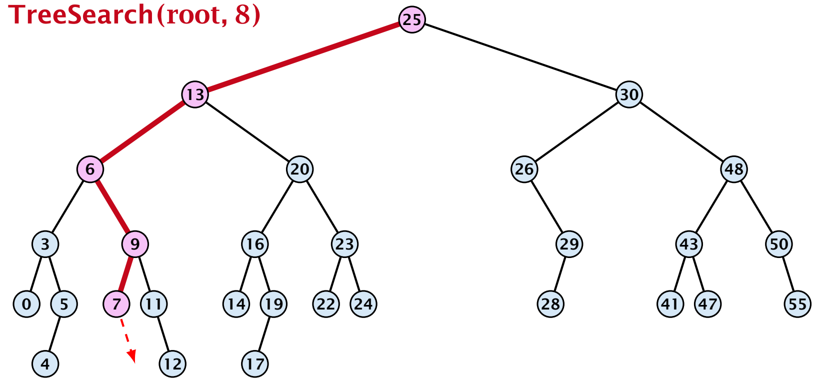
Binary Search Trees: Searching



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

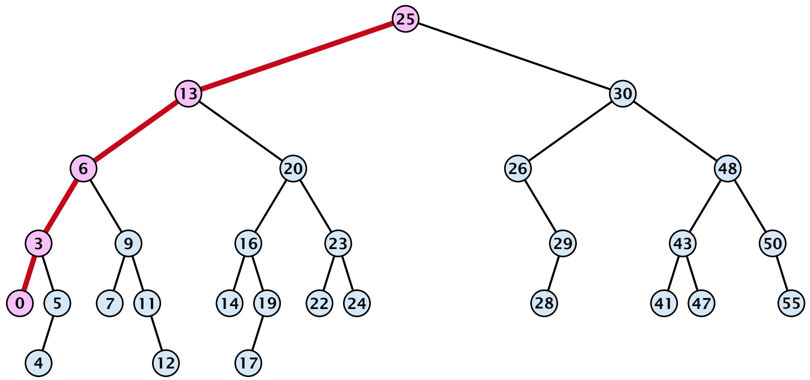
Binary Search Trees: Searching



Algorithm 5 TreeSearch(x, k)

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

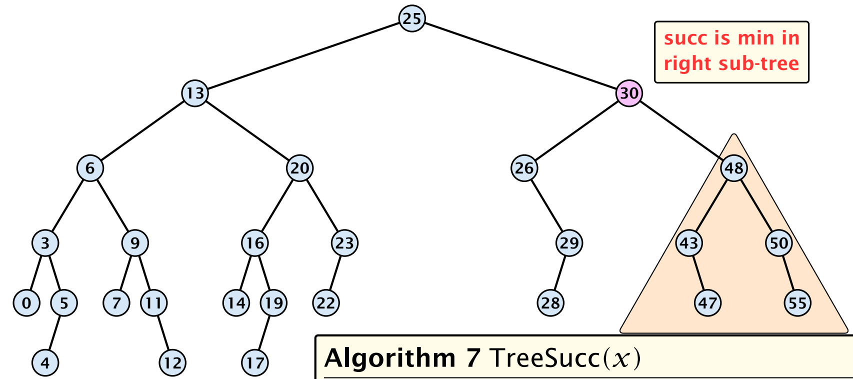
Binary Search Trees: Minimum



Algorithm 6 TreeMin(x)

if $x = \text{null}$ **or** $\text{left}[x] = \text{null}$ **return** x
return TreeMin$(\text{left}[x])$

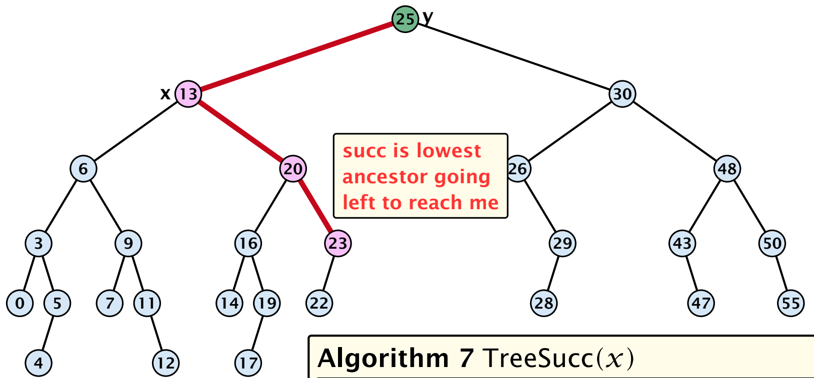
Binary Search Trees: Successor



Algorithm 7 TreeSucc(x)

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

Binary Search Trees: Successor



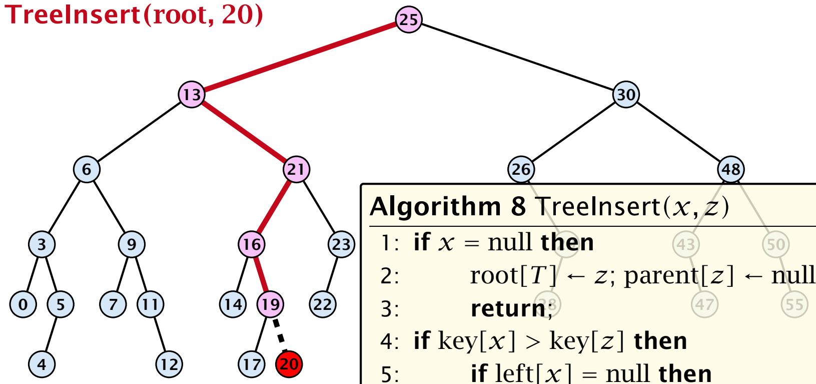
Algorithm 7 TreeSucc(x)

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: Insert

Insert element **not** in the tree.

TreeInsert(root, 20)

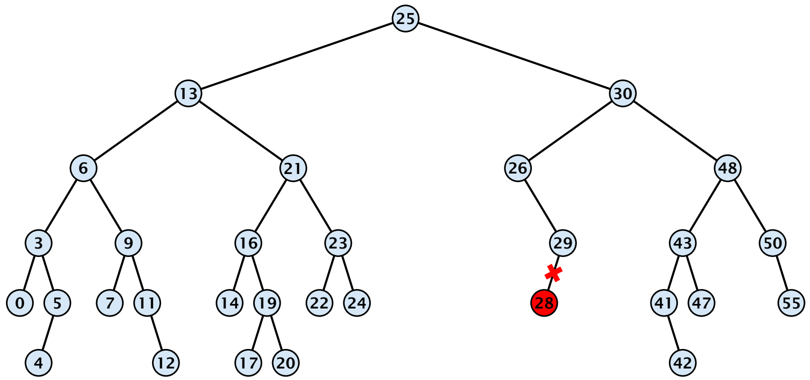


Search for z. At some point the search stops at a null-pointer. This is the place to insert z.

Algorithm 8 TreeInsert(x, z)

1: **if** $x =$ null **then**
2: $\quad$ root[T] $\leftarrow z$; parent[z] $\leftarrow$ null;
3: $\quad$ **return**;
4: **if** key[x] $>$ key[z] **then**
5: $\quad$ **if** left[x] $=$ null **then**
6: $\quad\quad$ left[x] $\leftarrow z$; parent[z] $\leftarrow x$;
7: $\quad$ **else** TreeInsert(left[x], z);
8: **else**
9: $\quad$ **if** right[x] $=$ null **then**
10: $\quad\quad$ right[x] $\leftarrow z$; parent[z] $\leftarrow x$;
11: $\quad$ **else** TreeInsert(right[x], z);

Binary Search Trees: Delete

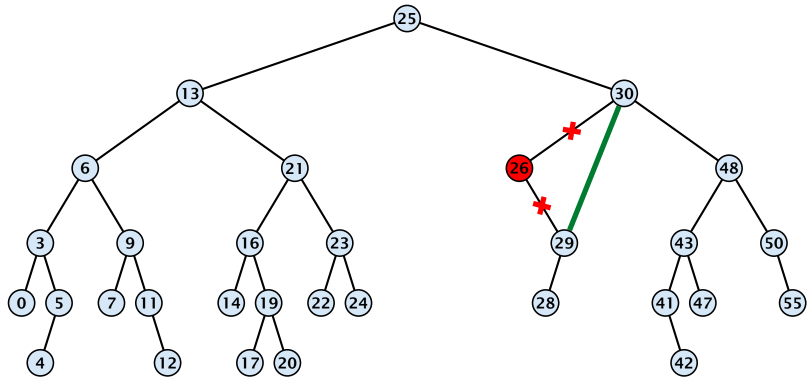


Case 1:

Element does not have any children

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

Binary Search Trees: Delete

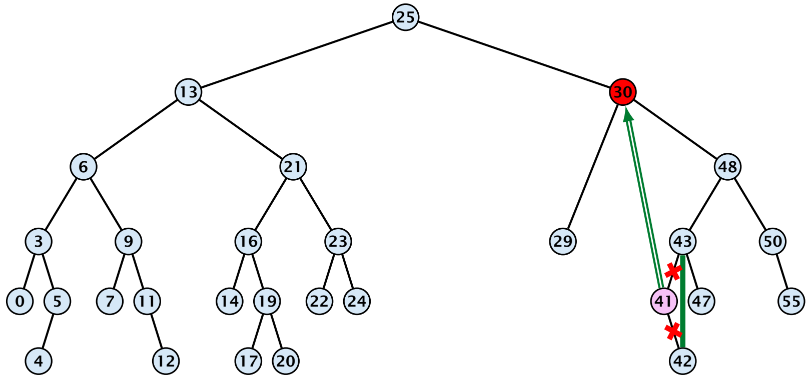


Case 2:

Element has exactly one child

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

Binary Search Trees: Delete



Case 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 ← z else y ← TreeSucc(z);          select y to splice out
if left[y] ≠ null
    then x ← left[y] else x ← right[y];       x is child of y (or null)
if x ≠ null then parent[x] ← parent[y];       parent[x] is correct
if parent[y] = null then
    root[T] ← x
else
    if y = left[parent[x]] then                 } fix pointer to x
        left[parent[y]] ← x
    else
        right[parent[y]] ← x
if y ≠ z then copy y-data to z
```


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.

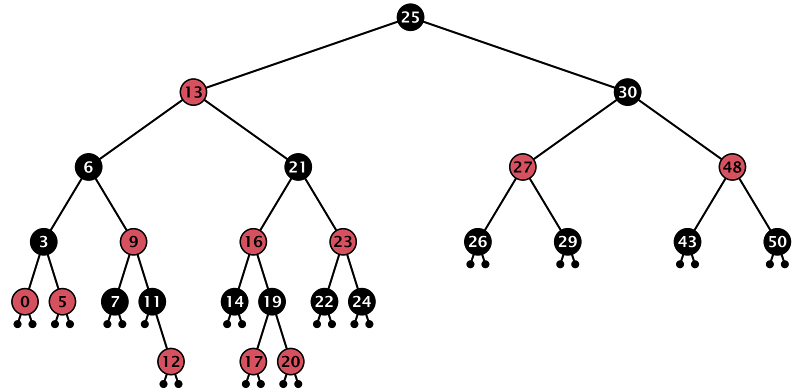
7.2 Red Black Trees

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

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

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 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

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.

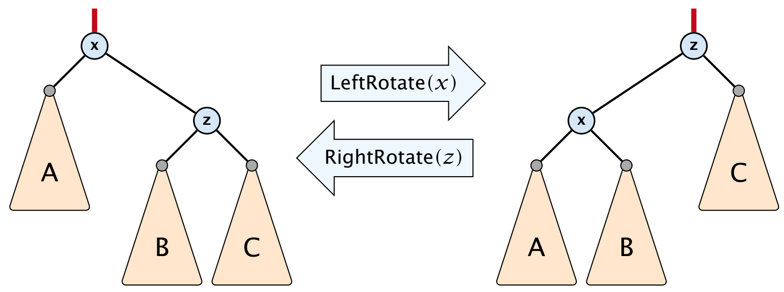
The null-pointers in a binary search tree are replaced by pointers to special null-vertices, that do not carry any object-data.

7.2 Red Black Trees

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

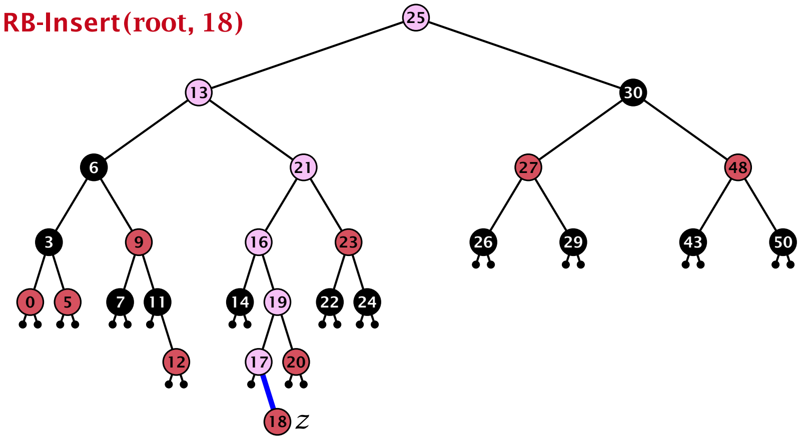
Rotations

The properties will be maintained through rotations:



Red Black Trees: Insert

RB-Insert(root, 18)



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 parent$[z]$
  - either both of them are red (most important case)
  - or the parent does not exist (violation since root must be black)

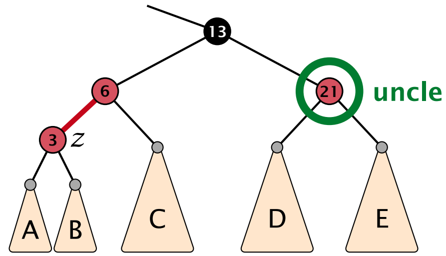
If z has a parent but no grand-parent we could simply color the parent/root black; however this case never happens.

Red Black Trees: Insert

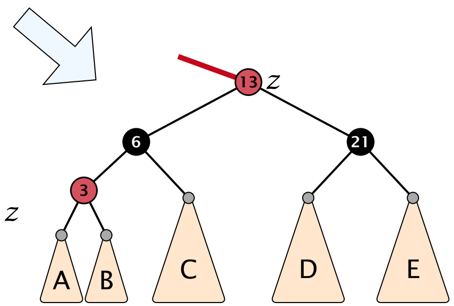
Algorithm 10 InsertFix(z)

```
while parent[z] ≠ null and col[parent[z]] = red do
    if parent[z] = left[gp[z]] then                z in left subtree of grandparent
        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                                       Case 2: uncle black
            if z = right[parent[z]] then           2a: z right child
                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

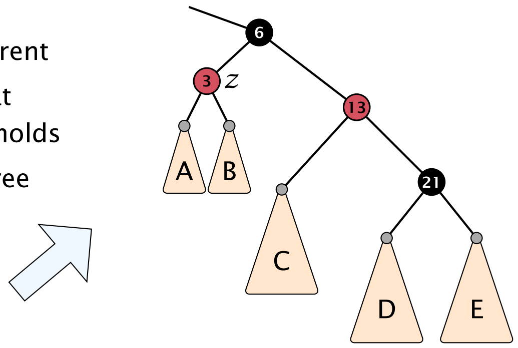


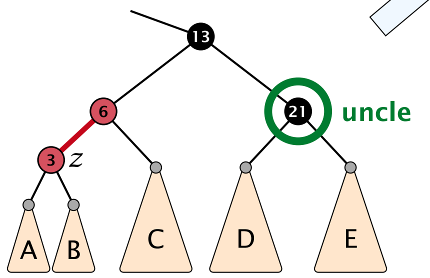
1. recolour
2. move z to grand-parent
3. invariant is fulfilled for new z
4. you made progress



Case 2b: Black uncle and z is left child

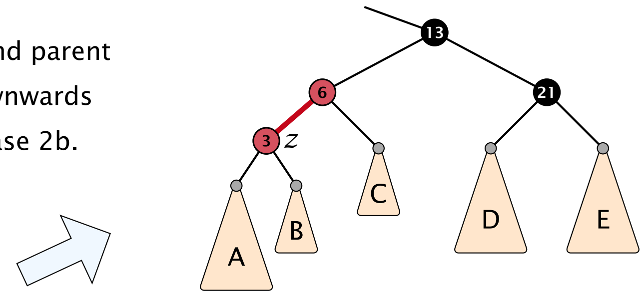
1. rotate around grandparent
2. re-colour to ensure that black height property holds
3. you have a red black tree

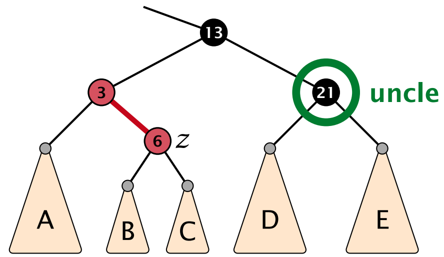




Case 2a: Black uncle and z is right child

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: Delete

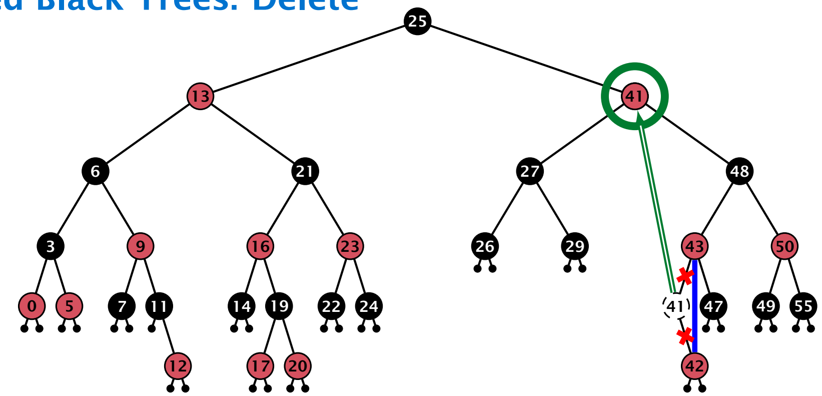
First do a standard delete.

