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 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

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 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

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

- **S.build($x_1, \dots, x_n$):** Creates a data-structure that contains just the elements $x_1, \dots, x_n$.
- **S.insert(x):** Adds element x to the data-structure.
- **element S.minimum():** Returns an element $x \in S$ with minimum key-value 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

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

- **S.build$(x_1, \dots, x_n)$:** Creates a data-structure that contains just the elements $x_1, \dots, x_n$.
- **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

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

- **S.build($x_1, \dots, x_n$):** Creates a data-structure that contains just the elements $x_1, \dots, x_n$.
- **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

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

- **S.build($x_1, \dots, x_n$):** Creates a data-structure that contains just the elements $x_1, \dots, x_n$.
- **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

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.

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.

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.

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 $v \in V \setminus \{s\}$ **do**
    $v.\mathrm{key} \leftarrow \infty$;
    $h_v \leftarrow S$.insert(v);
$s.\mathrm{key} \leftarrow 0$; S.insert(s);
while S.is-empty$()$ $= \mathrm{false}$ **do**
    $v \leftarrow S$.delete-min$()$;
    for all $x \in V$ s.t. $(v, x) \in E$ **do**
        if $x.\mathrm{key} > v.\mathrm{key} + d(v, x)$ **then**
            S.decrease-key$(h_x, v.\mathrm{key} + d(v, x))$;
            $x.\mathrm{key} \leftarrow v.\mathrm{key} + d(v, x)$;

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?

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.

The standard version of binary heaps is not addressable, and hence does not support a delete operation.

Fibonacci heaps only give an amortized guarantee.

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.

The standard version of binary heaps is not addressable, and hence does not support a delete operation.

Fibonacci heaps only give an amortized guarantee.

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.

The standard version of binary heaps is not addressable, and hence does not support a delete operation.

Fibonacci heaps only give an amortized guarantee.

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.

The standard version of binary heaps is not addressable, and hence does not support a delete operation.

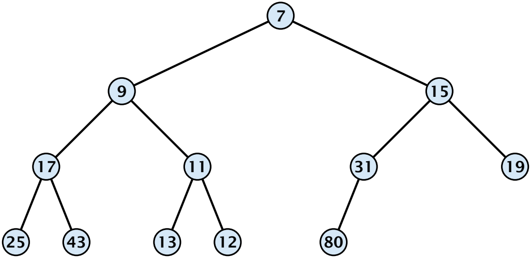
Fibonacci heaps only give an amortized guarantee.

8 Priority Queues

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

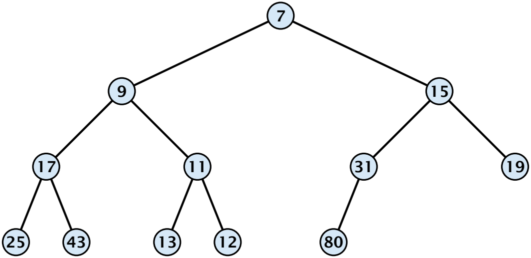
Using Fibonacci Heaps, Prim and Dijkstra run in time $\mathcal{O}(|V|\log|V| + |E|)$.

8.1 Binary Heaps



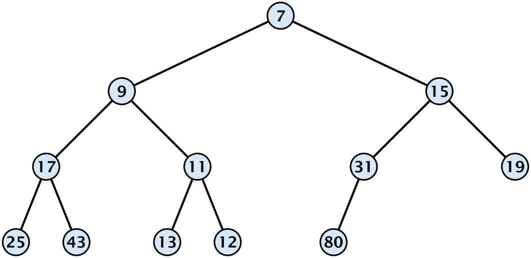
8.1 Binary Heaps

- Nearly complete binary tree; only the last level is not full, and this one is filled from left to right.



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 $\mathcal{O}(1)$.

Binary Heaps

Operations:

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

Binary Heaps

Operations:

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

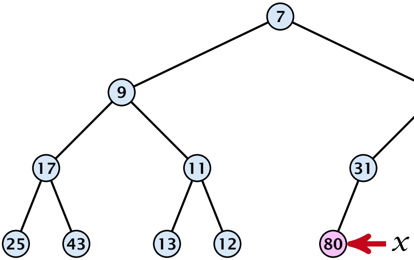
8.1 Binary Heaps

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



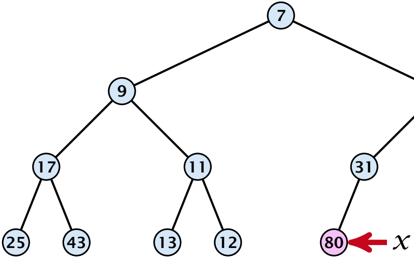
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the 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



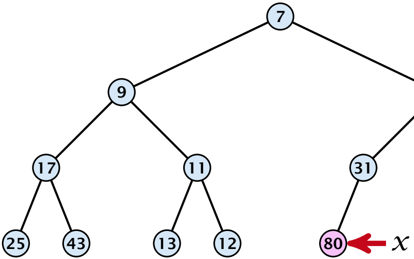
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the 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



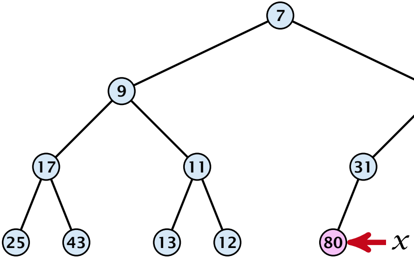
8.1 Binary Heaps

Maintain a pointer to the last element x.

- We can compute the 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



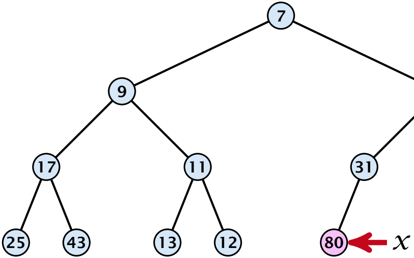
8.1 Binary Heaps

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;



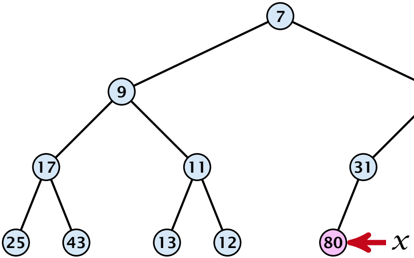
8.1 Binary Heaps

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;



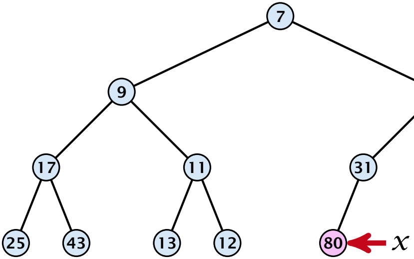
8.1 Binary Heaps

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;



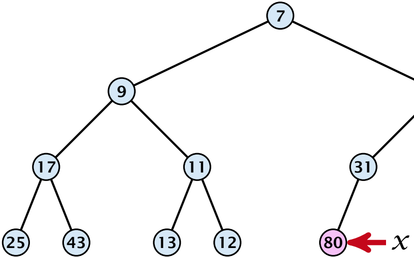
8.1 Binary Heaps

Maintain a pointer to the last element x.

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

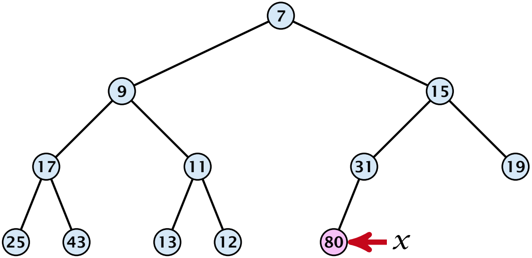
  go up until the last edge used was a left edge.
  go right; go left until you reach a null-pointer.