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

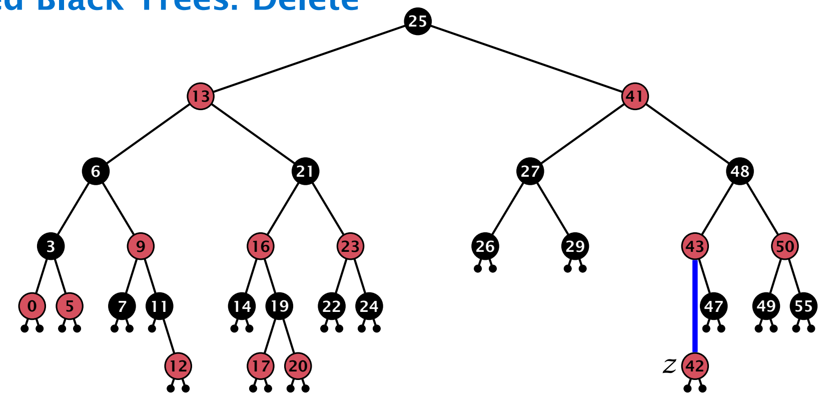


Case 3:

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.

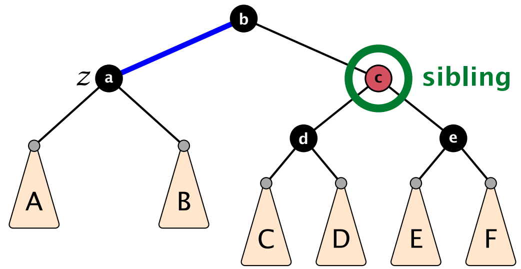
Red Black Trees: Delete

Invariant of the fix-up algorithm

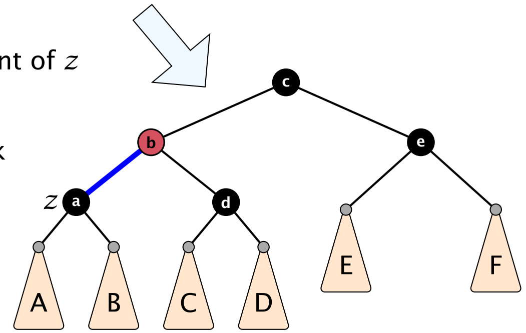
- the node z is black
- if we "assign" a fake black unit to the edge from z to its parent then the black-height property is fulfilled

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

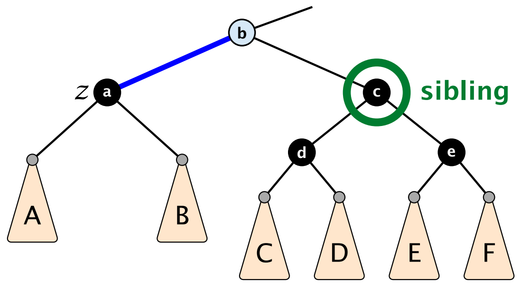
Case 1: Sibling of z is red



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

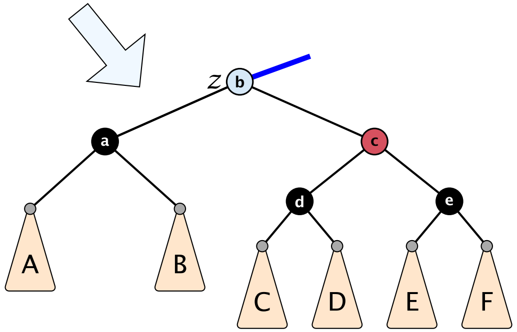


Case 2: Sibling is black with two black children



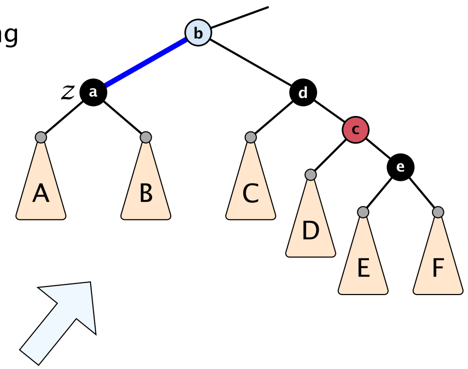
Here b is either black or red. If it is red we are in a special case that directly leads to a red-black tree.

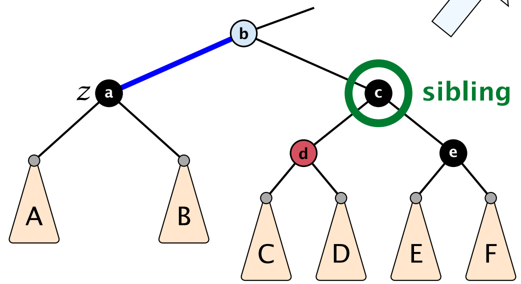
1. re-color node c
2. move fake black unit upwards
3. move z upwards
4. we made progress
5. if b is red we color it black and are done



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

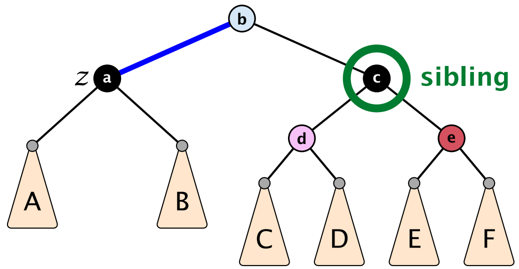
1. do a right-rotation at sibling
2. recolor c and d
3. new sibling is black with red right child (Case 4)





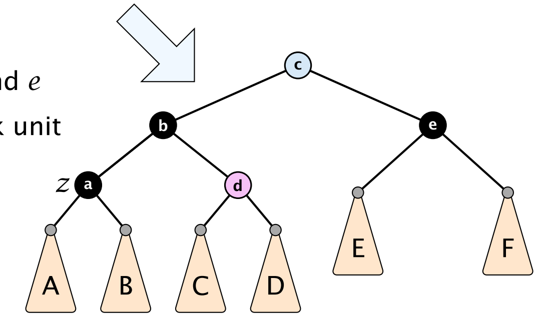
Again the blue color of b indicates that it can either be black or red.

Case 4: Sibling is black with red right child



- Here b and d are either red or black but have possibly different colors.
- We recolor c by giving it the color of b.

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.

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

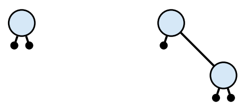
$$\sum_{j=0}^{h-1} 2^j = 2^h - 1$$

internal nodes.

Proof (cont.)

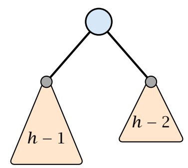
Induction (base cases):

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, $2 \geq F_4 - 1 = 3 - 1 = 2$



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

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

we get

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

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

7.3 AVL-Trees

We need to maintain the balance condition through rotations.

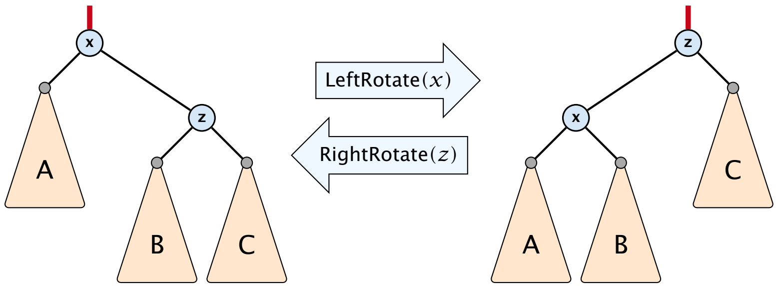
For this we store in every internal tree-node v the balance of the node. Let v denote a tree node with left child c_ℓ and right child c_r.

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

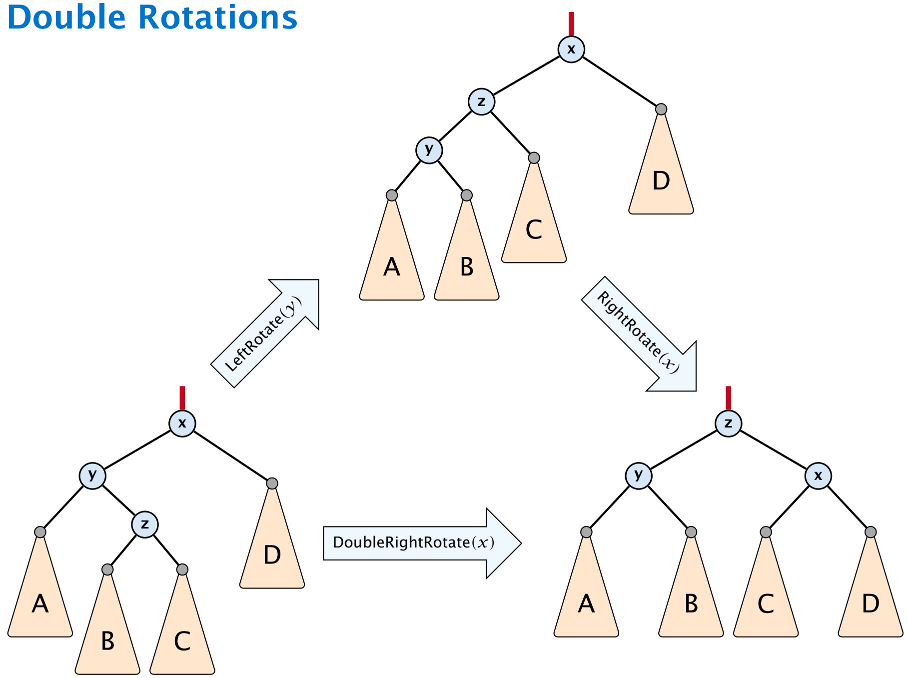
where T_{c_ℓ} and T_{c_r}, are the sub-trees rooted at c_ℓ and c_r, respectively.

Rotations

The properties will be maintained through rotations:



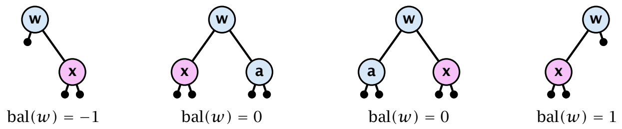
Double Rotations



AVL-trees: Insert

Note that before the insertion w is right above the leaf level, i.e., x replaces a child of w that was a dummy leaf.

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

Note that these constraints hold for the first call AVL-fix-up-insert$(\text{parent}[w])$.

AVL-trees: Insert

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

```
if balance[v] ∈ {−2, 2} then DoRotationInsert(v);
if balance[v] ∈ {0} return;
AVL-fix-up-insert(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.

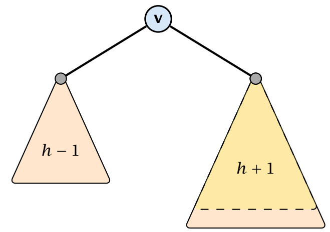
We show that after doing a rotation at v:

- 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

We have the following situation:

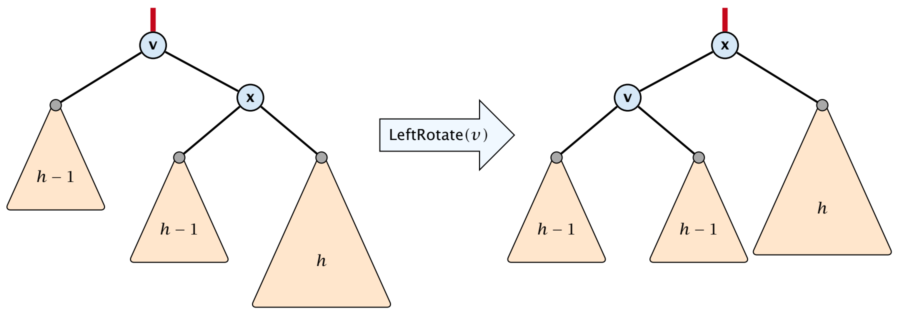


The right sub-tree of v has increased its height which results in a balance of -2 at v.

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

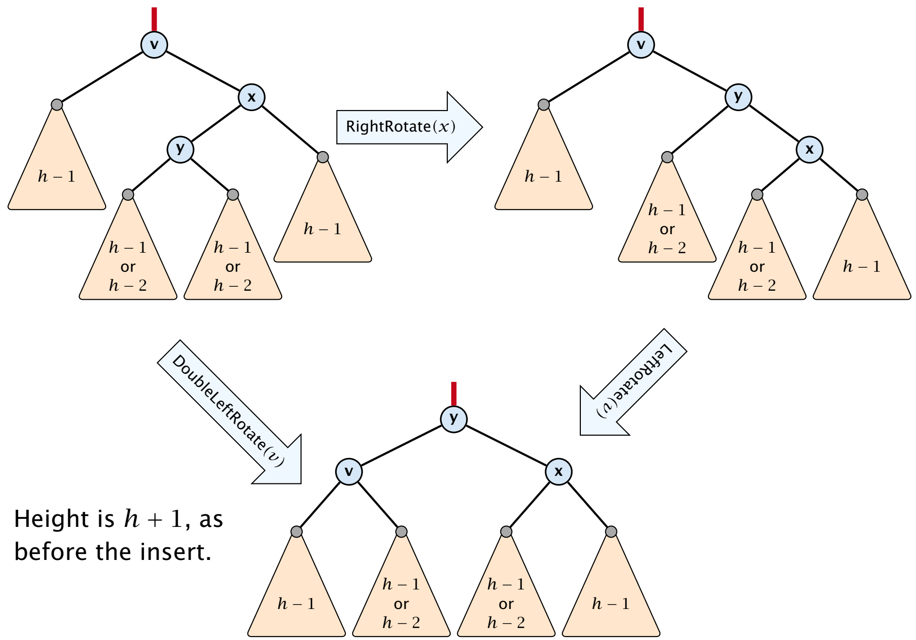
Case 1: balance[right[v]] = -1

We do a left rotation at v



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

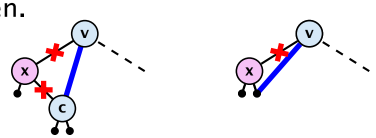
Case 2: balance[right[v]] = 1



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

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

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

if $\text{balance}[v] \in \{-2, 2\}$ **then** DoRotationDelete(v);
if $\text{balance}[v] \in \{-1, 1\}$ **return**;
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 $\text{balance}[v] = -2$ **then** // deletion in left sub-tree
  if $\text{balance}[\text{right}[v]] \in \{0, -1\}$ **then**
    LeftRotate(v);
  else
    DoubleLeftRotate(v);
else // deletion in right sub-tree
  if $\text{balance}[\text{left}[v]] = \{0, 1\}$ **then**
    RightRotate(v);
  else
    DoubleRightRotate(v);

Note that the case distinction on the second level ($\text{bal}[\text{right}[v]]$ and $\text{bal}[\text{left}[v]]$) is not done w.r.t. the child c for which the sub-tree T_c has changed. This is different to AVL-fix-up-insert.

AVL-trees: Delete

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

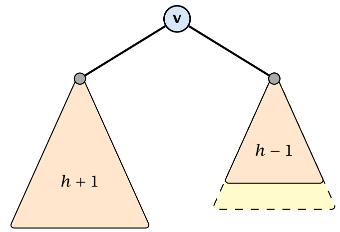
We show that after doing a rotation at v:

- 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

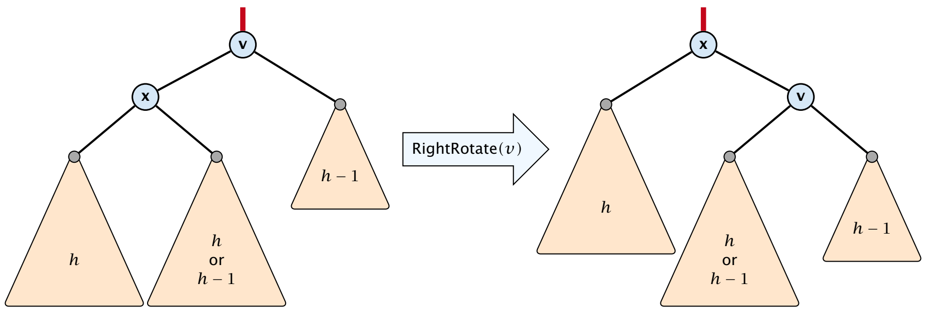
We have the following situation:



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 $h + 2$.

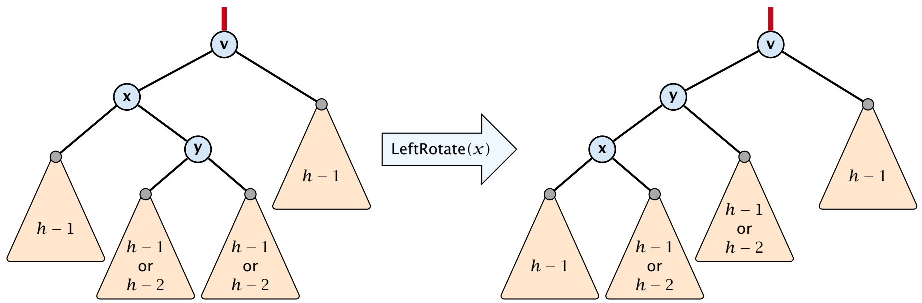
Case 1: $\text{balance}[\text{left}[v]] \in \{0, 1\}$

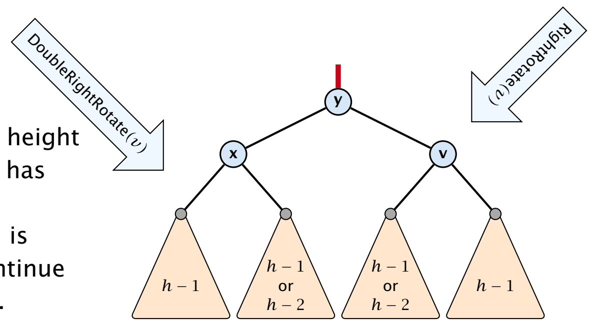


If the middle subtree has height h the whole tree has height $h + 2$ as before the deletion. The iteration stops as the balance at the root is non-zero.

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: $\text{balance}[\text{left}[v]] = -1$





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

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

- Of course, the above steps heavily depend on each other. For example it makes no sense to choose additional information to be stored (Step 2), and later realize that either the information cannot be maintained efficiently (Step 3) or is not sufficient to support the new operations (Step 4).
- However, the above outline is a good way to describe/document a new data-structure.

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

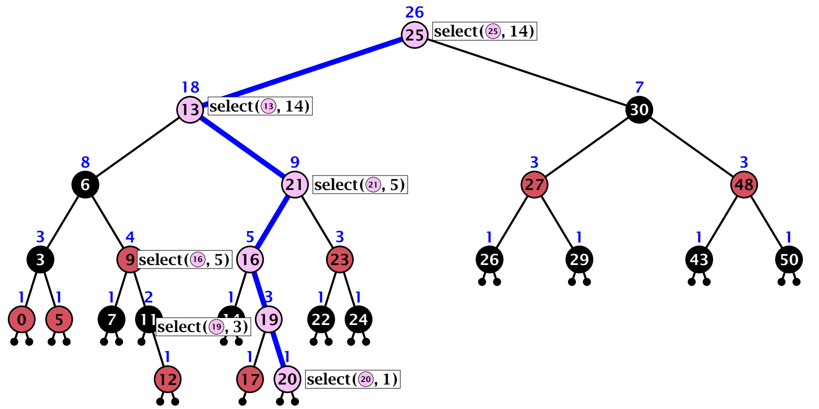
Goal: Design a data-structure that supports insert, delete, search, and find-by-rank in time $\mathcal{O}(\log n)$.

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

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?

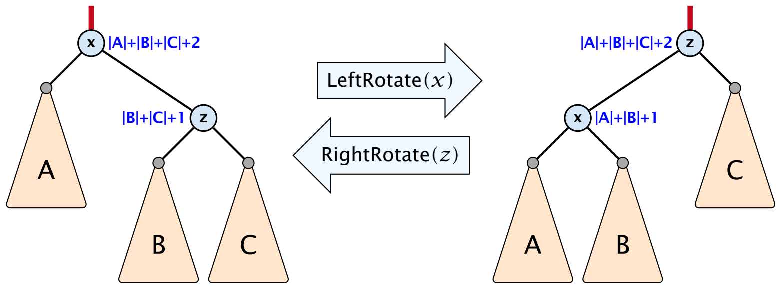
Search(k): Nothing to do.

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

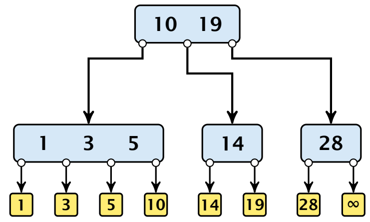
Each internal node v with $d(v)$ children stores $d - 1$ keys $k_1, \dots, k_d - 1$. The i-th subtree of v fulfills

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

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).

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

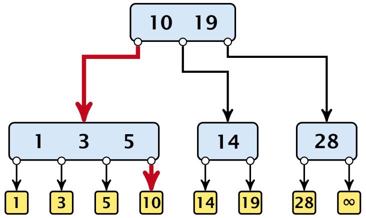
Proof.

- 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.

□

Search

Search(8)

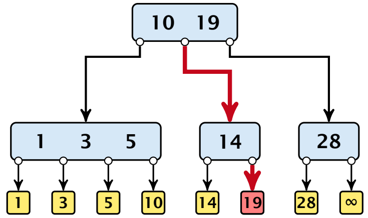


The search is straightforward. It is only important that you need to go all the way to the leaf.

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

Search

Search(19)



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).

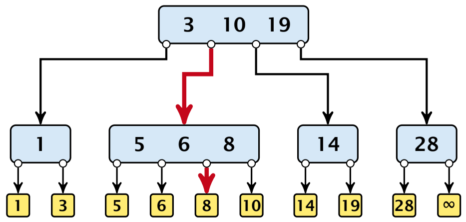
Insert

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.

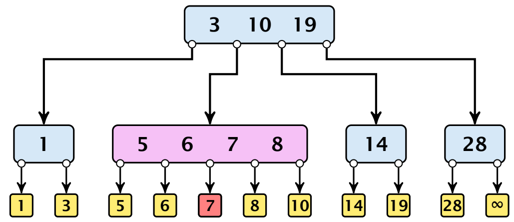
Insert

Insert(7)



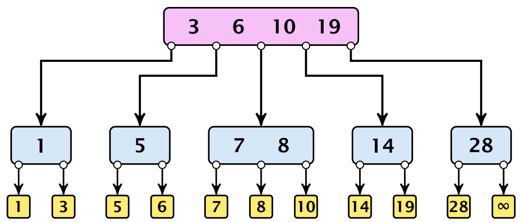
Insert

Insert(7)



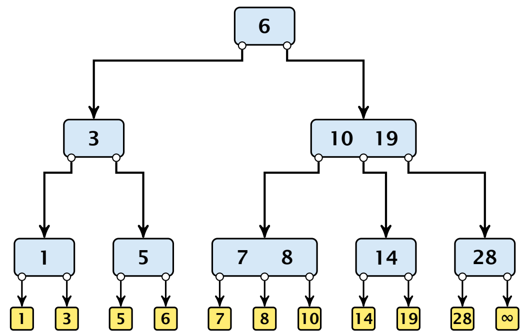
Insert

Insert(7)



Insert

Insert(7)



Delete

Delete element x (pointer to leaf vertex):

- Let v denote the parent of x. If $\text{key}[x]$ 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):

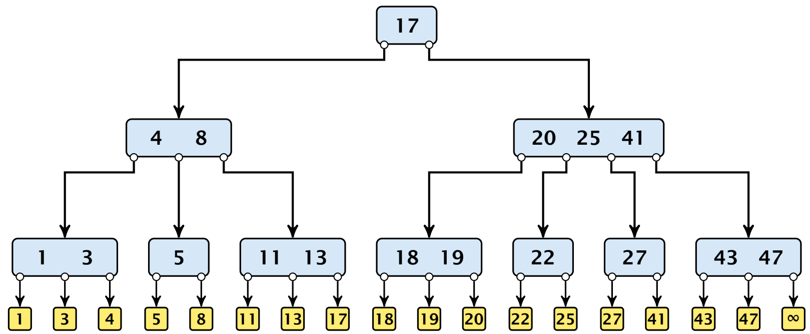
- If there is a neighbour of v that has at least a keys take over the largest (if right neighbor) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge v with one of its neighbours.
- The merged node contains at most $(a-2)+(a-1)+1$ keys, and has therefore at most $2a-1 \le b$ 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

Animation for deleting in an (a,b)-tree is only available in the lecture version of the slides.

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

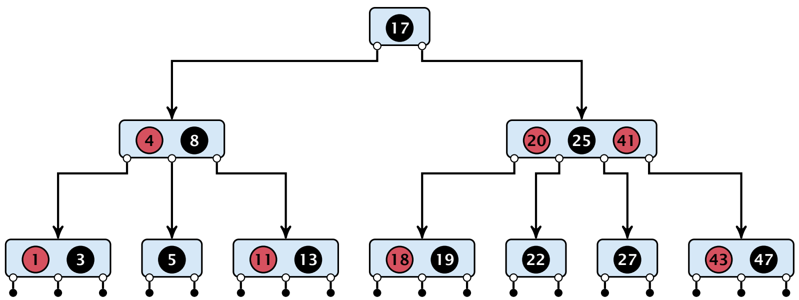
There is a close relation between red-black trees and $(2,4)$-trees:



First make it into an internal search tree by moving the satellite-data from the leaves to internal nodes. Add dummy leaves.

(2, 4)-trees and red black trees

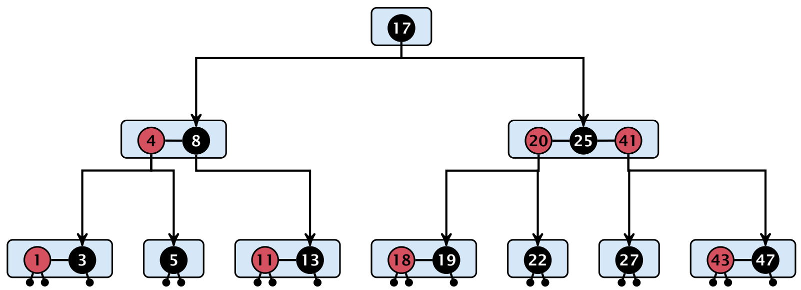
There is a close relation between red-black trees and $(2,4)$-trees:



Then, color one key in each internal node v black. If v contains 3 keys you need to select the middle key otherwise choose a black key arbitrarily. The other keys are colored red.

(2, 4)-trees and red black trees

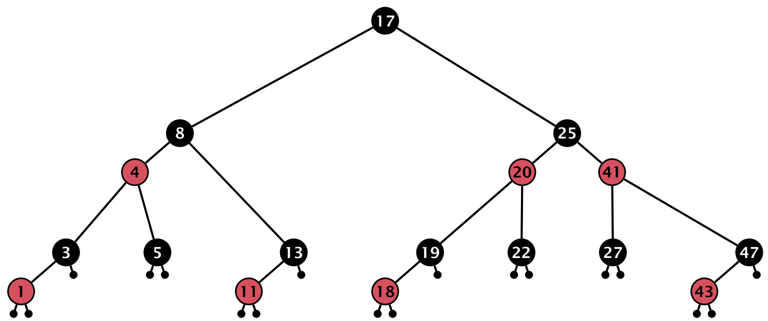
There is a close relation between red-black trees and $(2,4)$-trees:



Re-attach the pointers to individual keys. A pointer that is between two keys is attached as a child of the red key. The incoming pointer, points to the black key.

(2, 4)-trees and red black trees

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

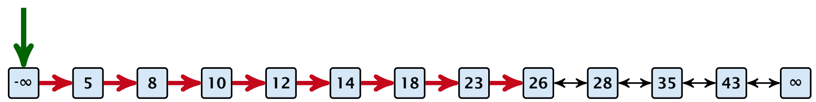


Note that this correspondence is not unique. In particular, there are different red-black trees that correspond to the same $(2,4)$-tree.

7.6 Skip Lists

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

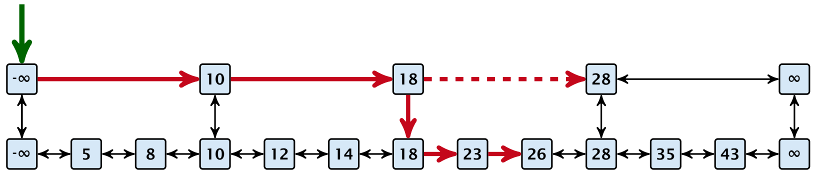
- time for search $\Theta(n)$
- time for insert $\Theta(n)$ (dominated by searching the item)
- time for delete $\Theta(1)$ if we are given a handle to the object, otw. $\Theta(n)$



7.6 Skip Lists

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}.

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.
- ...
- 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, $L_k \approx r^{-k}n$.

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

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

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

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 $L_0, \dots, L_{t-1}$.

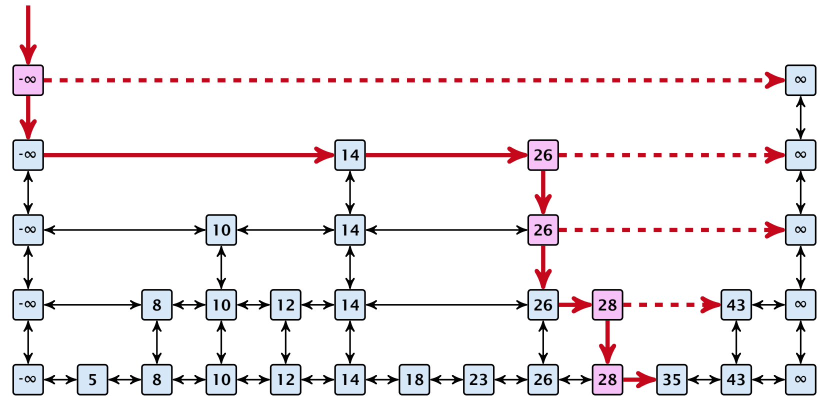
Delete:

- 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):



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 $E_1, E_2, \dots, E_\ell$, $\ell = n^c$ 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

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

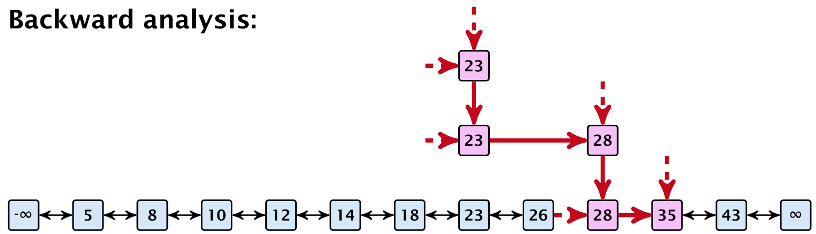
This means $\Pr[E_1 \wedge \cdots \wedge E_\ell]$ holds with high probability.

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:



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.

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

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

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

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

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

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

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

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

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,

$$\Pr[\text{search requires } z \text{ steps}] \leq \Pr[E_{z,k}] + \Pr[A_{k+1}] \leq n^{-\alpha} + n^{-(\gamma-1)}$$

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

Definitions:

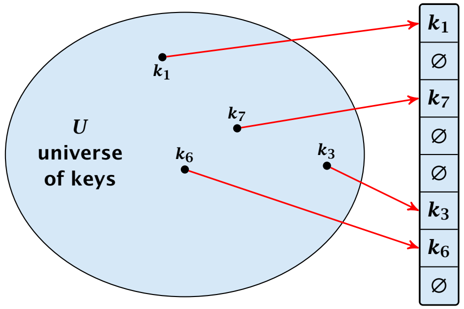
- Universe U of keys, e.g., $U \subseteq \mathbb{N}_0$. U very large.
- Set $S \subseteq U$ of keys, $|S| = m \leq |U|$.
- Array $T[0, \ldots, n-1]$ hash-table.
- Hash function $h : U \to [0, \ldots, n-1]$.

The hash-function h should fulfill:

- Fast to evaluate.
- Small storage requirement.
- Good distribution of elements over the whole table.

7 Dictionary

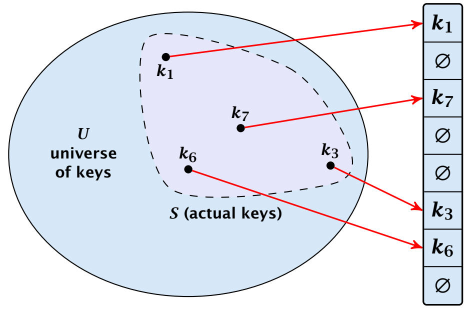
Ideally the hash function maps all keys to different memory locations.



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

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

$$1 - e^{-\frac{m(m-1)}{2n}} \approx 1 - e^{-\frac{m^2}{2n}} .$$

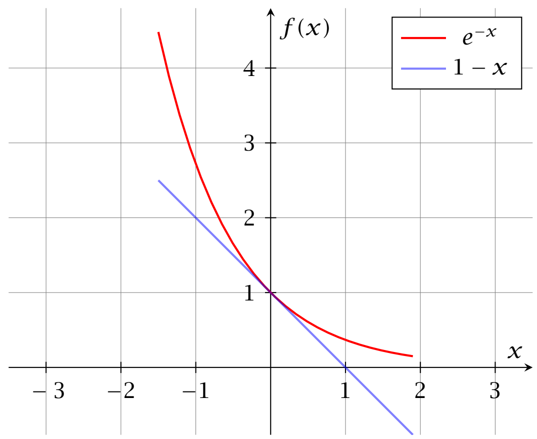
Uniform hashing:
Choose a hash function uniformly at random from all functions $f : U \to [0, \ldots, n-1]$.

7 Dictionary

Proof.
Let $A_{m,n}$ denote the event that inserting m keys into a table of size n does not generate a collision. Then

$$\Pr[A_{m,n}] = \prod_{\ell=1}^{m} \frac{n-\ell+1}{n} = \prod_{j=0}^{m-1}\left(1-\frac{j}{n}\right)$$
$$\le \prod_{j=0}^{m-1} e^{-j/n} = e^{-\sum_{j=0}^{m-1}\frac{j}{n}} = e^{-\frac{m(m-1)}{2n}} \;.$$

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

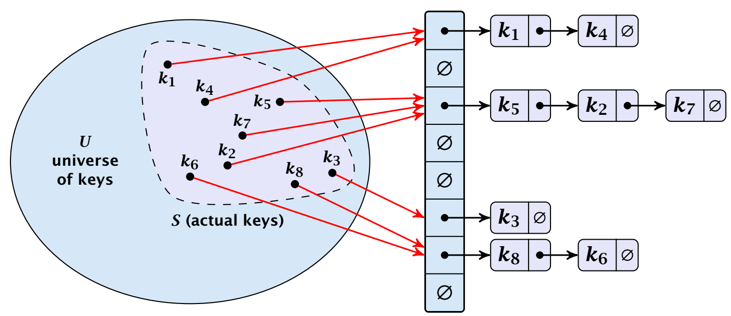
- **open addressing**, aka. closed hashing
- **hashing with chaining**, aka. closed addressing, open hashing.

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 $\text{key}[x]$.
- 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.

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

$$\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\underbrace{\left(1 + \overbrace{\sum_{j=i+1}^{m} X_{ij}}^{\text{keys before } k_i}\right)}_{\text{cost for key } k_i}\right]$$

Hashing with Chaining

$$\begin{aligned}
\mathrm{E}\left[\frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} X_{ij}\right)\right] &= \frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} \mathrm{E}\left[X_{ij}\right]\right) \\
&= \frac{1}{m}\sum_{i=1}^{m}\left(1 + \sum_{j=i+1}^{m} \frac{1}{n}\right) \\
&= 1 + \frac{1}{mn}\sum_{i=1}^{m}(m - i) \\
&= 1 + \frac{1}{mn}\left(m^2 - \frac{m(m+1)}{2}\right) \\
&= 1 + \frac{m-1}{2n} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2m} \ .
\end{aligned}$$

Hence, the expected cost for a successful search is $A^+ \le 1 + \frac{\alpha}{2}$.

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

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).

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:

$$Q^+ \approx 1 + \ln\left(\frac{1}{1-\alpha}\right) - \frac{\alpha}{2}$$

$$Q^- \approx \frac{1}{1-\alpha} + \ln\left(\frac{1}{1-\alpha}\right) - \alpha$$

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:

$$D^+ \approx \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$$

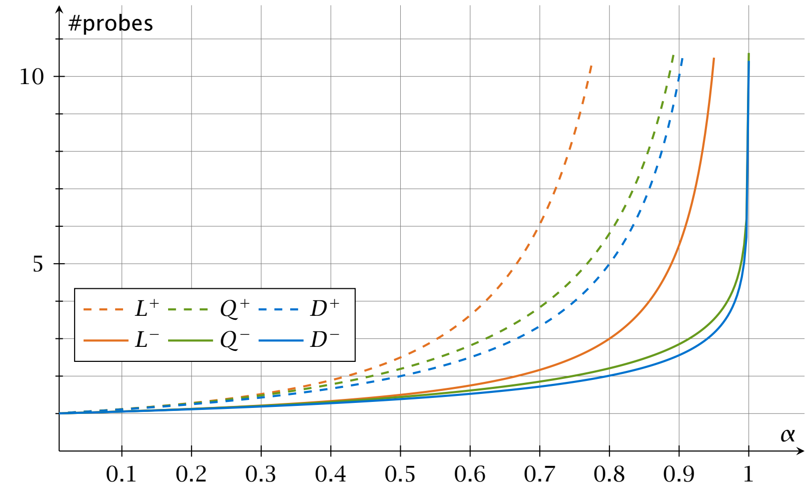
$$D^- \approx \frac{1}{1-\alpha}$$

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

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.

$$\begin{aligned}\Pr[A_1 \cap A_2 \cap \cdots \cap A_{i-1}] \\ &= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \cdot \\ &\quad \ldots \cdot \Pr[A_{i-1} \mid A_1 \cap \cdots \cap A_{i-2}]\end{aligned}$$

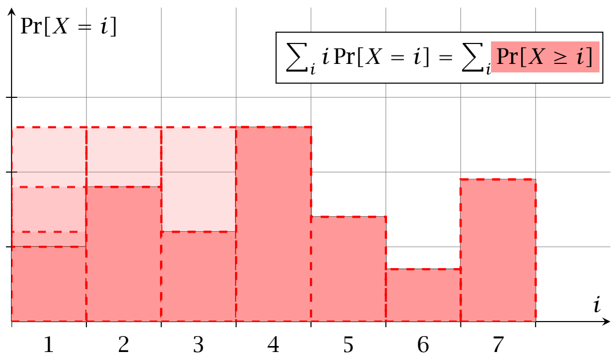
$$\begin{aligned}\Pr[X \geq i] &= \frac{m}{n} \cdot \frac{m-1}{n-1} \cdot \frac{m-2}{n-2} \cdot \ldots \cdot \frac{m-i+2}{n-i+2} \\ &\leq \left(\frac{m}{n}\right)^{i-1} = \alpha^{i-1} .\end{aligned}$$

Analysis of Idealized Open Address Hashing

$$\mathrm{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha} .$$

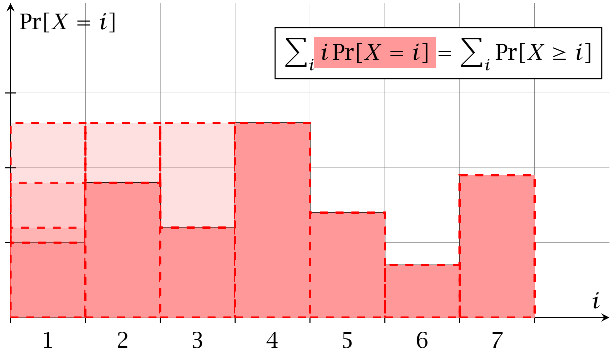
$$\frac{1}{1-\alpha} = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots$$

$i = 3$



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 $i = 1, 2, \ldots, j$)

$i = 4$



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 $i = 1, 2, \dots, j$)

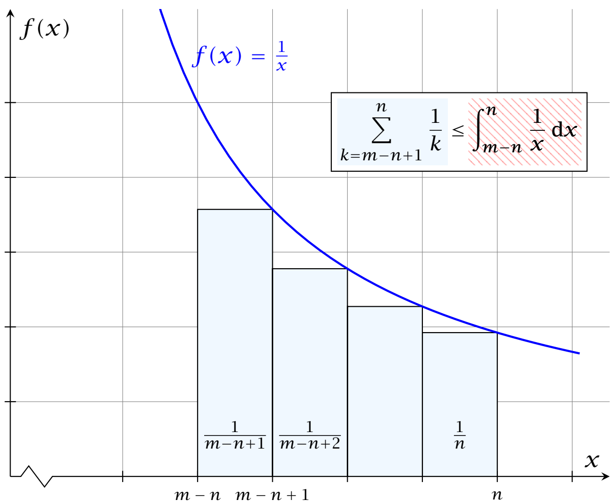
Analysis of Idealized Open Address Hashing

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 $\frac{1}{1-i/n} = \frac{n}{n-i}$.

$$\frac{1}{m}\sum_{i=0}^{m-1}\frac{n}{n-i} = \frac{n}{m}\sum_{i=0}^{m-1}\frac{1}{n-i} = \frac{1}{\alpha}\sum_{k=n-m+1}^{n}\frac{1}{k}$$

$$\le \frac{1}{\alpha}\int_{n-m}^{n}\frac{1}{x}\,dx = \frac{1}{\alpha}\ln\frac{n}{n-m} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}\ .$$



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 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)

$T[p] \leftarrow \text{null}$
$p \leftarrow \text{succ}(p)$
while $T[p] \neq \text{null}$ **do**
    $y \leftarrow T[p]$
    $T[p] \leftarrow \text{null}$
    $p \leftarrow \text{succ}(p)$
    $\text{insert}(y)$

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

Definition 17
A class $\mathcal{H}$ of hash-functions from the universe U into the set $\{0,\dots,n-1\}$ 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 $\{0,\dots,n-1\}$ is called 2-independent (pairwise independent) if the following two conditions hold

- For any key $u \in U$, and $t \in \{0,\dots,n-1\}$ $\Pr[h(u) = t] = \frac{1}{n}$, 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 $\{0,\dots,n-1\}$ is called 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 $t_1,\dots,t_\ell$:

$$\Pr[h(u_1) = t_1 \wedge \cdots \wedge h(u_\ell) = t_\ell] \leq \frac{1}{n^\ell} \ ,$$

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 $U := \{0, \dots, p-1\}$ for a prime p. Let $\mathbb{Z}_p := \{0, \dots, p-1\}$, and let $\mathbb{Z}_p^* := \{1, \dots, p-1\}$ 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 $\{0, \dots, n-1\}$.

Universal Hashing

Proof.

Let $x, y \in U$ be two distinct keys. We have to show that the probability of a collision is only $1/n$.

- $\boldsymbol{ax + b \not\equiv ay + b \pmod{p}}$

  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 $t_y := ay + b$.

  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 $\bmod n$-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

As $t_y \neq t_x$ there are

$$\left\lceil \frac{p}{n} \right\rceil - 1 \le \frac{p}{n} + \frac{n-1}{n} - 1 \le \frac{p-1}{n}$$

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

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 $d \in \mathbb{N}$; $q \ge (d+1)n$ 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.

Fix $\ell \le d+1$; 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 $h_{\bar a}(x_i) = t_i$).

- A^ℓ denotes the set of hash-functions such that every x_i hits its pre-defined position t_i.
- B_i is the set of positions that $f_{\bar a}$ can hit so that $h_{\bar a}$ still hits 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 $h_{\bar a} \in A_\ell$.

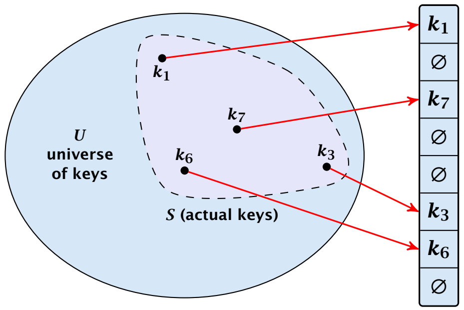
Therefore the probability of choosing $h_{\bar a}$ from A_ℓ is only

$$\frac{\lceil \frac{q}{n} \rceil^\ell \cdot q^{d-\ell+1}}{q^{d+1}} \le \frac{(\frac{q+n}{n})^\ell}{q^\ell} \le \left(\frac{q+n}{q}\right)^\ell \cdot \frac{1}{n^\ell}$$
$$\le \left(1 + \frac{1}{\ell}\right)^\ell \cdot \frac{1}{n^\ell} \le \frac{e}{n^\ell} .$$

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

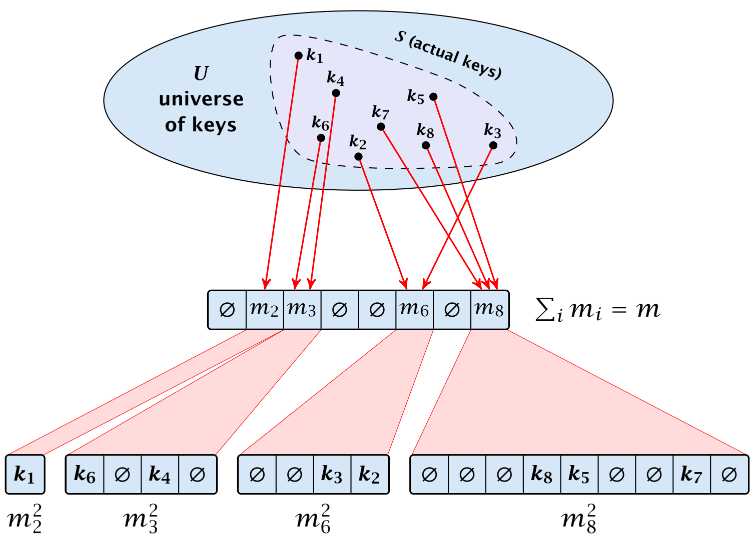
We can find such a hash-function by a few trials.

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

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

$$= 2\binom{m}{2}\frac{1}{m} + m = 2m - 1 \ .$$

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!!!

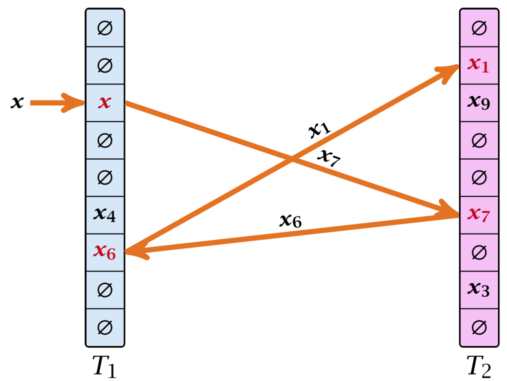
Cuckoo Hashing

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

Insert:



Cuckoo Hashing

Algorithm 17 Cuckoo-Insert(x)

1: **if** $T_1[h_1(x)] = x \vee T_2[h_2(x)] = x$ **then return**
2: steps $\leftarrow 1$
3: **while** steps $\le$ maxsteps **do**
4: exchange x and $T_1[h_1(x)]$
5: **if** $x =$ null **then return**
6: exchange x and $T_2[h_2(x)]$
7: **if** $x =$ null **then return**
8: steps $\leftarrow$ steps $+1$
9: rehash() // change hash-functions; rehash everything
10: 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 $x = \text{null}$.

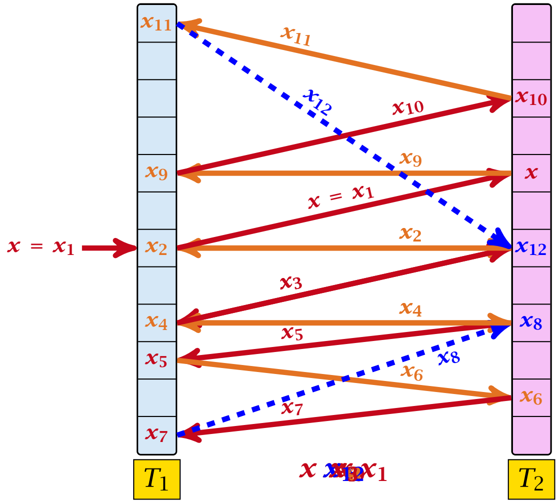
Cuckoo Hashing

What is the expected time for an insert-operation?

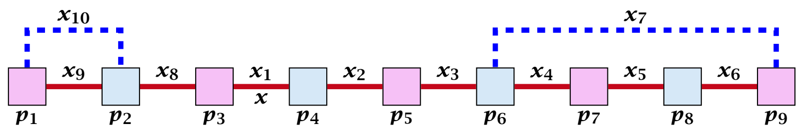
We first analyze the probability that we end-up in an infinite loop (that is then terminated after maxsteps steps).

Formally what is the probability to enter an infinite loop that touches s different keys?

Cuckoo Hashing: Insert



Cuckoo Hashing



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

$$h_1(x_\ell) = p_i \quad \text{and} \quad h_2(x_\ell) = p_j$$

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

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 probability that there exists an active cycle-structure is therefore at most

$$\sum_{s=3}^{\infty} s^3 \cdot n^{s-1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}} = \frac{\mu^2}{nm} \sum_{s=3}^{\infty} s^3 \left(\frac{m}{n}\right)^s$$
$$\le \frac{\mu^2}{m^2} \sum_{s=3}^{\infty} s^3 \left(\frac{1}{1+\epsilon}\right)^s \le \mathcal{O}\left(\frac{1}{m^2}\right) .$$

Here we used the fact that $(1+\epsilon)m \le n$.

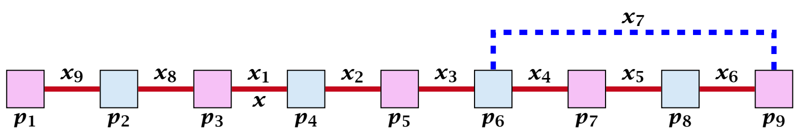
Hence,

$$\Pr[\text{cycle}] = \mathcal{O}\left(\frac{1}{m^2}\right) .$$

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

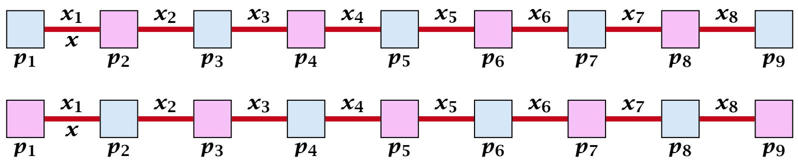
If the sequence is of length p then there exists a sub-sequence of at least $p/3$ keys starting with x of distinct keys.

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



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 $x = x_1, x_2, \dots, x_s$, 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

$$h_1(x_\ell) = p_i \quad \text{and} \quad h_2(x_\ell) = p_j$$

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 $\frac{\mu^2}{n^{2s}}$.

The probability that there exists an active path-structure of size s is at most

$$2 \cdot n^{s+1} \cdot m^{s-1} \cdot \frac{\mu^2}{n^{2s}}$$
$$\leq 2\mu^2 \left(\frac{m}{n}\right)^{s-1} \leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{s-1}$$
$$\leq 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{(2t-1)/3-1} = 2\mu^2 \left(\frac{1}{1+\epsilon}\right)^{2(t-2)/3} .$$

Cuckoo Hashing

We choose $\mathrm{maxsteps} \geq \frac{3\ell}{2} + 1$. Then the probability that a phase is terminated unsuccessfully without running into a cycle is at most

$$\Pr[\text{unsuccessful} \mid \text{no cycle}]$$
$$\leq \Pr[\exists \text{ active path-structure of size at least } \tfrac{2(\mathrm{maxsteps}+1)-1}{3}]$$
$$\leq \Pr[\exists \text{ active path-structure of size at least } \ell+1]$$
$$\leq 2\mu^2\Big(\frac{1}{1+\epsilon}\Big)^{\ell} \leq \frac{1}{m^2}$$

by choosing $\ell \geq \log\left(\frac{1}{2\mu^2 m^2}\right)/\log\left(\frac{1}{1+\epsilon}\right) = \log\left(2\mu^2 m^2\right)/\log\left(1+\epsilon\right)$

Note that this gives $\mathrm{maxsteps} = \Theta(\log m)$.

Cuckoo Hashing

The expected number of steps in the successful phase of an insert operation is:

$$\begin{aligned}&\mathrm{E}[\text{number of steps} \mid \text{phase successful}]\\&\quad= \sum_{t\ge1}\Pr[\text{search takes at least } t \text{ steps} \mid \text{phase successful}]\end{aligned}$$

We have

$$\begin{aligned}&\Pr[\text{search at least } t \text{ steps} \mid \text{successful}]\\&\quad= \Pr[\text{search at least } t \text{ steps} \wedge \text{successful}]/\Pr[\text{successful}]\\&\quad\le \frac{1}{c}\Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}]\ ,\end{aligned}$$

where we use the fact that for a suitable **constant** $c \ge 0$

$$\begin{aligned}\Pr[\text{successful}] &= \Pr[\text{no cycle}] - \Pr[\text{unsuccessful} \mid \text{no cycle}]\\&\ge c\cdot\Pr[\text{no cycle}]\end{aligned}$$

Hence,

$$\begin{aligned}&\mathrm{E}[\text{number of steps} \mid \text{phase successful}]\\&\quad= \frac{1}{c}\sum_{t\ge1}\Pr[\text{search at least } t \text{ steps} \mid \text{no cycle}]\\&\quad\le \frac{1}{c}\Big[1+\sum_{t\ge2}2\mu^2\Big(\frac{1}{1+\epsilon}\Big)^{2(t-2)/3}\Big]\\&\quad= \frac{1}{c}+\frac{2\mu^2}{c}\sum_{t\ge0}\Big(\frac{1}{(1+\epsilon)^{2/3}}\Big)^t = \mathcal{O}(1)\ .\end{aligned}$$

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 $\sum_{i\ge1}p^i = \frac{1}{1-p}-1 = \frac{p}{1-p} = \mathcal{O}(p)$.

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 $\mathrm{maxsteps}$ 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

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 $\frac{1}{2(1+\epsilon)}$.

8 Priority Queues

A Priority Queue S is a dynamic set data structure that supports the following operations:

- **S.build$(x_1, \ldots, x_n)$:** Creates a data-structure that contains just the elements $x_1, \ldots, x_n$.
- **S.insert(x):** Adds element x to the data-structure.
- **element S.minimum()**: Returns an element $x \in S$ with minimum key-value $\text{key}[x]$.
- **element S.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **boolean S.is-empty()**: Returns true if the data-structure is empty and false otherwise.

Sometimes we also have

- **S.merge(S')**: $S := S \cup S'$; $S' := \emptyset$.

8 Priority Queues

An addressable Priority Queue also supports:

- **handle S.insert(x):** Adds element x to the data-structure, and returns a handle to the object for future reference.
- **S.delete(h):** Deletes element specified through handle h.
- **S.decrease-key(h, k)**: Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

Dijkstra's Shortest Path Algorithm

Algorithm 18 Shortest-Path($G = (V, E, d), s \in V$)