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



Insert

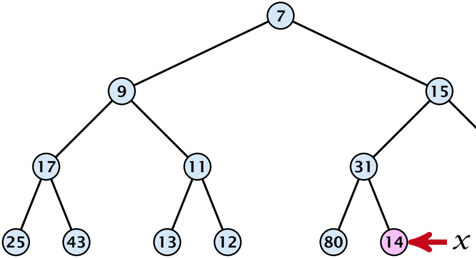
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



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.

Insert

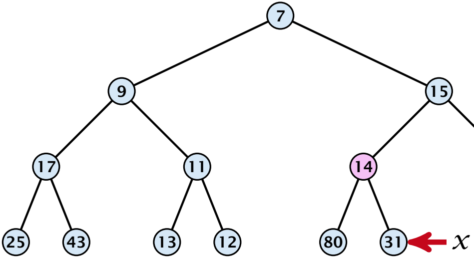
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



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.

Insert

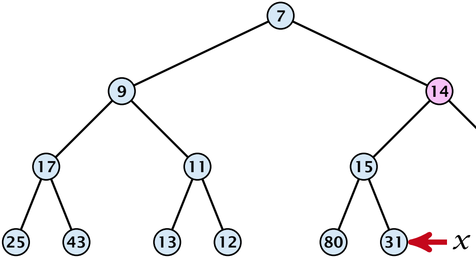
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



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.

Insert

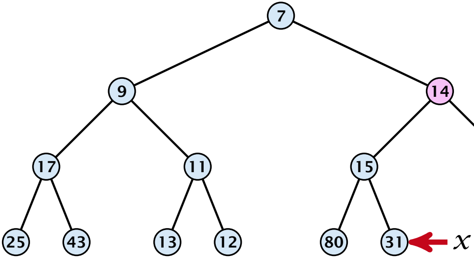
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



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.

Insert

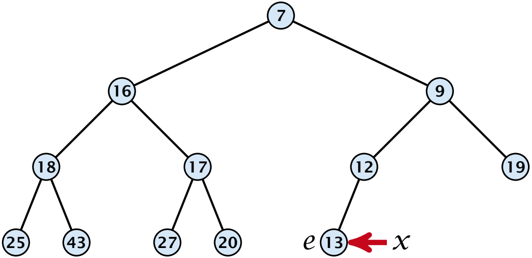
1. Insert element at successor of x.
2. Exchange with parent until heap property is fulfilled.



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

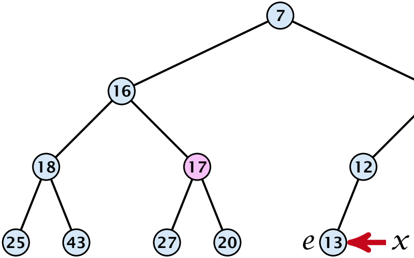
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



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

Delete

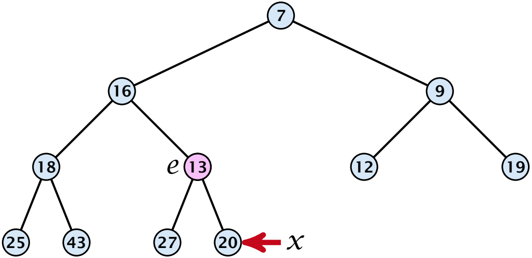
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



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

Delete

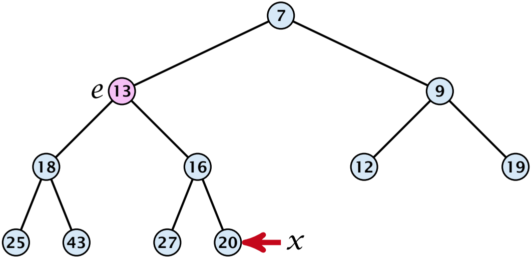
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



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

Delete

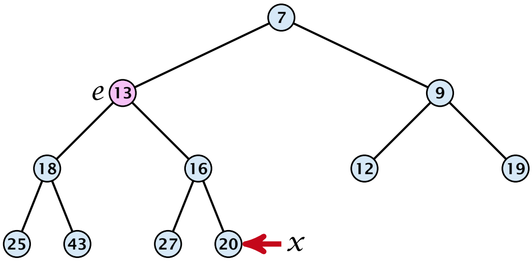
1. Exchange the element to be deleted with the element e pointed to by x.
2. Restore the heap-property for the element e.



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

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).

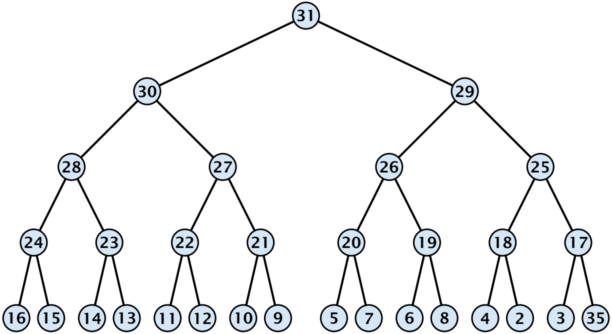
Binary Heaps

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

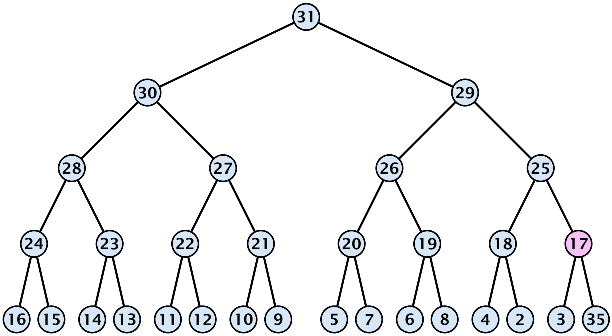
We can build a heap in linear time:



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

Build Heap

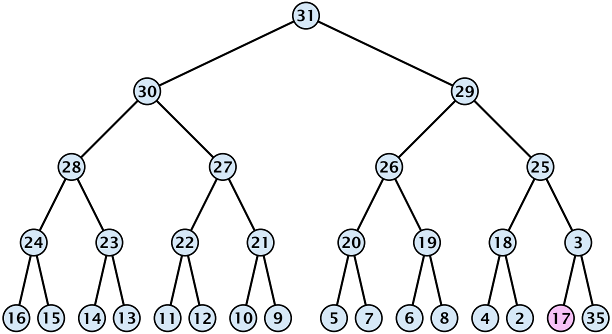
We can build a heap in linear time:



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

Build Heap

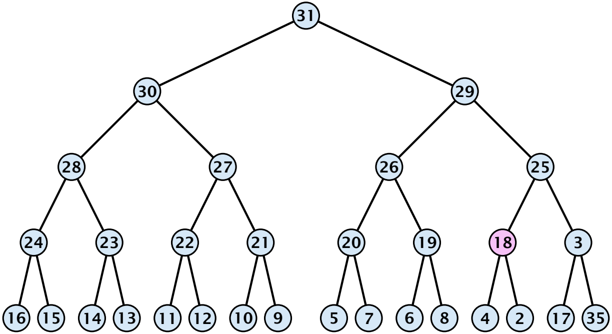
We can build a heap in linear time:



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

Build Heap

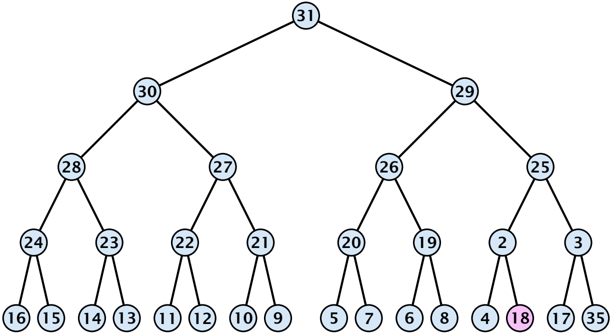
We can build a heap in linear time:



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

Build Heap

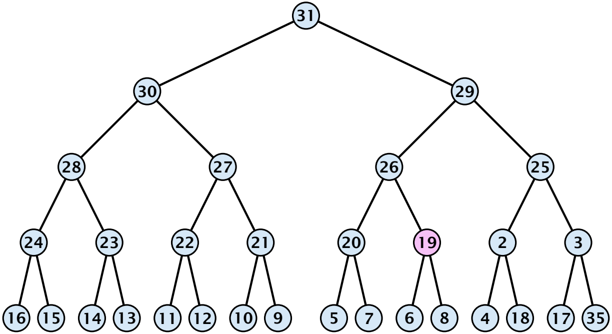
We can build a heap in linear time:



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

Build Heap

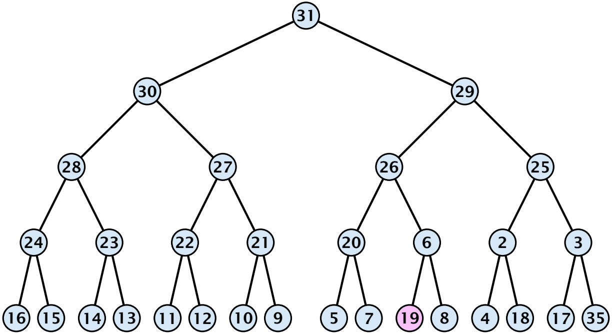
We can build a heap in linear time:



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

Build Heap

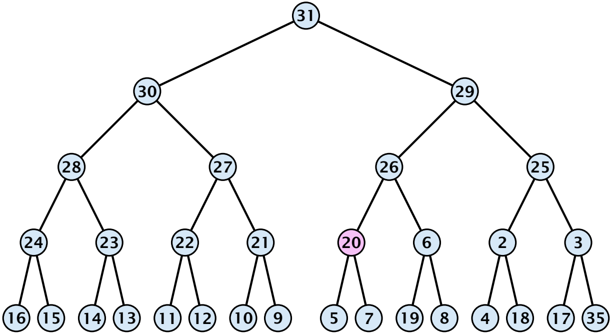
We can build a heap in linear time:



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

Build Heap

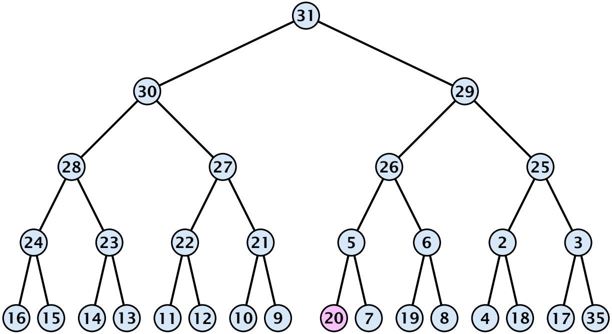
We can build a heap in linear time:



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

Build Heap

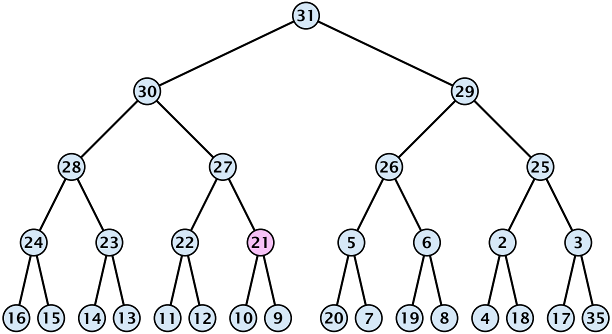
We can build a heap in linear time:



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

Build Heap

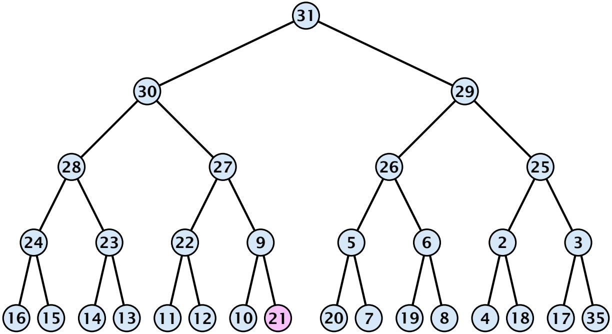
We can build a heap in linear time:



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

Build Heap

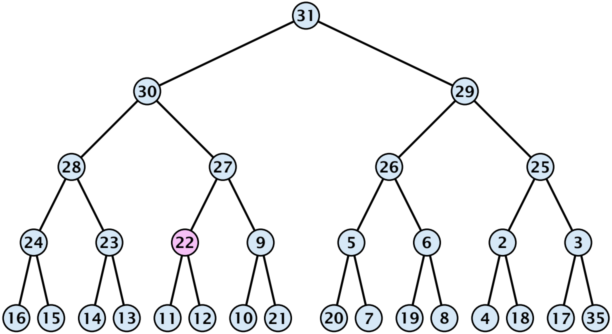
We can build a heap in linear time:



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

Build Heap

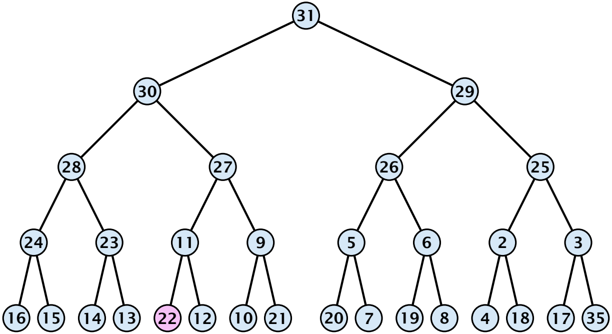
We can build a heap in linear time:



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

Build Heap

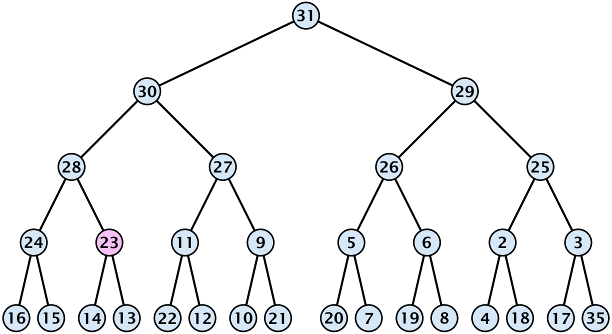
We can build a heap in linear time:



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

Build Heap

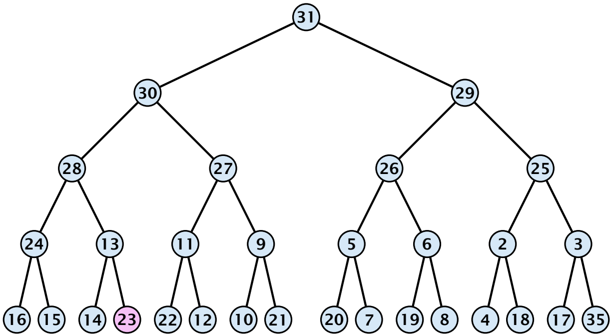
We can build a heap in linear time:



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

Build Heap

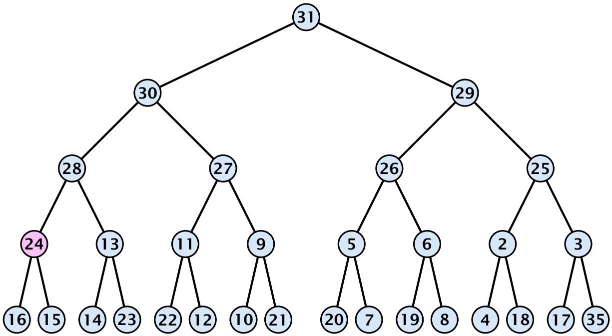
We can build a heap in linear time:



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

Build Heap

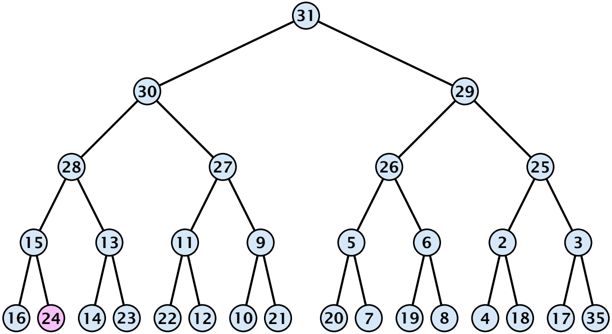
We can build a heap in linear time:



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

Build Heap

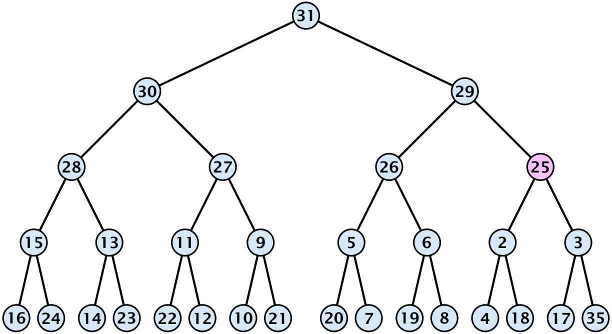
We can build a heap in linear time:



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

Build Heap

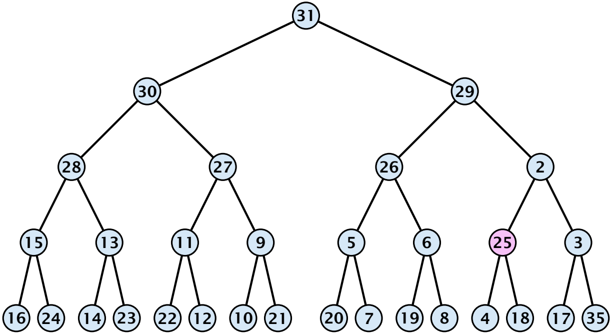
We can build a heap in linear time:



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

Build Heap

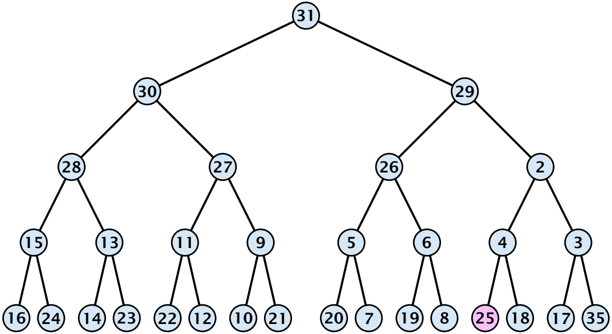
We can build a heap in linear time:



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

Build Heap

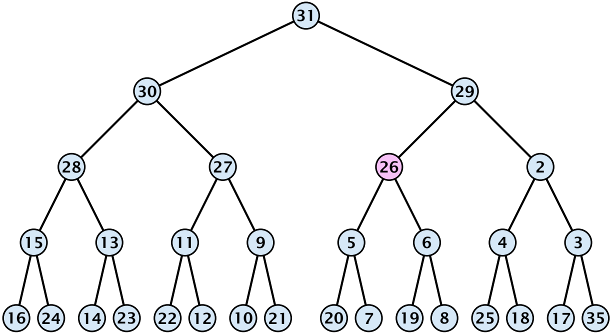
We can build a heap in linear time:



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

Build Heap

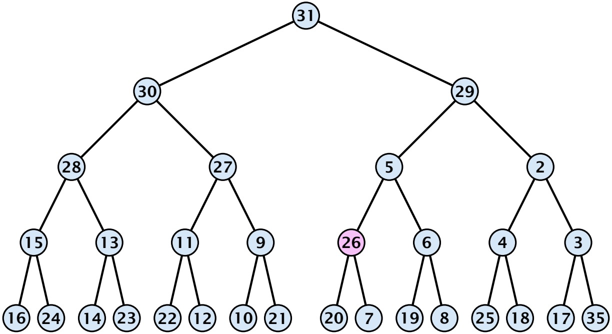
We can build a heap in linear time:



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

Build Heap

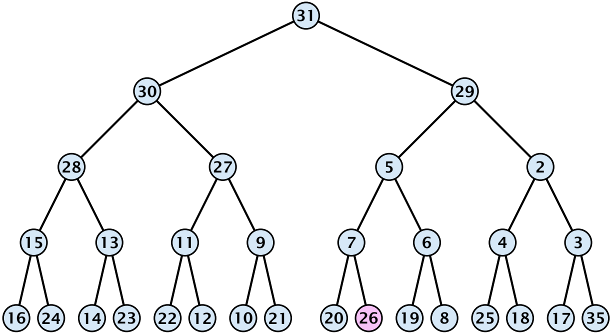
We can build a heap in linear time:



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

Build Heap

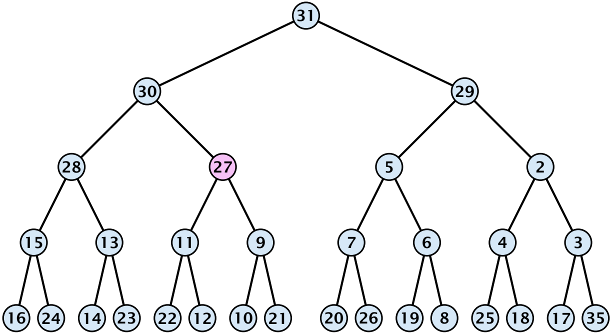
We can build a heap in linear time:



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

Build Heap

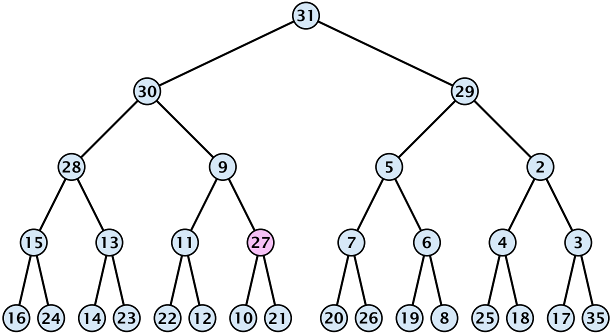
We can build a heap in linear time:



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

Build Heap

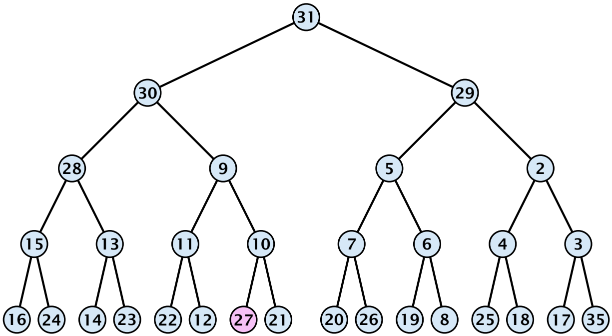
We can build a heap in linear time:



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

Build Heap

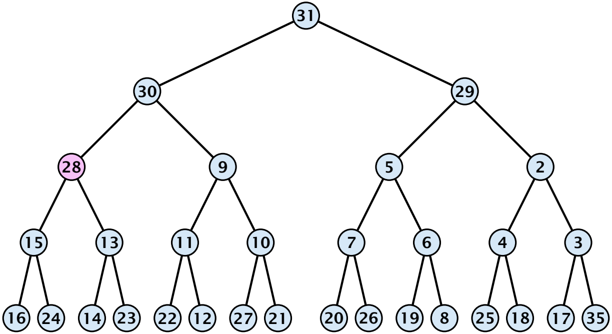
We can build a heap in linear time:



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

Build Heap

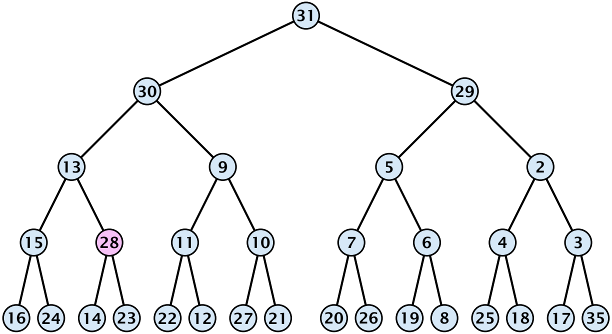
We can build a heap in linear time:



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

Build Heap

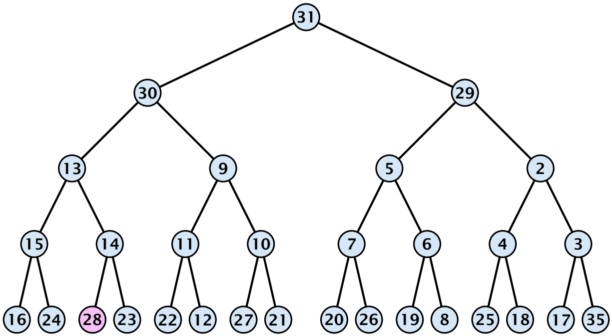
We can build a heap in linear time:



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

Build Heap

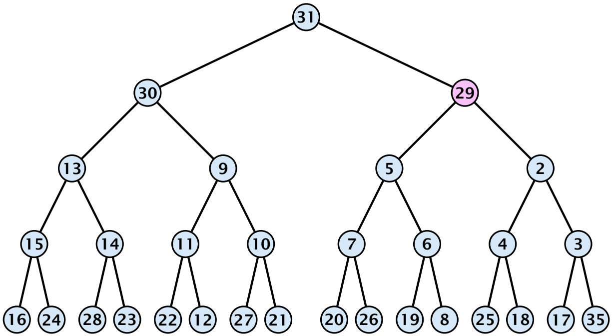
We can build a heap in linear time:



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

Build Heap

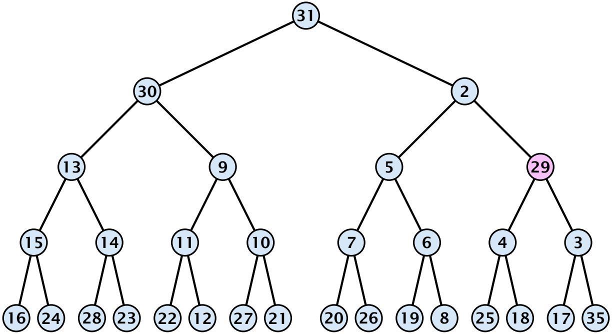
We can build a heap in linear time:



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

Build Heap

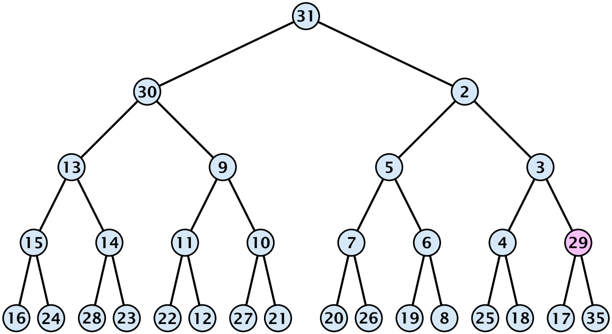
We can build a heap in linear time:



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

Build Heap

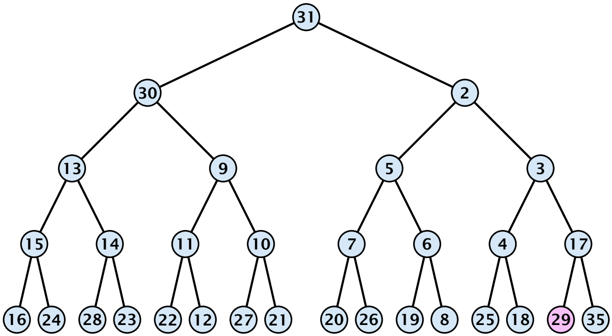
We can build a heap in linear time:



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

Build Heap

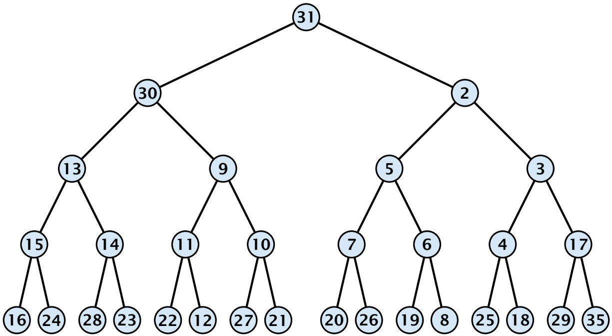
We can build a heap in linear time:



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

Build Heap

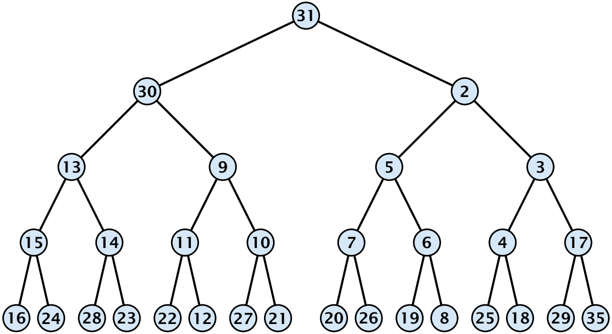
We can build a heap in linear time:



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

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 $A[0,\ldots,n-1]$ 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.

Binary Heaps

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

Binary Heaps

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

Binary Heaps

The standard implementation of binary heaps is via arrays. Let $A[0,\ldots,n-1]$ 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 $2i+2$.

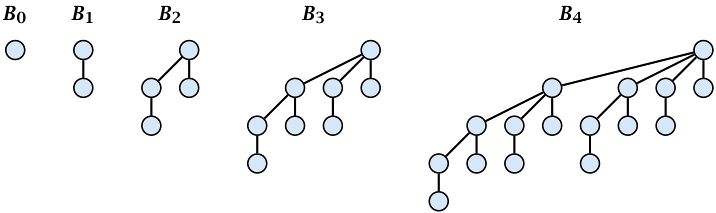
Finding the successor of x is much easier than in the description on the previous slide. Simply increase or decrease x.

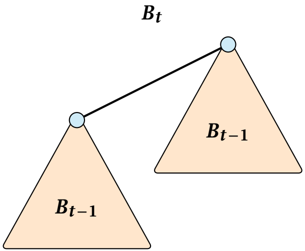
The resulting binary heap is not addressable. The elements don't maintain their positions and therefore there are no stable handles.

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

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 $B_0, B_1, \dots, B_{k-1}$.