```
Input: weighted graph G = (V, E, d); start vertex s;
Output: key-field of every node contains distance from s;
S.build(); // build empty priority queue
for all v ∈ V \ {s} do
    v.key ← ∞;
    h_v ← S.insert(v);
s.key ← 0; S.insert(s);
while S.is-empty() = false do
    v ← S.delete-min();
    for all x ∈ V s.t. (v, x) ∈ E do
        if x.key > v.key + d(v, x) then
            S.decrease-key(h_x, v.key + d(v, x));
            x.key ← v.key + d(v, x);
```


Prim's Minimum Spanning Tree Algorithm

Algorithm 19 Prim-MST($G = (V, E, d), s \in V$)

```
Input: weighted graph G = (V, E, d); start vertex s;
Output: pred-fields encode MST;
S.build(); // build empty priority queue
for all v ∈ V \ {s} do
    v.key ← ∞;
    h_v ← S.insert(v);
s.key ← 0; S.insert(s);
while S.is-empty() = false do
    v ← S.delete-min();
    for all x ∈ V s.t. {v, x} ∈ E do
        if x.key > d(v, x) then
            S.decrease-key(h_x, d(v, x));
            x.key ← d(v, x);
            x.pred ← v;
```


Analysis of Dijkstra and Prim

Both algorithms require:

- 1 build() operation
- $|V|$ insert() operations
- $|V|$ delete-min() operations
- $|V|$ is-empty() operations
- $|E|$ decrease-key() operations

How good a running time can we obtain?

8 Priority Queues

Operation	*Binary Heap*	*BST*	*Binomial Heap*	*Fibonacci Heap**
build	n	$n \log n$	$n \log n$	n
minimum	1	$\log n$	$\log n$	1
is-empty	1	1	1	1
insert	$\log n$	$\log n$	$\log n$	1
delete	$\log n$**	$\log n$	$\log n$	$\log n$
delete-min	$\log n$	$\log n$	$\log n$	$\log n$
decrease-key	$\log n$	$\log n$	$\log n$	1
merge	n	$n \log n$	$\log n$	1

Note that most applications use **build()** only to create an empty heap which then costs time 1.

* Fibonacci heaps only give an amortized guarantee.

** The standard version of binary heaps is not addressable. Hence, it does not support a delete.

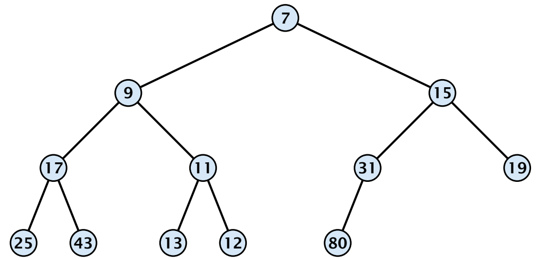
8 Priority Queues

Using Binary Heaps, Prim and Dijkstra run in time $\mathcal{O}((|V| + |E|) \log |V|)$.

Using Fibonacci Heaps, Prim and Dijkstra run in time $\mathcal{O}(|V| \log |V| + |E|)$.

8.1 Binary Heaps

- Nearly complete binary tree; only the last level is not full, and this one is filled from left to right.
- Heap property: A node's key is not larger than the key of one of its children.



Binary Heaps

Operations:

- **minimum():** return the root-element. Time $\mathcal{O}(1)$.
- **is-empty():** check whether root-pointer is null. Time $\mathcal{O}(1)$.

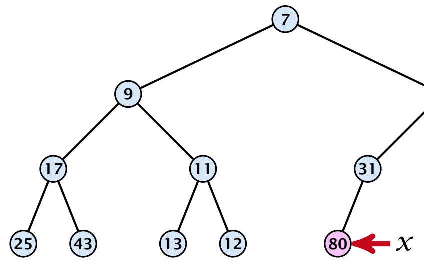
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the predecessor of x (last element when x is deleted) in time $\mathcal{O}(\log n)$.

  go up until the last edge used was a right edge.
  go left; go right until you reach a leaf

  if you hit the root on the way up, go to the rightmost element



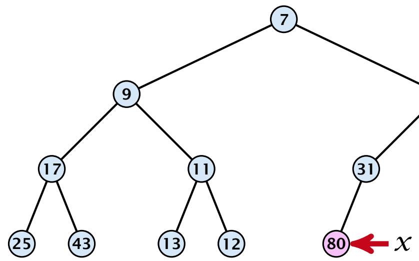
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the successor of x (last element when an element is inserted) in time $\mathcal{O}(\log n)$.

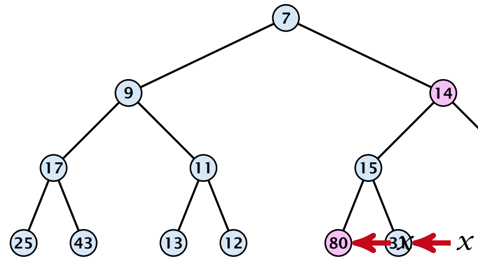
  go up until the last edge used was a left edge.
  go right; go left until you reach a null-pointer.

  if you hit the root on the way up, go to the leftmost element; insert a new element as a left child;



Insert

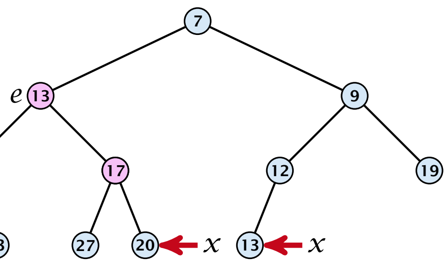
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



Note that an exchange can either be done by moving the data or by changing pointers. The latter method leads to an addressable priority queue.

Delete

1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



At its new position e may either travel up or down in the tree (but not both directions).

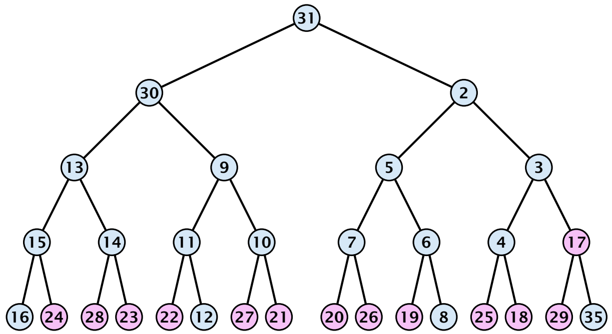
Binary Heaps

Operations:

- **minimum():** return the root-element. Time $\mathcal{O}(1)$.
- **is-empty():** check whether root-pointer is null. Time $\mathcal{O}(1)$.
- **insert(k):** insert at x and bubble up. Time $\mathcal{O}(\log n)$.
- **delete(h):** swap with x and bubble up or sift-down. Time $\mathcal{O}(\log n)$.

Build Heap

We can build a heap in linear time:



$$\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \mathcal{O}(2^h) = \mathcal{O}(n)$$

Binary Heaps

Operations:

- **minimum():** Return the root-element. Time $\mathcal{O}(1)$.
- **is-empty():** Check whether root-pointer is null. Time $\mathcal{O}(1)$.
- **insert(k):** Insert at x and bubble up. Time $\mathcal{O}(\log n)$.
- **delete(h):** Swap with x and bubble up or sift-down. Time $\mathcal{O}(\log n)$.
- **build($x_1, \dots, x_n$):** Insert elements arbitrarily; then do sift-down operations starting with the lowest layer in the tree. Time $\mathcal{O}(n)$.

Binary Heaps

The standard implementation of binary heaps is via arrays. Let $A[0, \dots, n-1]$ be an array

- The parent of i-th element is at position $\lfloor \frac{i-1}{2} \rfloor$.
- The left child of i-th element is at position $2i + 1$.
- The right child of i-th element is at position $2i + 2$.

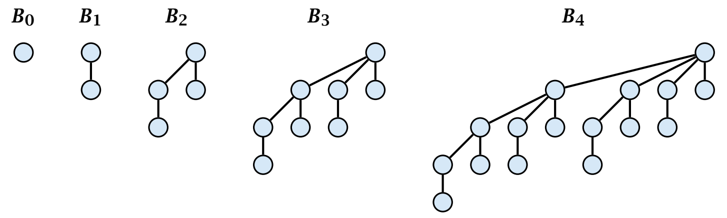
Finding the successor of x is much easier than in the description on the previous slide. Simply increase or decrease x.

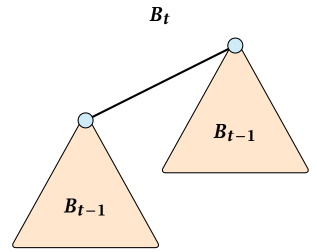
The resulting binary heap is not addressable. The elements don't maintain their positions and therefore there are no stable handles.

8.2 Binomial Heaps

Operation	*Binary Heap*	*BST*	*Binomial Heap*	*Fibonacci Heap**
build	n	$n \log n$	$n \log n$	n
minimum	1	$\log n$	$\log n$	1
is-empty	1	1	1	1
insert	$\log n$	$\log n$	$\log n$	1
delete	$\log n^{**}$	$\log n$	$\log n$	$\log n$
delete-min	$\log n$	$\log n$	$\log n$	$\log n$
decrease-key	$\log n$	$\log n$	$\log n$	1
merge	n	$n \log n$	$\log n$	1

Binomial Trees



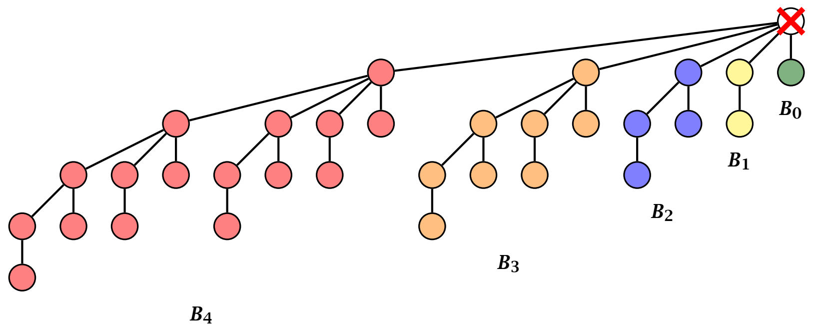


Binomial Trees

Properties of Binomial Trees

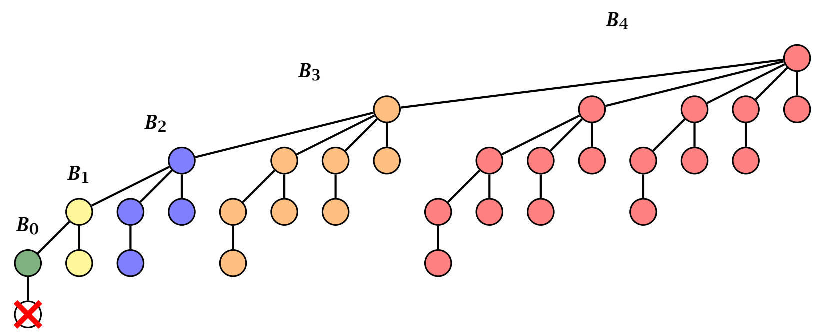
- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

Binomial Trees



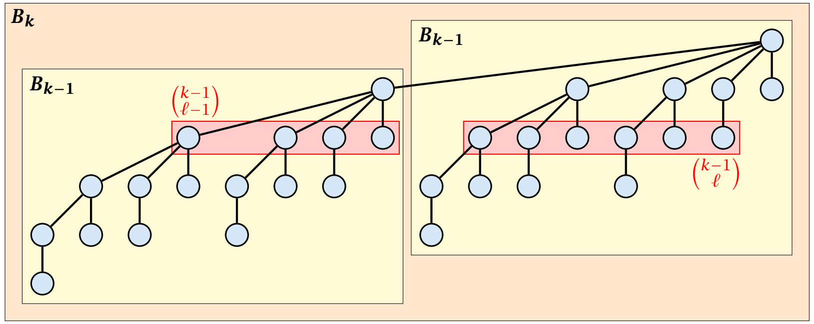
Deleting the root of B_5 leaves sub-trees B_4, B_3, B_2, B_1, and B_0.

Binomial Trees



Deleting the leaf furthest from the root (in B_5) leaves a path that connects the roots of sub-trees B_4, B_3, B_2, B_1, and B_0.

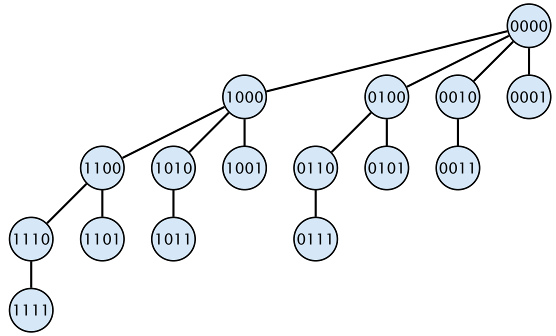
Binomial Trees



The number of nodes on level ℓ in tree B_k is therefore

$$\binom{k-1}{\ell-1} + \binom{k-1}{\ell} = \binom{k}{\ell}$$

Binomial Trees



The binomial tree B_k is a sub-graph of the hypercube H_k.

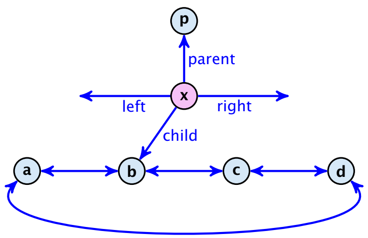
The parent of a node with label $b_n, \ldots, b_1, b_0$ is obtained by setting the least significant 1-bit to 0.

The ℓ-th level contains nodes that have ℓ 1's in their label.

8.2 Binomial Heaps

How do we implement trees with non-constant degree?

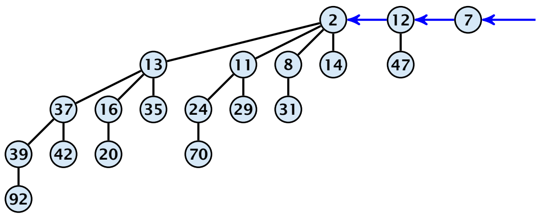
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\text{left}$ and $x.\text{right}$ point to the left and right sibling of x (if x does not have siblings then $x.\text{left} = x.\text{right} = x$).



8.2 Binomial Heaps

- Given a pointer to a node x we can splice out the sub-tree rooted at x in constant time.
- We can add a child-tree T to a node x in constant time if we are given a pointer to x and a pointer to the root of T.

Binomial Heap



In a binomial heap the keys are arranged in a collection of binomial trees.

Every tree fulfills the heap-property

There is at most one tree for every dimension/order. For example the above heap contains trees B_0, B_1, and B_4.

Binomial Heap: Merge

Given the number n of keys to be stored in a binomial heap we can deduce the binomial trees that will be contained in the collection.

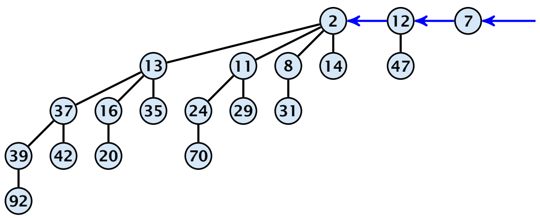
Let $B_{k_1}, B_{k_2}, B_{k_3}, k_i < k_{i+1}$ denote the binomial trees in the collection and recall that every tree may be contained at most once.

Then $n = \sum_i 2^{k_i}$ must hold. But since the k_i are all distinct this means that the k_i define the non-zero bit-positions in the dual representation of n.

Binomial Heap

Properties of a heap with n keys:

- Let $n = b_d b_{d-1}, \dots, b_0$ denote the dual representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees are stored in a single-linked list; ordered by dimension/size.



Binomial Heap: Merge

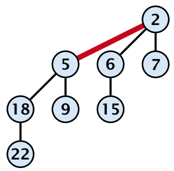
The merge-operation is instrumental for binomial heaps.

A merge is easy if we have two heaps with different binomial trees. We can simply merge the tree-lists.

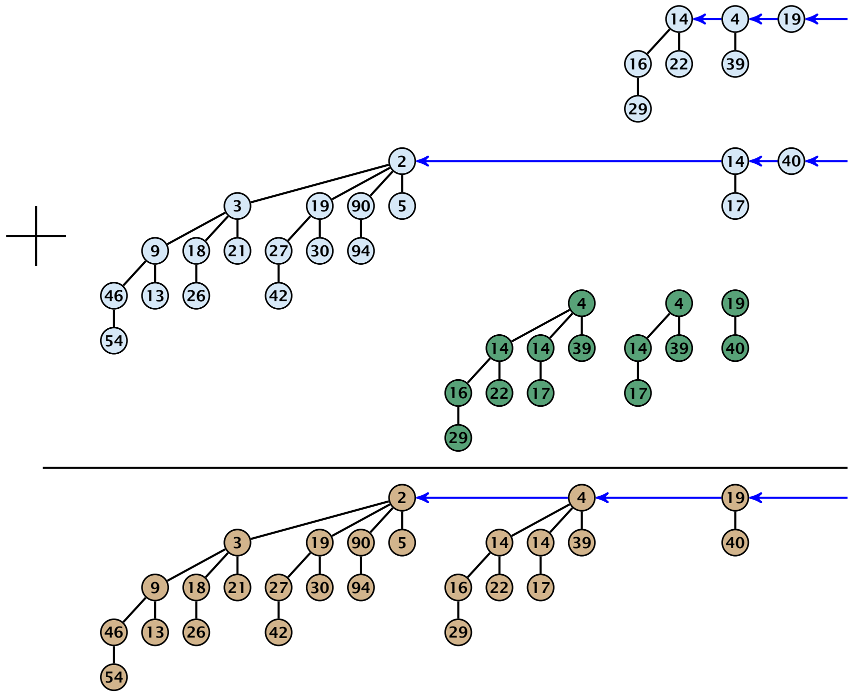
Note that we do not just do a concatenation as we want to keep the trees in the list sorted according to size.

Otherwise, we cannot do this because the merged heap is not allowed to contain two trees of the same order.

Merging two trees of the same size: Add the tree with larger root-value as a child to the other tree.



For more trees the technique is analogous to binary addition.



8.2 Binomial Heaps

S_1.merge(S_2):

- Analogous to binary addition.
- Time is proportional to the number of trees in both heaps.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

All other operations can be reduced to merge().

S.insert(x):

- Create a new heap S' that contains just the element x.
- Execute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.minimum():

- Find the minimum key-value among all roots.
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete-min():

- Find the minimum key-value among all roots.
- Remove the corresponding tree $T_{\min}$ from the heap.
- Create a new heap S' that contains the trees obtained from $T_{\min}$ after deleting the root (note that these are just $\mathcal{O}(\log n)$ trees).
- Compute S.merge(S').
- Time: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.decrease-key(handle h):

- Decrease the key of the element pointed to by h.
- Bubble the element up in the tree until the heap property is fulfilled.
- Time: $\mathcal{O}(\log n)$ since the trees have height $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete(handle h):

- Execute S.decrease-key$(h, -\infty)$.
- Execute S.delete-min().
- Time: $\mathcal{O}(\log n)$.

Amortized Analysis

Definition 25
A data structure with operations $\text{op}_1(), \ldots, \text{op}_k()$ has amortized running times $t_1, \ldots, t_k$ for these operations if the following holds.

Suppose you are given a sequence of operations (starting with an empty data-structure) that operate on at most n elements, and let k_i denote the number of occurences of $\text{op}_i()$ within this sequence. Then the actual running time must be at most $\sum_i k_i t_i(n)$.

Potential Method

Introduce a potential for the data structure.

- $\Phi(D_i)$ is the potential after the i-th operation.
- Amortized cost of the i-th operation is

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \ .$$

- Show that $\Phi(D_i) \geq \Phi(D_0)$.

Then

$$\sum_{i=1}^{k} c_i \leq \sum_{i+1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^{k} \hat{c}_i$$

This means the amortized costs can be used to derive a bound on the total cost.

Example: Stack

Stack

- $\boldsymbol{S.}$ **push()**
- $\boldsymbol{S.}$ **pop()**
- $\boldsymbol{S.}$ **multipop(k)**: removes k items from the stack. If the stack currently contains less than k items it empties the stack.
- The user has to ensure that pop and multipop do not generate an underflow.

Actual cost:

- $\boldsymbol{S.}$ **push()**: cost 1.
- $\boldsymbol{S.}$ **pop()**: cost 1.
- $\boldsymbol{S.}$ **multipop(k)**: cost $\min\{\text{size}, k\} = k$.

Example: Stack

Use potential function $\Phi(S)$ = number of elements on the stack.

Amortized cost:

- $\boldsymbol{S.}$ **push()**: cost

$$\hat{C}_{\text{push}} = C_{\text{push}} + \Delta\Phi = 1 + 1 \leq 2 \ .$$

- $\boldsymbol{S.}$ **pop()**: cost

$$\hat{C}_{\text{pop}} = C_{\text{pop}} + \Delta\Phi = 1 - 1 \leq 0 \ .$$

- $\boldsymbol{S.}$ **multipop(k)**: cost

$$\hat{C}_{\text{mp}} = C_{\text{mp}} + \Delta\Phi = \min\{\text{size}, k\} - \min\{\text{size}, k\} \leq 0 \ .$$

Note that the analysis becomes wrong if pop() or multipop() are called on an empty stack.

Example: Binary Counter

Incrementing a binary counter:
Consider a computational model where each bit-operation costs one time-unit.

Incrementing an n-bit binary counter may require to examine n-bits, and maybe change them.

Actual cost:

- Changing bit from 0 to 1: cost 1.