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

Properties of Binomial Trees

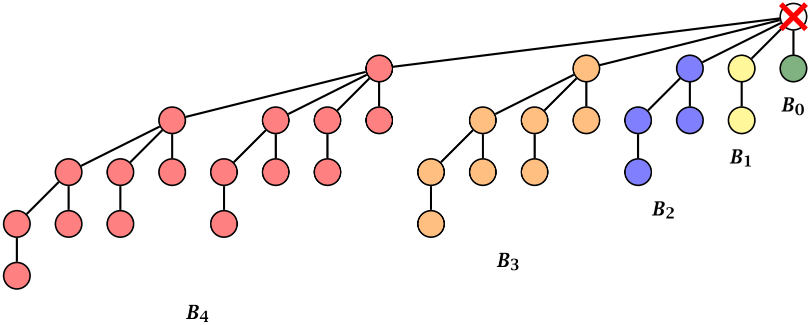
- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

Binomial Trees

Properties of Binomial Trees

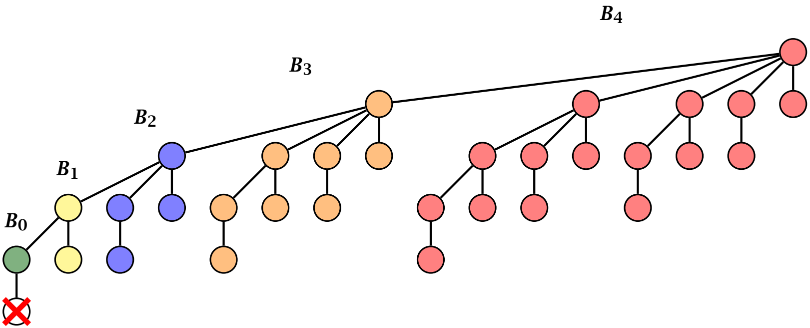
- B_k has 2^k nodes.
- B_k has height k.
- The root of B_k has degree k.
- B_k has $\binom{k}{\ell}$ nodes on level ℓ.
- Deleting the root of B_k gives trees $B_0, B_1, \ldots, B_{k-1}$.

Binomial Trees



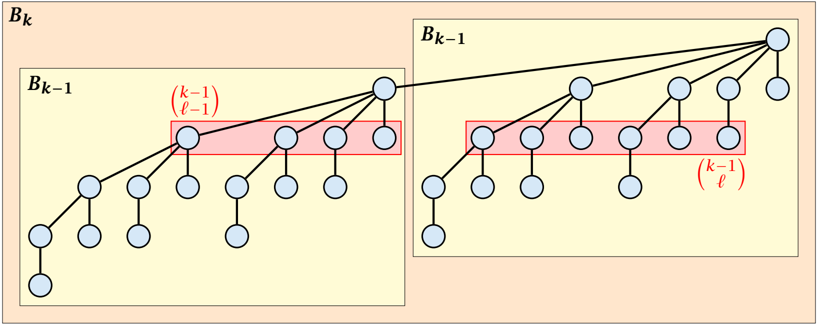
Deleting the root of B_5 leaves sub-trees B_4, B_3, B_2, B_1, and B_0.

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.

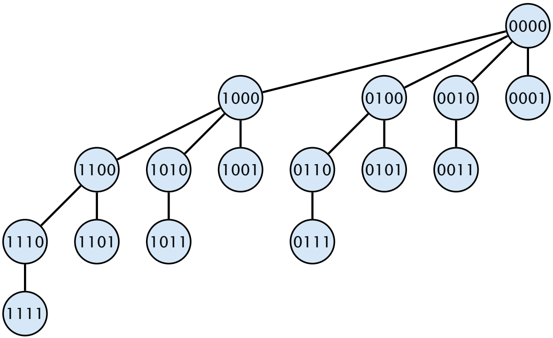
Binomial Trees



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

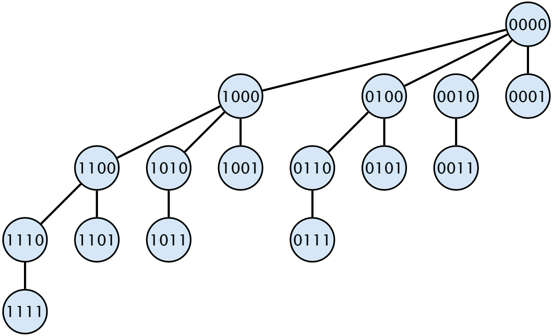


The binomial tree B_k is a sub-graph of the hypercube H_k.

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.

Binomial Trees

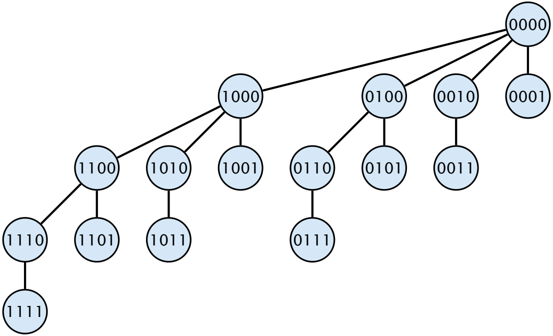


The binomial tree B_k is a sub-graph of the hypercube H_k.

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.

Binomial Trees

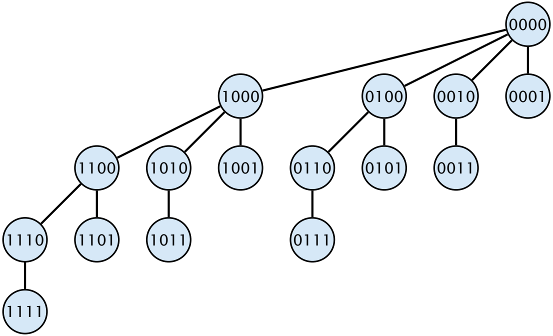


The binomial tree B_k is a sub-graph of the hypercube H_k.

The parent of a node with label $b_n, \dots, b_1, b_0$ 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.

Binomial Trees



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

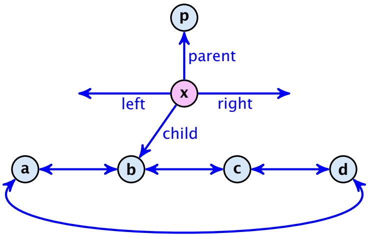
The parent of a node with label $b_n, \dots, b_1, b_0$ is obtained by setting the least significant 1-bit to 0.

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?

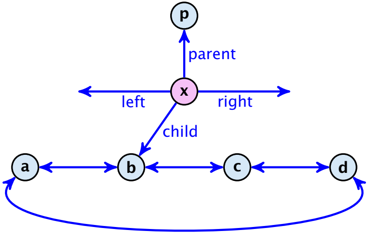
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\text{left}$ and $x.\text{right}$ point to the left and right sibling of x (if x does not have siblings then $x.\text{left}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



8.2 Binomial Heaps

How do we implement trees with non-constant degree?

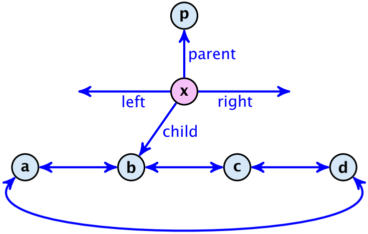
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\text{left}$ and $x.\text{right}$ point to the left and right sibling of x (if x does not have siblings then $x.\text{left}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



8.2 Binomial Heaps

How do we implement trees with non-constant degree?

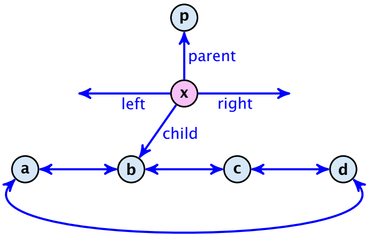
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\text{left}$ and $x.\text{right}$ point to the left and right sibling of x (if x does not have siblings then $x.\text{left}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



8.2 Binomial Heaps

How do we implement trees with non-constant degree?

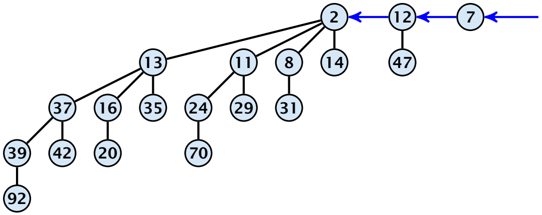
- The children of a node are arranged in a circular linked list.
- A child-pointer points to an arbitrary node within the list.
- A parent-pointer points to the parent node.
- Pointers $x.\text{left}$ and $x.\text{right}$ point to the left and right sibling of x (if x does not have siblings then $x.\text{left}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



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

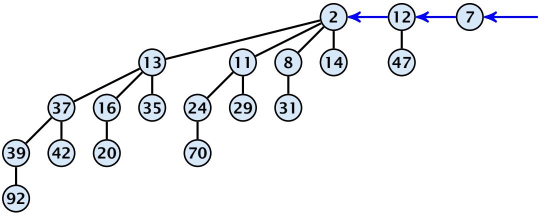


In a binomial heap the keys are arranged in a collection of binomial trees.

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

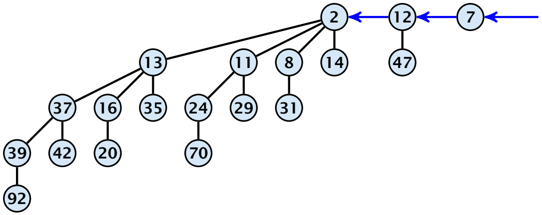


In a binomial heap the keys are arranged in a collection of binomial trees.

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

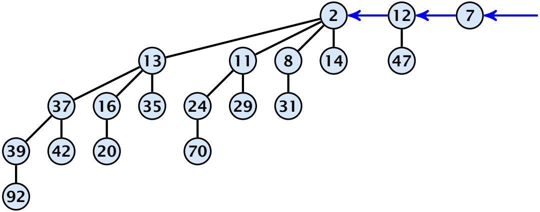


In a binomial heap the keys are arranged in a collection of binomial trees.

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



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 binary representation of n.

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 binary representation of n.

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 binary representation of n.

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.

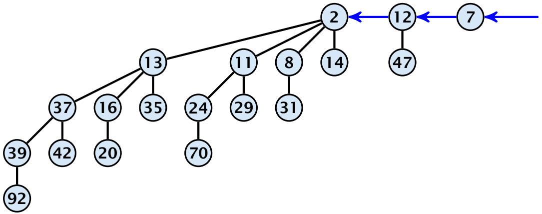
Let $B_{k_1}, B_{k_2}, B_{k_3}, k_i < k_{i+1}$ denote the binomial trees in the collection and recall that every tree may be contained at most once.

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 binary representation of n.

Binomial Heap

Properties of a heap with n keys:

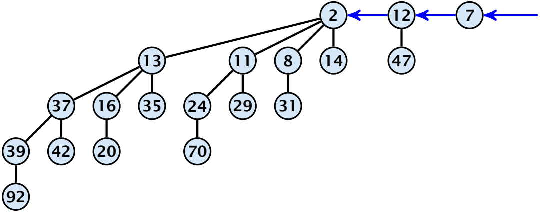
- Let $n = b_d b_{d-1}, \dots, b_0$ denote binary representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees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



Binomial Heap

Properties of a heap with n keys:

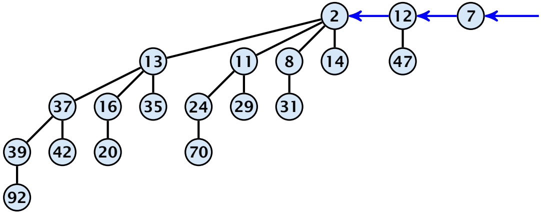
- Let $n = b_d b_{d-1}, \dots, b_0$ denote binary representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees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



Binomial Heap

Properties of a heap with n keys:

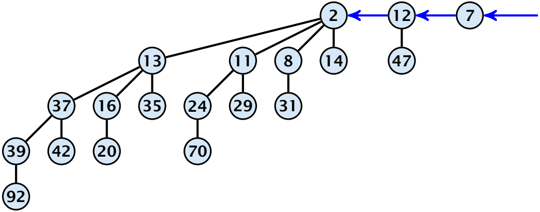
- Let $n = b_d b_{d-1}, \ldots, b_0$ denote binary representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees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



Binomial Heap

Properties of a heap with n keys:

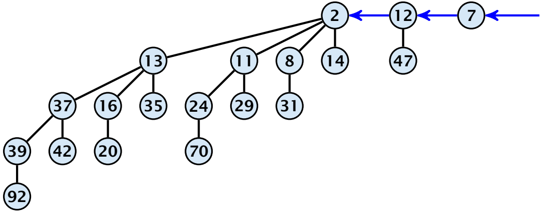
- Let $n = b_d b_{d-1}, \ldots, b_0$ denote binary representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees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



Binomial Heap

Properties of a heap with n keys:

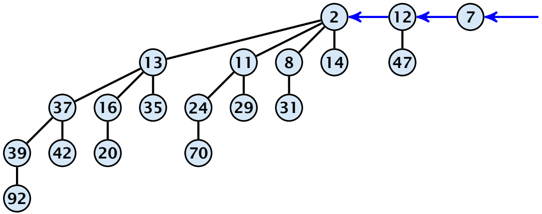
- Let $n = b_d b_{d-1}, \ldots, b_0$ denote binary representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees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



Binomial Heap

Properties of a heap with n keys:

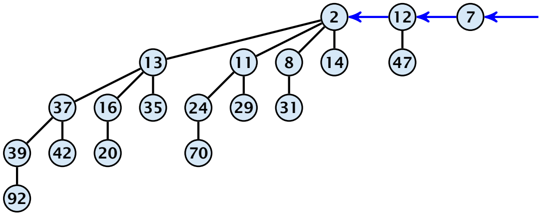
- Let $n = b_d b_{d-1}, \dots, b_0$ denote binary representation of n.
- The heap contains tree B_i iff $b_i = 1$.
- Hence, at most $\lfloor \log n \rfloor + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lfloor \log n \rfloor$.
- The trees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



Binomial Heap

Properties of a heap with n keys:

- Let $n = b_d b_{d-1}, \dots, b_0$ denote binary 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.

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.

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.

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.

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.

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.

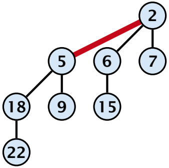
Binomial Heap: Merge

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.

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.

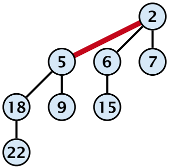
Binomial Heap: Merge

The merge-operation is instrumental for binomial heaps.

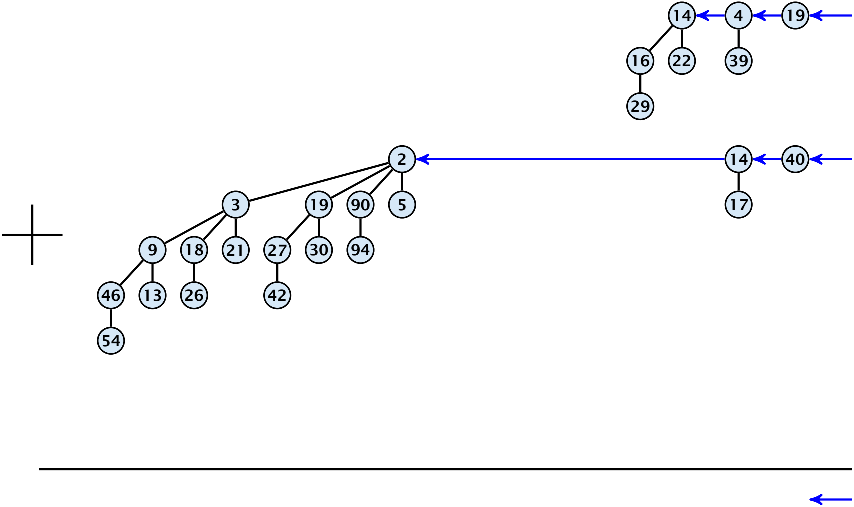
A merge is easy if we have two heaps with different binomial trees. We can simply merge the tree-lists.