- Changing bit from 1 to 0: cost 1.
- Increment: cost is $k + 1$, where k is the number of consecutive ones in the least significant bit-positions (e.g, 001101 has $k = 1$).

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from 0 to 1:

$$\hat{C}_{0\to 1} = C_{0\to 1} + \Delta\Phi = 1 + 1 \le 2 \ .$$

- Changing bit from 1 to 0:

$$\hat{C}_{1\to 0} = C_{1\to 0} + \Delta\Phi = 1 - 1 \le 0 \ .$$

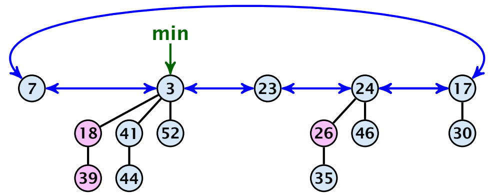
- Increment: Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \le 2$.

8.3 Fibonacci Heaps

Collection of trees that fulfill the heap property.

Structure is much more relaxed than binomial heaps.



8.3 Fibonacci Heaps

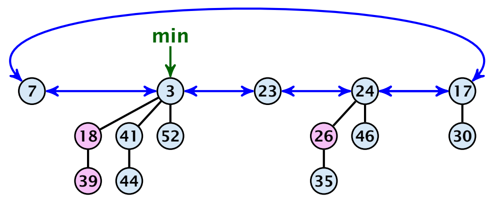
Additional implementation details:

- Every node x stores its degree in a field x.degree. Note that this can be updated in constant time when adding a child to x.
- Every node stores a boolean value x.marked that specifies whether x is marked or not.

8.3 Fibonacci Heaps

The potential function:

- $t(S)$ denotes the number of trees in the heap.
- $m(S)$ denotes the number of marked nodes.
- We use the potential function $\Phi(S) = t(S) + 2m(S)$.



The potential is $\Phi(S) = 5 + 2 \cdot 3 = 11$.

8.3 Fibonacci Heaps

We assume that one unit of potential can pay for a constant amount of work, where the constant is chosen “big enough” (to take care of the constants that occur).

To make this more explicit we use c to denote the amount of work that a unit of potential can pay for.

8.3 Fibonacci Heaps

$S.$ **minimum()**

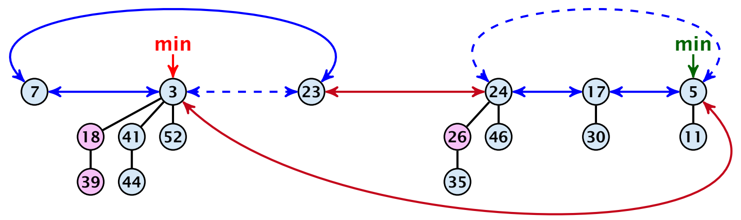
- Access through the min-pointer.
- Actual cost $\mathcal{O}(1)$.
- No change in potential.
- Amortized cost $\mathcal{O}(1)$.

8.3 Fibonacci Heaps

- In the figure below the dashed edges are replaced by red edges.
- The minimum of the left heap becomes the new minimum of the merged heap.

$S.$ **merge**(S')

- Merge the root lists.
- Adjust the min-pointer



Running time:

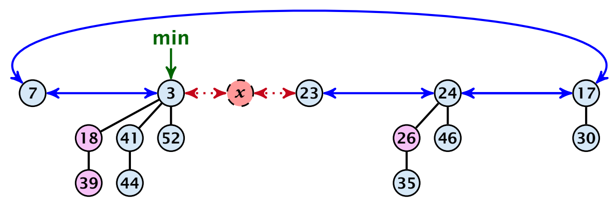
- Actual cost $\mathcal{O}(1)$.
- No change in potential.
- Hence, amortized cost is $\mathcal{O}(1)$.

8.3 Fibonacci Heaps

x is inserted next to the min-pointer as this is our entry point into the root-list.

$S.$ **insert**(x)

- Create a new tree containing x.
- Insert x into the root-list.
- Update min-pointer, if necessary.



Running time:

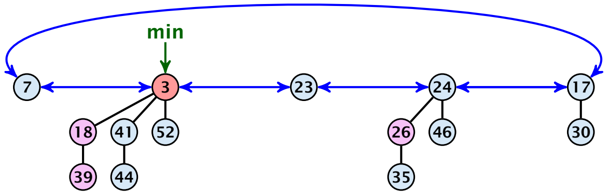
- Actual cost $\mathcal{O}(1)$.
- Change in potential is $+1$.
- Amortized cost is $c + \mathcal{O}(1) = \mathcal{O}(1)$.

8.3 Fibonacci Heaps

$D(\min)$ is the number of children of the node that stores the minimum.

S. delete-min(x)

- Delete minimum; add child-trees to heap; time: $D(\min) \cdot \mathcal{O}(1)$.
- Update min-pointer; time: $(t + D(\min)) \cdot \mathcal{O}(1)$.

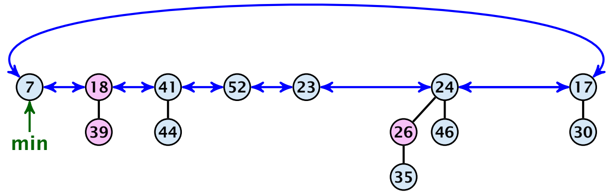


8.3 Fibonacci Heaps

$D(\min)$ is the number of children of the node that stores the minimum.

S. delete-min(x)

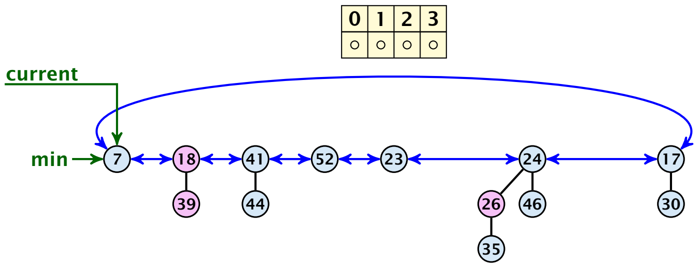
- Delete minimum; add child-trees to heap; time: $D(\min) \cdot \mathcal{O}(1)$.
- Update min-pointer; time: $(t + D(\min)) \cdot \mathcal{O}(1)$.



- Consolidate root-list so that no roots have the same degree. Time $t \cdot \mathcal{O}(1)$ (see next slide).

8.3 Fibonacci Heaps

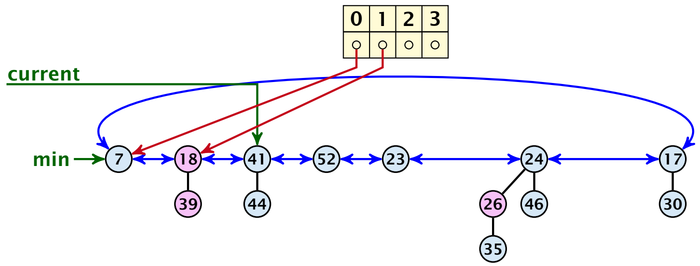
Consolidate:



During the consolidation we traverse the root list. Whenever we discover two trees that have the same degree we merge these trees. In order to efficiently check whether two trees have the same degree, we use an array that contains for every degree value d a pointer to a tree left of the current pointer whose root has degree d (if such a tree exist).

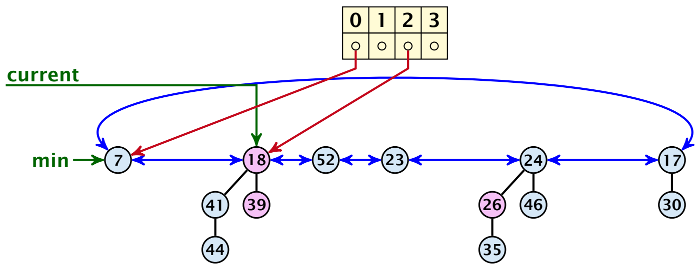
8.3 Fibonacci Heaps

Consolidate:



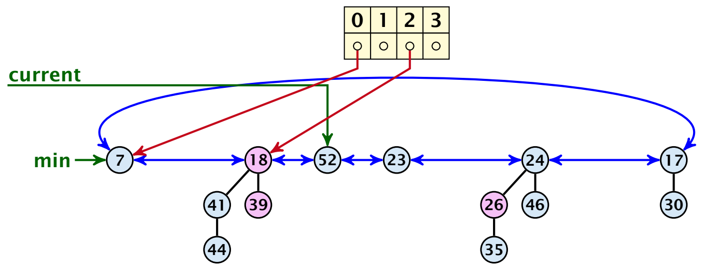
8.3 Fibonacci Heaps

Consolidate:



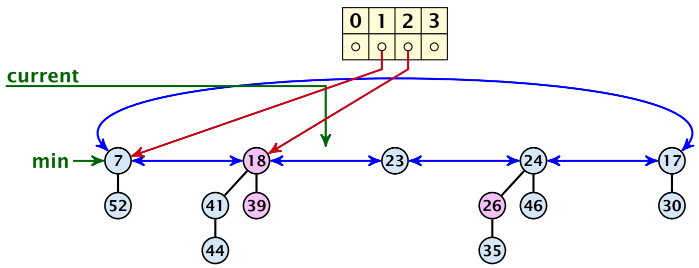
8.3 Fibonacci Heaps

Consolidate:



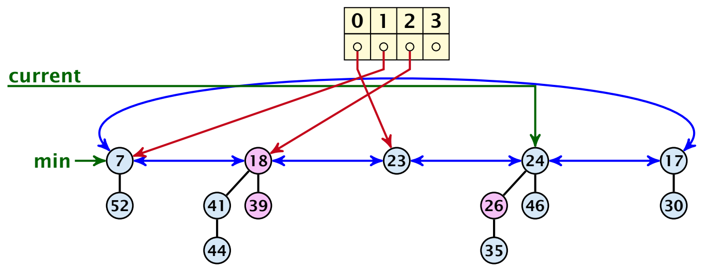
8.3 Fibonacci Heaps

Consolidate:



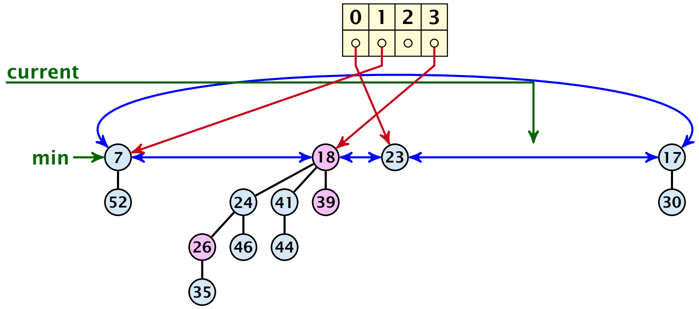
8.3 Fibonacci Heaps

Consolidate:



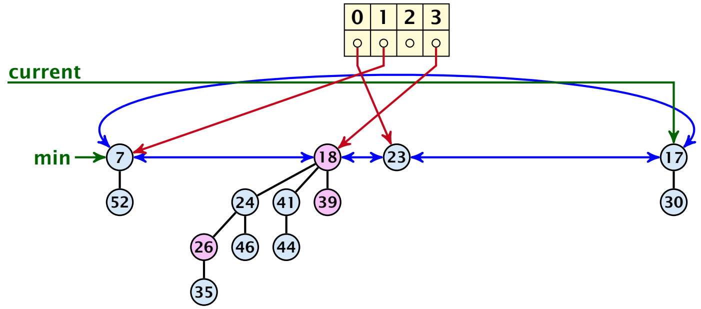
8.3 Fibonacci Heaps

Consolidate:



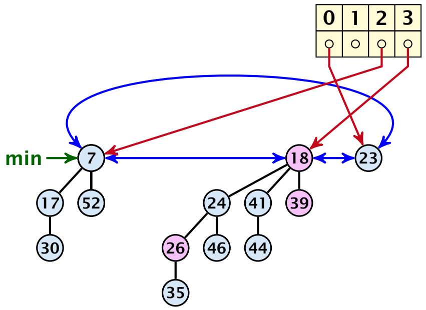
8.3 Fibonacci Heaps

Consolidate:



8.3 Fibonacci Heaps

Consolidate:



8.3 Fibonacci Heaps

t and t' denote the number of trees before and after the delete-min() operation, respectively. D_n is an upper bound on the degree (i.e., number of children) of a tree node.

Actual cost for delete-min()

- At most $D_n + t$ elements in root-list before consolidate.
- Actual cost for a delete-min is at most $\mathcal{O}(1) \cdot (D_n + t)$. Hence, there exists c_1 s.t. actual cost is at most $c_1 \cdot (D_n + t)$.

Amortized cost for delete-min()

- $t' \leq D_n + 1$ as degrees are different after consolidating.
- Therefore $\Delta\Phi \leq D_n + 1 - t$;
- We can pay $c \cdot (t - D_n - 1)$ from the potential decrease.
- The amortized cost is

$$c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1)$$
$$\leq (c_1 + c)D_n + (c_1 - c)t + c \leq 2c(D_n + 1) \leq \mathcal{O}(D_n)$$

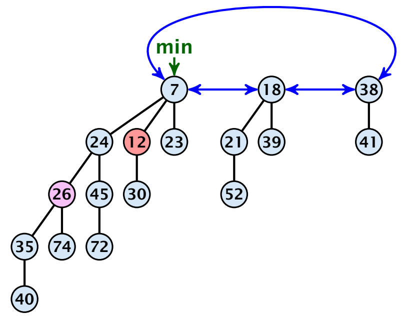
for $c \geq c_1$.

8.3 Fibonacci Heaps

If the input trees of the consolidation procedure are binomial trees (for example only singleton vertices) then the output will be a set of distinct binomial trees, and, hence, the Fibonacci heap will be (more or less) a Binomial heap right after the consolidation.

If we do not have delete or decrease-key operations then $D_n \le \log n$.

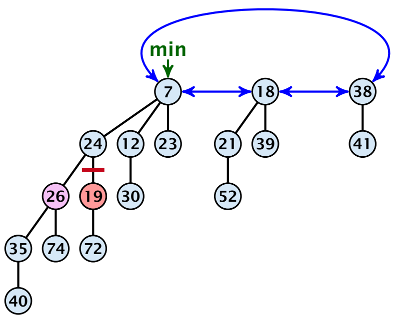
Fibonacci Heaps: decrease-key(handle h, v)



Case 1: decrease-key does not violate heap-property

- Just decrease the key-value of element referenced by h. Nothing else to do.

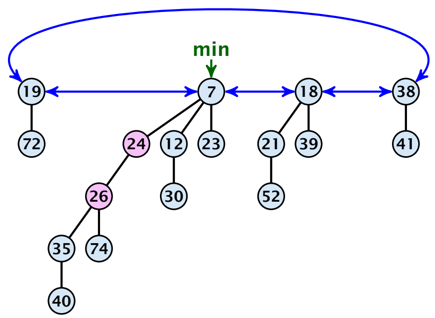
Fibonacci Heaps: decrease-key(handle h, v)



Case 2: heap-property is violated, but parent is not marked

- Decrease key-value of element x reference by h.
- If the heap-property is violated, cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Mark the (previous) parent of x (unless it's a root).

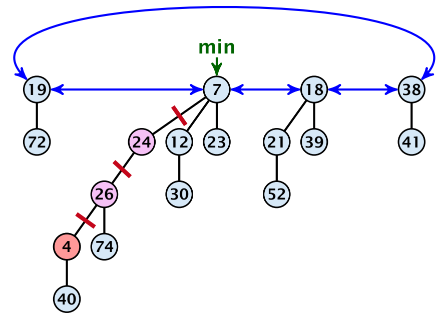
Fibonacci Heaps: decrease-key(handle h, v)



Case 2: heap-property is violated, but parent is not marked

- Decrease key-value of element x reference by h.
- If the heap-property is violated, cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Mark the (previous) parent of x (unless it's a root).

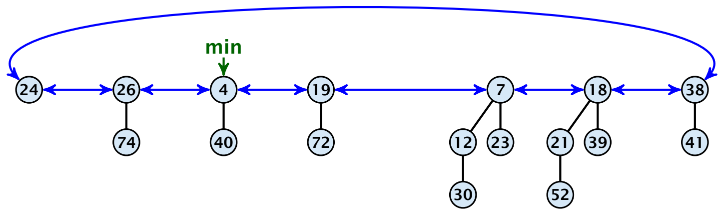
Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

Fibonacci Heaps: decrease-key(handle h, v)



Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

Fibonacci Heaps: decrease-key(handle h, v)

Case 3: heap-property is violated, and parent is marked

- Decrease key-value of element x reference by h.
- Cut the parent edge of x, and make x into a root.
- Adjust min-pointers, if necessary.
- Execute the following:

  $p \leftarrow \text{parent}[x]$;
  while (p is marked)
    $pp \leftarrow \text{parent}[p]$;
    cut of p; make it into a root; unmark it;
    $p \leftarrow pp$;
  if p is unmarked and not a root mark it;

Marking a node can be viewed as a first step towards becoming a root. The first time x loses a child it is marked; the second time it loses a child it is made into a root.

Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most $c_2 \cdot (\ell + 1)$, for some constant c_2.

Amortized cost:

- $t' = t + \ell$, as every cut creates one new root.
- $m' \le m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut unmarks a node; the last cut may mark a node.
- $\Delta\Phi \le \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

  $$c_2(\ell + 1) + c(4 - \ell) \le (c_2 - c)\ell + 4c = \mathcal{O}(1),$$

  if $c \ge c_2$.

t and t': number of trees before and after operation.
m and m': number of marked nodes before and after operation.

Delete node

$\boldsymbol{H}.\mathbf{delete}(\boldsymbol{x})$:

- decrease value of x to $-\infty$.
- delete-min.

Amortized cost: $\mathcal{O}(D(n))$

- $\mathcal{O}(1)$ for decrease-key.
- $\mathcal{O}(D(n))$ for delete-min.

8.3 Fibonacci Heaps

Lemma 26

Let x be a node with degree k and let $y_1, \ldots, y_k$ denote the children of x in the order that they were linked to x. Then

$$\text{degree}(y_i) \geq \begin{cases} 0 & \text{if } i = 1 \\ i-2 & \text{if } i > 1 \end{cases}$$

The marking process is very important for the proof of this lemma. It ensures that a node can have lost at most one child since the last time it became a non-root node. When losing a first child the node gets marked; when losing the second child it is cut from the parent and made into a root.

8.3 Fibonacci Heaps

Proof

- When y_i was linked to x, at least $y_1, \ldots, y_{i-1}$ were already linked to x.
- Hence, at this time $\text{degree}(x) \geq i-1$, and therefore also $\text{degree}(y_i) \geq i-1$ as the algorithm links nodes of equal degree only.
- Since, then y_i has lost at most one child.
- Therefore, $\text{degree}(y_i) \geq i-2$.

8.3 Fibonacci Heaps

- Let s_k be the minimum possible size of a sub-tree rooted at a node of degree k that can occur in a Fibonacci heap.
- s_k monotonically increases with k
- $s_0 = 1$ and $s_1 = 2$.

Let x be a degree k node of size s_k and let $y_1, \ldots, y_k$ be its children.

$$\begin{aligned} s_k &= 2 + \sum_{i=2}^{k} \text{size}(y_i) \\ &\geq 2 + \sum_{i=2}^{k} s_{i-2} \\ &= 2 + \sum_{i=0}^{k-2} s_i \end{aligned}$$

8.3 Fibonacci Heaps

Definition 27

Consider the following non-standard Fibonacci type sequence:

$$F_k = \begin{cases} 1 & \text{if } k = 0 \\ 2 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \geq 2 \end{cases}$$

Facts:

1. $F_k \geq \phi^k$.
2. For $k \geq 2$: $F_k = 2 + \sum_{i=0}^{k-2} F_i$.

The above facts can be easily proved by induction. From this it follows that $s_k \geq F_k \geq \phi^k$, which gives that the maximum degree in a Fibonacci heap is logarithmic.

9 Union Find

Union Find Data Structure $\mathcal{P}$: Maintains a partition of disjoint sets over elements.

- **$\mathcal{P}$.makeset(x):** Given an element x, adds x to the data-structure and creates a singleton set that contains only this element. Returns a locator/handle for x in the data-structure.
- **$\mathcal{P}$.find(x):** Given a handle for an element x; find the set that contains x. Returns a representative/identifier for this set.
- **$\mathcal{P}$.union(x, y):** Given two elements x, and y that are currently in sets S_x and S_y, respectively, the function replaces S_x and S_y by $S_x \cup S_y$ and returns an identifier for the new set.

9 Union Find

Applications:

- Keep track of the connected components of a dynamic graph that changes due to insertion of nodes and edges.
- Kruskals Minimum Spanning Tree Algorithm

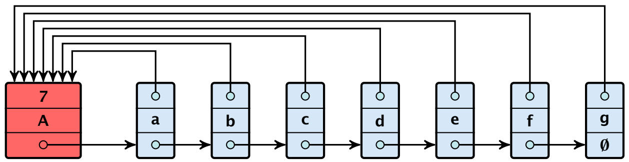
9 Union Find

Algorithm 20 Kruskal-MST($G = (V, E), w$)

$A \leftarrow \emptyset$;
for all $v \in V$ **do**
    v.set $\leftarrow \mathcal{P}$.makeset(v.label)
sort edges in non-decreasing order of weight w
for all $(u, v) \in E$ in non-decreasing order **do**
    if $\mathcal{P}$.find(u.set) $\neq \mathcal{P}$.find(v.set) **then**
        $A \leftarrow A \cup \{(u, v)\}$
        $\mathcal{P}$.union(u.set, v.set)

List Implementation

- The elements of a set are stored in a list; each node has a backward pointer to the head.
- The head of the list contains the identifier for the set and a field that stores the size of the set.



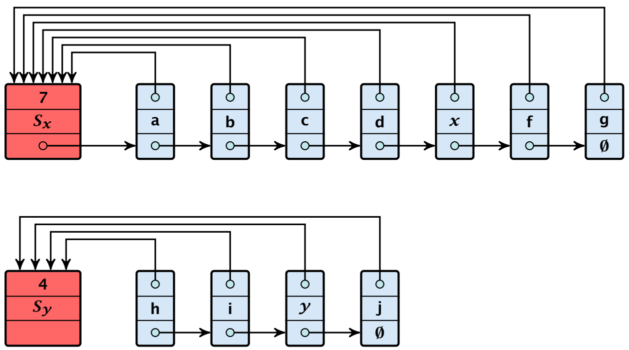
- $\text{makeset}(x)$ can be performed in constant time.
- $\text{find}(x)$ can be performed in constant time.

List Implementation

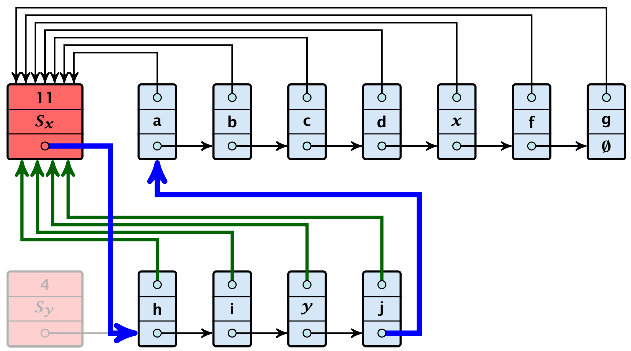
union(x, y)

- Determine sets S_x and S_y.
- Traverse the smaller list (say S_y), and change all backward pointers to the head of list S_x.
- Insert list S_y at the head of S_x.
- Adjust the size-field of list S_x.
- Time: $\min\{|S_x|, |S_y|\}$.

List Implementation



List Implementation



List Implementation

Running times:

- $\text{find}(x)$: constant
- $\text{makeset}(x)$: constant
- $\text{union}(x, y)$: $\mathcal{O}(n)$, where n denotes the number of elements contained in the set system.

List Implementation

Lemma 28

The list implementation for the ADT union find fulfills the following amortized time bounds:

- $\text{find}(x)$: $\mathcal{O}(1)$.
- $\text{makeset}(x)$: $\mathcal{O}(\log n)$.
- $\text{union}(x, y)$: $\mathcal{O}(1)$.

The Accounting Method for Amortized Time Bounds

- There is a bank account for every element in the data structure.
- Initially the balance on all accounts is zero.
- Whenever for an operation the amortized time bound exceeds the actual cost, the difference is credited to some bank accounts of elements involved.
- Whenever for an operation the actual cost exceeds the amortized time bound, the difference is charged to bank accounts of some of the elements involved.
- If we can find a charging scheme that guarantees that balances always stay positive the amortized time bounds are proven.

List Implementation

- For an operation whose actual cost exceeds the amortized cost we charge the excess to the elements involved.
- In total we will charge at most $\mathcal{O}(\log n)$ to an element (regardless of the request sequence).
- For each element a makeset operation occurs as the first operation involving this element.
- We inflate the amortized cost of the makeset-operation to $\Theta(\log n)$, i.e., at this point we fill the bank account of the element to $\Theta(\log n)$.
- Later operations charge the account but the balance never drops below zero.

List Implementation

makeset(x) : The actual cost is $\mathcal{O}(1)$. Due to the cost inflation the amortized cost is $\mathcal{O}(\log n)$.

find(x) : For this operation we define the amortized cost and the actual cost to be the same. Hence, this operation does not change any accounts. Cost: $\mathcal{O}(1)$.

union(x, y):

- If $S_x = S_y$ the cost is constant; no bank accounts change.
- Otw. the actual cost is $\mathcal{O}(\min\{|S_x|, |S_y|\})$.
- Assume wlog. that S_x is the smaller set; let c denote the hidden constant, i.e., the actual cost is at most $c \cdot |S_x|$.
- Charge c to every element in set S_x.

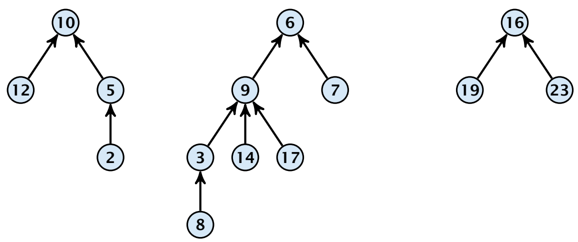
List Implementation

Lemma 29
An element is charged at most $\lfloor \log_2 n \rfloor$ times, where n is the total number of elements in the set system.

Proof.
Whenever an element x is charged the number of elements in x's set doubles. This can happen at most $\lfloor \log n \rfloor$ times. □

Implementation via Trees

- Maintain nodes of a set in a tree.
- The root of the tree is the label of the set.
- Only pointer to parent exists; we cannot list all elements of a given set.
- Example:



Set system $\{2, 5, 10, 12\}$, $\{3, 6, 7, 8, 9, 14, 17\}$, $\{16, 19, 23\}$.

Implementation via Trees

makeset(x)

- Create a singleton tree. Return pointer to the root.
- Time: $\mathcal{O}(1)$.

find(x)

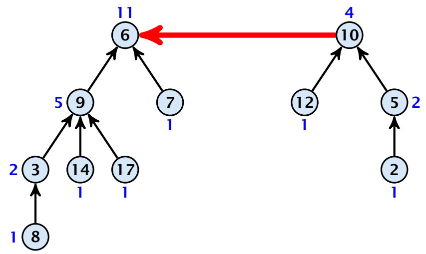
- Start at element x in the tree. Go upwards until you reach the root.
- Time: $\mathcal{O}(\text{level}(x))$, where $\text{level}(x)$ is the distance of element x to the root in its tree. Not constant.

Implementation via Trees

To support union we store the size of a tree in its root.

union(x, y)

- Perform $a \leftarrow \text{find}(x)$; $b \leftarrow \text{find}(y)$. Then: $\text{link}(a, b)$.
- $\text{link}(a, b)$ attaches the smaller tree as the child of the larger.
- In addition it updates the size-field of the new root.



- Time: constant for $\text{link}(a, b)$ plus two find-operations.

Implementation via Trees

Lemma 30
The running time (non-amortized!!!) for $\text{find}(x)$ *is* $\mathcal{O}(\log n)$.

Proof.

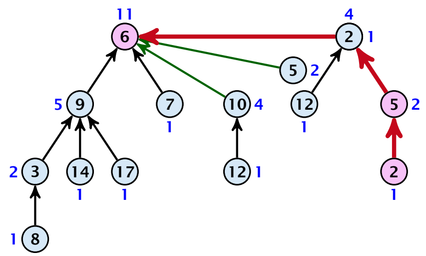
- When we attach a tree with root c to become a child of a tree with root p, then $\text{size}(p) \geq 2\,\text{size}(c)$, where size denotes the value of the size-field right after the operation.
- After that the value of $\text{size}(c)$ stays fixed, while the value of $\text{size}(p)$ may still increase.
- Hence, at any point in time a tree fulfills $\text{size}(p) \geq 2\,\text{size}(c)$, for any pair of nodes (p, c), where p is a parent of c.

□

Path Compression

find(x):

- Go upward until you find the root.
- Re-attach all visited nodes as children of the root.
- Speeds up successive find-operations.



- Note that the size-fields now only give an upper bound on the size of a sub-tree.

Asymptotically the cost for a find-operation does not increase due to the path compression heuristic.

However, for a worst-case analysis there is no improvement on the running time. It can still happen that a find-operation takes time $\mathcal{O}(\log n)$.

Amortized Analysis

Definitions:

- $\text{size}(v)$: the number of nodes that were in the sub-tree rooted at v when v became the child of another node (or the number of nodes if v is the root).
- $\text{rank}(v)$: $\lfloor \log(\text{size}(v)) \rfloor$.
- $\Longrightarrow \text{size}(v) \geq 2^{\text{rank}(v)}$.

Lemma 31

The rank of a parent must be strictly larger than the rank of a child.

Amortized Analysis

Lemma 32

There are at most $n/2^s$ nodes of rank s.

Proof.

- Let's say a node v sees the rank s node x if v is in x's sub-tree at the time that x becomes a child.
- A node v sees at most one node of rank s during the running time of the algorithm.
- This holds because the rank-sequence of the roots of the different trees that contains v during the running time of the algorithm is a strictly increasing sequence.
- Hence, every node *sees* at most one rank s node, but every rank s node is seen by at least 2^s different nodes. □

Amortized Analysis

We define

$$\text{tow}(i) := \begin{cases} 1 & \text{if } i = 0 \\ 2^{\text{tow}(i-1)} & \text{otw.} \end{cases} \qquad \text{tow}(i) = 2^{2^{2^{2^{2}}}} \Big\} i \text{ times}$$

and

$$\log^*(n) := \min\{i \mid \text{tow}(i) \geq n\} \ .$$

Theorem 33

Union find with path compression fulfills the following amortized running times:

- $\text{makeset}(x) : \mathcal{O}(\log^*(n))$
- $\text{find}(x) : \mathcal{O}(\log^*(n))$
- $\text{union}(x, y) : \mathcal{O}(\log^*(n))$

Amortized Analysis

In the following we assume $n \geq 3$.

rank-group:

- A node with rank $\text{rank}(v)$ is in rank group $\log^*(\text{rank}(v))$.
- The rank-group $g = 0$ contains only nodes with rank 0 or rank 1.
- A rank group $g \geq 1$ contains ranks $\text{tow}(g-1) + 1, \ldots, \text{tow}(g)$.
- The maximum non-empty rank group is $\log^*(\lfloor \log n \rfloor) \leq \log^*(n) - 1$ (which holds for $n \geq 3$).
- Hence, the total number of rank-groups is at most $\log^* n$.

Amortized Analysis

Accounting Scheme:

- create an account for every find-operation
- create an account for every node v

The cost for a find-operation is equal to the length of the path traversed. We charge the cost for going from v to $\text{parent}[v]$ as follows:

- If $\text{parent}[v]$ is the root we charge the cost to the find-account.
- If the group-number of $\text{rank}(v)$ is the same as that of $\text{rank}(\text{parent}[v])$ (before starting path compression) we charge the cost to the node-account of v.
- Otherwise we charge the cost to the find-account.

Observations:

- A find-account is charged at most $\log^*(n)$ times (once for the root and at most $\log^*(n) - 1$ times when increasing the rank-group).
- After a node v is charged its parent-edge is re-assigned. The rank of the parent strictly increases.
- After some charges to v the parent will be in a larger rank-group. $\Rightarrow$ v will never be charged again.
- The total charge made to a node in rank-group g is at most $\text{tow}(g) - \text{tow}(g-1) \le \text{tow}(g)$.

What is the total charge made to nodes?

- The total charge is at most

$$\sum_g n(g) \cdot \text{tow}(g) \ ,$$

where $n(g)$ is the number of nodes in group g.

For $g \ge 1$ we have

$$\begin{aligned} n(g) &\le \sum_{s=\text{tow}(g-1)+1}^{\text{tow}(g)} \frac{n}{2^s} = \frac{n}{2^{\text{tow}(g-1)+1}} \sum_{s=0}^{\text{tow}(g)-\text{tow}(g-1)-1} \frac{1}{2^s} \\ &\le \frac{n}{2^{\text{tow}(g-1)+1}} \sum_{s=0}^{\infty} \frac{1}{2^s} \le \frac{n}{2^{\text{tow}(g-1)+1}} \cdot 2 \\ &\le \frac{n}{2^{\text{tow}(g-1)}} = \frac{n}{\text{tow}(g)} \ . \end{aligned}$$

Hence,

$$\sum_g n(g)\,\text{tow}(g) \le n(0)\,\text{tow}(0) + \sum_{g\ge1} n(g)\,\text{tow}(g) \le n\log^*(n)$$

Amortized Analysis

Without loss of generality we can assume that all makeset-operations occur at the start.

This means if we inflate the cost of makeset to $\log^* n$ and add this to the node account of v then the balances of all node accounts will sum up to a positive value (this is sufficient to obtain an amortized bound).

The analysis is not tight. In fact it has been shown that the amortized time for the union-find data structure with path compression is $\mathcal{O}(\alpha(m,n))$, where $\alpha(m,n)$ is the inverse Ackermann function which grows a lot lot slower than $\log^* n$. (Here, we consider the average running time of m operations on at most n elements).

There is also a lower bound of $\Omega(\alpha(m,n))$.

$$A(x,y) = \begin{cases} y+1 & \text{if } x = 0 \\ A(x-1,1) & \text{if } y = 0 \\ A(x-1, A(x,y-1)) & \text{otw.} \end{cases}$$

$$\alpha(m,n) = \min\{i \geq 1 : A(i, \lfloor m/n \rfloor) \geq \log n\}$$

- $A(0,y) = y+1$
- $A(1,y) = y+2$
- $A(2,y) = 2y+3$
- $A(3,y) = 2^{y+3} - 3$
- $A(4,y) = \underbrace{2^{2^{2^{2}}}}_{y+3 \text{ times}} - 3$

10 van Emde Boas Trees

Dynamic Set Data Structure S:

- $S.\mathrm{insert}(x)$
- $S.\mathrm{delete}(x)$
- $S.\mathrm{search}(x)$
- $S.\mathrm{min}()$
- $S.\mathrm{max}()$
- $S.\mathrm{succ}(x)$
- $S.\mathrm{pred}(x)$

10 van Emde Boas Trees

For this chapter we ignore the problem of storing satellite data:

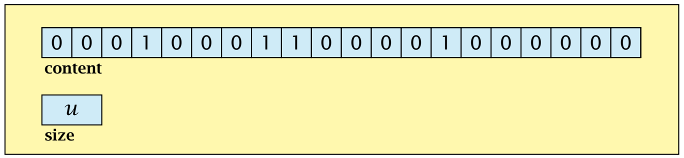
- **S. insert(x):** Inserts x into S.
- **S. delete(x):** Deletes x from S. Usually assumes that $x \in S$.
- **S. member(x):** Returns 1 if $x \in S$ and 0 otw.
- **S. min():** Returns the value of the minimum element in S.
- **S. max():** Returns the value of the maximum element in S.
- **S. succ(x):** Returns successor of x in S. Returns null if x is maximum or larger than any element in S. Note that x needs not to be in S.
- **S. pred(x):** Returns the predecessor of x in S. Returns null if x is minimum or smaller than any element in S. Note that x needs not to be in S.

10 van Emde Boas Trees

Can we improve the existing algorithms when the keys are from a restricted set?

In the following we assume that the keys are from $\{0, 1, \ldots, u - 1\}$, where u denotes the size of the universe.

Implementation 1: Array



one array of u bits

Use an array that encodes the indicator function of the dynamic set.

Implementation 1: Array

Algorithm 21 array.insert(x)

content[x] $\leftarrow 1$;

Algorithm 22 array.delete(x)

content[x] $\leftarrow 0$;

Algorithm 22 array.member(x)

return content[x];

- Note that we assume that x is valid, i.e., it falls within the array boundaries.
- Obviously(?) the running time is constant.

Implementation 1: Array

Algorithm 24 array.max()

for ($i = \text{size} - 1$; $i \geq 0$; i--) **do**
  if content[i] $= 1$ **then return** i;
return null;

Algorithm 25 array.min()

for ($i = 0$; $i < \text{size}$; i++) **do**
  if content[i] $= 1$ **then return** i;
return null;

- Running time is $\mathcal{O}(u)$ in the worst case.

Implementation 1: Array

Algorithm 26 array.succ(x)

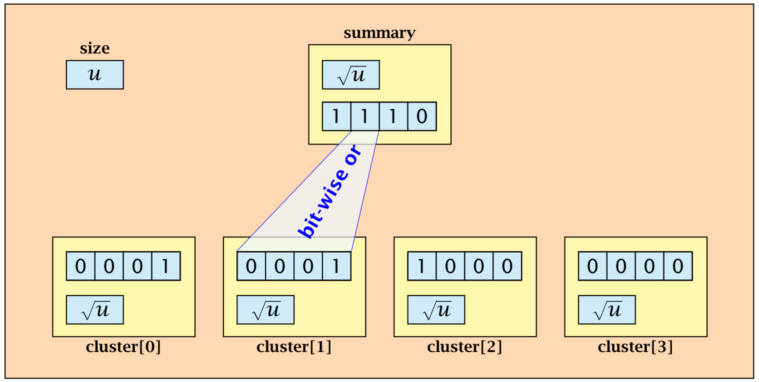
for ($i = x + 1$; $i < \text{size}$; i++) **do**
  if content[i] $= 1$ **then return** i;
return null;

Algorithm 27 array.pred(x)

for ($i = x - 1$; $i \geq 0$; i--) **do**
  if content[i] $= 1$ **then return** i;
return null;

- Running time is $\mathcal{O}(u)$ in the worst case.

Implementation 2: Summary Array



- $\sqrt{u}$ cluster-arrays of $\sqrt{u}$ bits.
- One summary-array of $\sqrt{u}$ bits. The i-th bit in the summary array stores the bit-wise or of the bits in the i-th cluster.

Implementation 2: Summary Array

The bit for a key x is contained in cluster number $\left\lfloor \frac{x}{\sqrt{u}} \right\rfloor$.

Within the cluster-array the bit is at position $x \bmod \sqrt{u}$.

For simplicity we assume that $u = 2^{2k}$ for some $k \geq 1$. Then we can compute the cluster-number for an entry x as $\text{high}(x)$ (the upper half of the dual representation of x) and the position of x within its cluster as $\text{low}(x)$ (the lower half of the dual representation).

Implementation 2: Summary Array

Algorithm 28 member(x)

```
return cluster[high(x)].member(low(x));
```

Algorithm 29 insert(x)

```
cluster[high(x)].insert(low(x));
summary.insert(high(x));
```

- The running times are constant, because the corresponding array-functions have constant running times.

Implementation 2: Summary Array

Algorithm 30 delete(x)

```
cluster[high(x)].delete(low(x));
if cluster[high(x)].min() = null then
    summary.delete(high(x));
```

- The running time is dominated by the cost of a minimum computation on an array of size $\sqrt{u}$. Hence, $\mathcal{O}(\sqrt{u})$.

Implementation 2: Summary Array

Algorithm 31 max()

```
maxcluster ← summary.max();
if maxcluster = null return null;
offs ← cluster[maxcluster].max()
return maxcluster ∘ offs;
```

Algorithm 32 min()

```
mincluster ← summary.min();
if mincluster = null return null;
offs ← cluster[mincluster].min();
return mincluster ∘ offs;
```

The operator $\circ$ stands for the concatenation of two bitstrings.
This means if
$x = 0111_2$ and
$y = 0001_2$ then
$x \circ y = 01110001_2$.

- Running time is roughly $2\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 2: Summary Array

Algorithm 33 succ(x)

```
m ← cluster[high(x)].succ(low(x))
if m ≠ null then return high(x) ∘ m;
succcluster ← summary.succ(high(x));
if succcluster ≠ null then
    offs ← cluster[succcluster].min();
    return succcluster ∘ offs;
return null;
```

- Running time is roughly $3\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 2: Summary Array

Algorithm 34 pred(x)

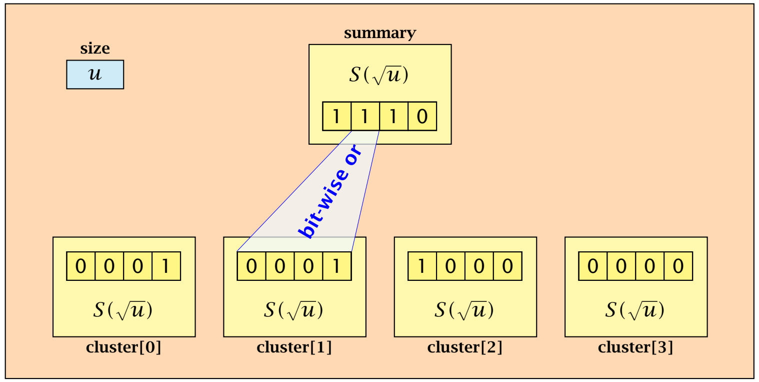
```
m ← cluster[high(x)].pred(low(x))
if m ≠ null then return high(x) ∘ m;
predcluster ← summary.pred(high(x));
if predcluster ≠ null then
    offs ← cluster[predcluster].max();
    return predcluster ∘ offs;
return null;
```

- Running time is roughly $3\sqrt{u} = \mathcal{O}(\sqrt{u})$ in the worst case.

Implementation 3: Recursion

Instead of using sub-arrays, we build a recursive data-structure.

$S(u)$ is a dynamic set data-structure representing u bits:



Implementation 3: Recursion

We assume that $u = 2^{2^k}$ for some k.

The data-structure $S(2)$ is defined as an array of 2-bits (end of the recursion).

Implementation 3: Recursion

The code from Implementation 2 can be used unchanged. We only need to redo the analysis of the running time.

Note that in the code we do not need to specifically address the non-recursive case. This is achieved by the fact that an $S(4)$ will contain $S(2)$'s as sub-datastructures, which are arrays. Hence, a call like $\text{cluster}[1].\min()$ from within the data-structure $S(4)$ is not a recursive call as it will call the function $\text{array}.\min()$.

This means that the non-recursive case is been dealt with while initializing the data-structure.

Implementation 3: Recursion

Algorithm 35 member(x)

```
return cluster[high(x)].member(low(x));
```

- $T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 36 insert(x)

```
cluster[high(x)].insert(low(x));
summary.insert(high(x));
```

- $T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 37 delete(x)

```
cluster[high(x)].delete(low(x));
if cluster[high(x)].min() = null then
        summary.delete(high(x));
```

- $T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\min}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 38 min()

```
mincluster ← summary.min();
if mincluster = null return null;
offs ← cluster[mincluster].min();
return mincluster ∘ offs;
```

- $T_{\min}(u) = 2T_{\min}(\sqrt{u}) + 1$.

Implementation 3: Recursion

Algorithm 39 succ(x)

```
m ← cluster[high(x)].succ(low(x))
if m ≠ null then return high(x) ∘ m;
succcluster ← summary.succ(high(x));
if succcluster ≠ null then
    offs ← cluster[succcluster].min();
    return succcluster ∘ offs;
return null;
```

- $T_{\text{succ}}(u) = 2T_{\text{succ}}(\sqrt{u}) + T_{\text{min}}(\sqrt{u}) + 1$.

Implementation 3: Recursion

$\boldsymbol{T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1}$**:**

Set $\ell := \log u$ and $X(\ell) := T_{\text{mem}}(2^\ell)$.Then

$$\begin{aligned} X(\ell) = T_{\text{mem}}(2^\ell) &= T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1 \\ &= T_{\text{mem}}(2^{\frac{\ell}{2}}) + 1 = X(\tfrac{\ell}{2}) + 1 \ . \end{aligned}$$

Using Master theorem gives $X(\ell) = \mathcal{O}(\log \ell)$, and hence $T_{\text{mem}}(u) = \mathcal{O}(\log\log u)$.

Implementation 3: Recursion

$\boldsymbol{T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1}$**.**

Set $\ell := \log u$ and $X(\ell) := T_{\text{ins}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{ins}}(2^\ell) &= T_{\text{ins}}(u) = 2T_{\text{ins}}(\sqrt{u}) + 1 \\ &= 2T_{\text{ins}}(2^{\frac{\ell}{2}}) + 1 = 2X(\tfrac{\ell}{2}) + 1 \ . \end{aligned}$$

Using Master theorem gives $X(\ell) = \mathcal{O}(\ell)$, and hence $T_{\text{ins}}(u) = \mathcal{O}(\log u)$.

The same holds for $T_{\text{max}}(u)$ and $T_{\text{min}}(u)$.

Implementation 3: Recursion

$\boldsymbol{T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + T_{\text{min}}(\sqrt{u}) + 1 \le 2T_{\text{del}}(\sqrt{u}) + c\log(u)}$**.**

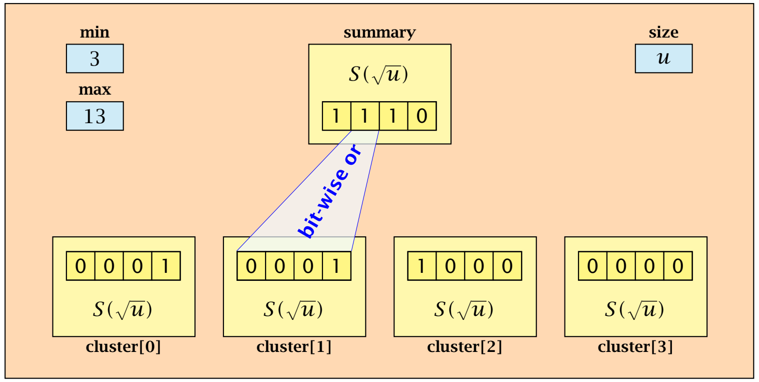
Set $\ell := \log u$ and $X(\ell) := T_{\text{del}}(2^\ell)$. Then

$$\begin{aligned} X(\ell) = T_{\text{del}}(2^\ell) &= T_{\text{del}}(u) = 2T_{\text{del}}(\sqrt{u}) + c\log u \\ &= 2T_{\text{del}}(2^{\frac{\ell}{2}}) + c\ell = 2X(\tfrac{\ell}{2}) + c\ell \ . \end{aligned}$$

Using Master theorem gives $X(\ell) = \Theta(\ell \log \ell)$, and hence $T_{\text{del}}(u) = \mathcal{O}(\log u \log\log u)$.

The same holds for $T_{\text{pred}}(u)$ and $T_{\text{succ}}(u)$.

Implementation 4: van Emde Boas Trees



- The bit referenced by min is not set within sub-datastructures.
- The bit referenced by max is set within sub-datastructures (if $\max \neq \min$).

Implementation 4: van Emde Boas Trees

Advantages of having max/min pointers:

- Recursive calls for $\min$ and $\max$ are constant time.
- $\min = \text{null}$ means that the data-structure is empty.
- $\min = \max \neq \text{null}$ means that the data-structure contains exactly one element.
- We can insert into an empty datastructure in constant time by only setting $\min = \max = x$.
- We can delete from a data-structure that just contains one element in constant time by setting $\min = \max = \text{null}$.

Implementation 4: van Emde Boas Trees

Algorithm 40 max()

1: **return** max;

Algorithm 41 min()

1: **return** min;

- Constant time.

Implementation 4: van Emde Boas Trees

Algorithm 42 member(x)

1: **if** $x = \min$ **then return** 1; // TRUE
2: **return** cluster[high(x)].member(low(x));

- $T_{\text{mem}}(u) = T_{\text{mem}}(\sqrt{u}) + 1 \Longrightarrow T(u) = \mathcal{O}(\log\log u)$.

Implementation 4: van Emde Boas Trees

Algorithm 43 succ(x)

1: **if** min $\neq$ null $\wedge$ $x <$ min **then return** min;
2: *maxincluster* $\leftarrow$ cluster[high(x)].max();
3: **if** *maxincluster* $\neq$ null $\wedge$ low(x) $<$ *maxincluster* **then**
4: *offs* $\leftarrow$ cluster[high(x)].succ(low(x));
5: **return** high(x) $\circ$ *offs*;
6: **else**
7: *succcluster* $\leftarrow$ summary.succ(high(x));
8: **if** *succcluster* = null **then return** null;
9: *offs* $\leftarrow$ cluster[*succcluster*].min();
10: **return** *succcluster* $\circ$ *offs*;

- $T_{\text{succ}}(u) = T_{\text{succ}}(\sqrt{u}) + 1 \Longrightarrow T_{\text{succ}}(u) = \mathcal{O}(\log\log u)$.

Implementation 4: van Emde Boas Trees

Algorithm 36 insert(x)

1: **if** min = null **then**
2: min = x; max = x;
3: **else**
4: **if** $x <$ min **then** exchange x and min;
5: **if** cluster[high(x)].min = null; **then**
6: summary.insert(high(x));
7: cluster[high(x)].insert(low(x));
8: **else**
9: cluster[high(x)].insert(low(x));
10: **if** $x >$ max **then** max = x;

- $T_{\text{ins}}(u) = T_{\text{ins}}(\sqrt{u}) + 1 \Longrightarrow T_{\text{ins}}(u) = \mathcal{O}(\log\log u)$.

Implementation 4: van Emde Boas Trees

Note that the recusive call in Line 7 takes constant time as the if-condition in Line 5 ensures that we are inserting in an empty sub-tree.

The only non-constant recursive calls are the call in Line 6 and in Line 9. These are mutually exclusive, i.e., only one of these calls will actually occur.

From this we get that $T_{\text{ins}}(u) = T_{\text{ins}}(\sqrt{u}) + 1$.

Implementation 4: van Emde Boas Trees

- **Assumes that x is contained in the structure.**

Algorithm 36 delete(x)

1: **if** min = max **then**
2: min = null; max = null;
3: **else**
4: **if** x = min **then** — find new minimum
5: *firstcluster* $\leftarrow$ summary.min();
6: *offs* $\leftarrow$ cluster[*firstcluster*].min();
7: $x \leftarrow$ *firstcluster* $\circ$ *offs*;
8: min $\leftarrow x$;
9: cluster[high(x)].delete(low(x)); — delete

continued...

Implementation 4: van Emde Boas Trees

Algorithm 35 delete(x)

...continued — fix maximum

```
10: if cluster[high(x)].min() = null then
11:     summary.delete(high(x));
12:     if x = max then
13:         summax ← summary.max();
14:         if summax = null then max ← min;
15:         else
16:             offs ← cluster[summax].max();
17:             max ← summax ∘ offs
18: else
19:     if x = max then
20:         offs ← cluster[high(x)].max();
21:         max ← high(x) ∘ offs;
```


Implementation 4: van Emde Boas Trees

Note that only one of the possible recusive calls in Line 9 and Line 11 in the deletion-algorithm may take non-constant time.

To see this observe that the call in Line 11 only occurs if the cluster where x was deleted is now empty. But this means that the call in Line 9 deleted the last element in $\text{cluster}[\text{high}(x)]$. Such a call only takes constant time.

Hence, we get a recurrence of the form

$$T_{\text{del}}(u) = T_{\text{del}}(\sqrt{u}) + c \ .$$

This gives $T_{\text{del}}(u) = \mathcal{O}(\log\log u)$.

10 van Emde Boas Trees

Space requirements:

- The space requirement fulfills the recurrence

$$S(u) = (\sqrt{u} + 1)S(\sqrt{u}) + \mathcal{O}(\sqrt{u}) \ .$$

- Note that we cannot solve this recurrence by the Master theorem as the branching factor is not constant.
- One can show by induction that the space requirement is $S(u) = \mathcal{O}(u)$. Exercise.

- Let the "real" recurrence relation be

$$S(k^2) = (k+1)S(k) + c_1 \cdot k;\ S(4) = c_2$$

- Replacing $S(k)$ by $R(k) := S(k)/c_2$ gives the recurrence

$$R(k^2) = (k+1)R(k) + ck;\ R(4) = 1$$

  where $c = c_1/c_2 < 1$.
- Now, we show $R(k) \leq k - 2$ for squares $k \geq 4$.
  - Obviously, this holds for $k = 4$.
  - For $k = \ell^2 > 4$ with ℓ integral we have

$$\begin{aligned} R(k) &= (1+\ell)R(\ell) + c\ell \\ &\leq (1+\ell)(\ell-2) + \ell \leq k - 2 \end{aligned}$$

  - This shows that $R(k)$ and, hence, $S(k)$ grows linearly.