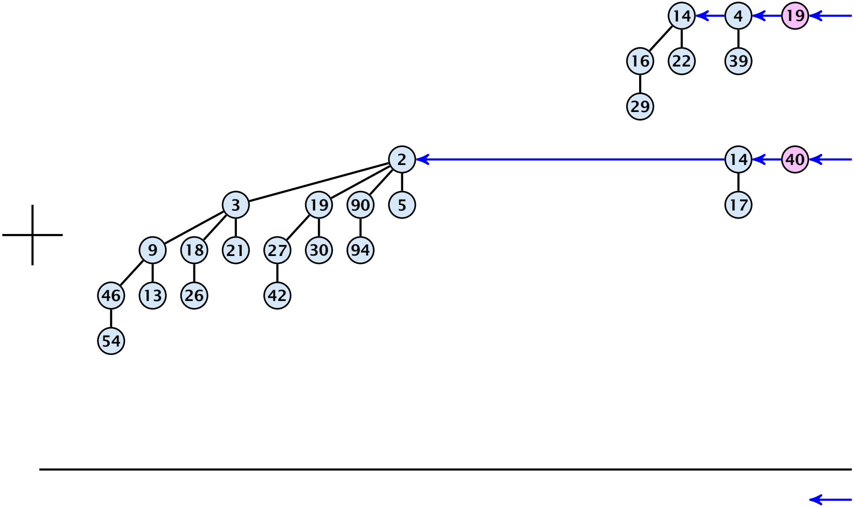
Otherwise, we cannot do this because the merged heap is not allowed to contain two trees of the same order.

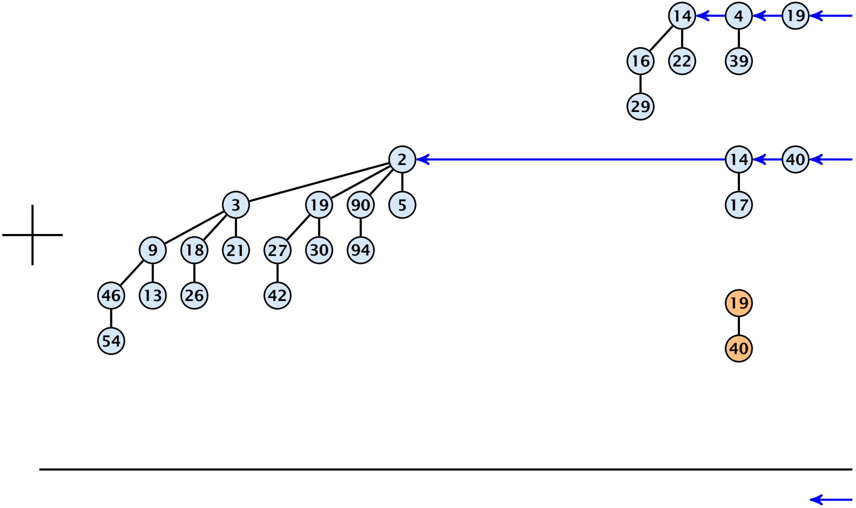
Merging two trees of the same size: Add the tree with larger root-value as a child to the other tree.

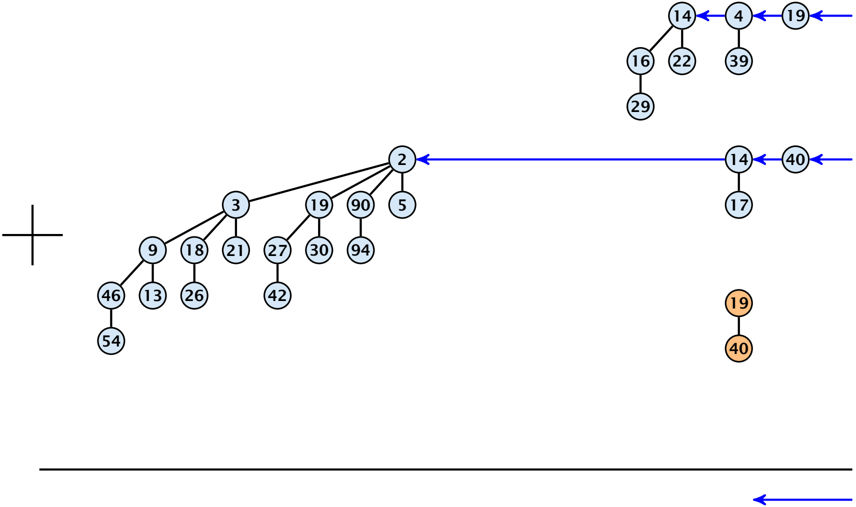


For more trees the technique is analogous to binary addition.

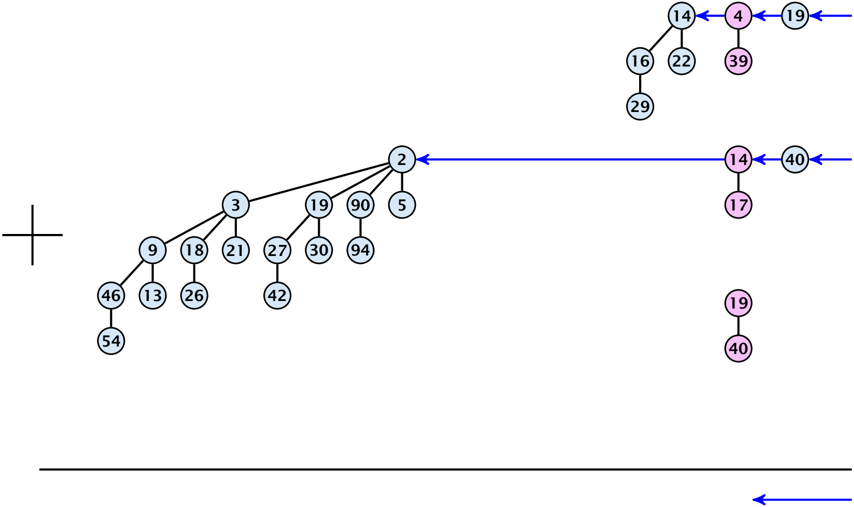

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54

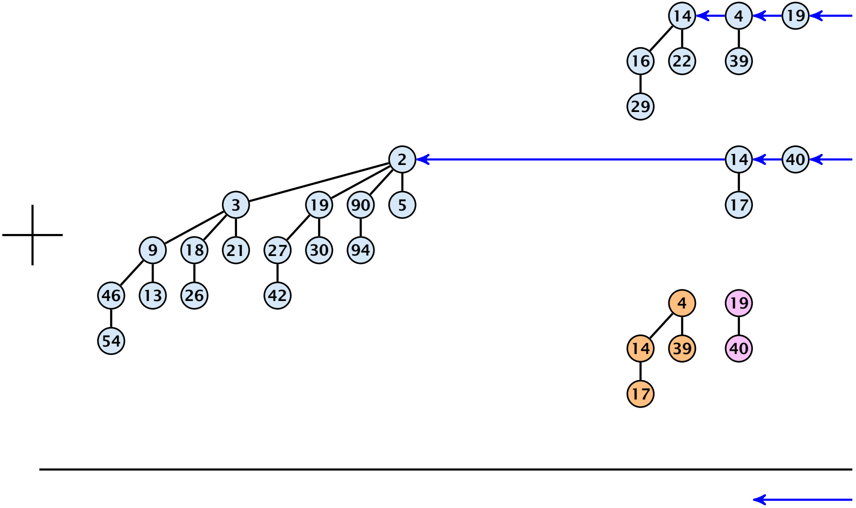

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54


14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
19
54
40

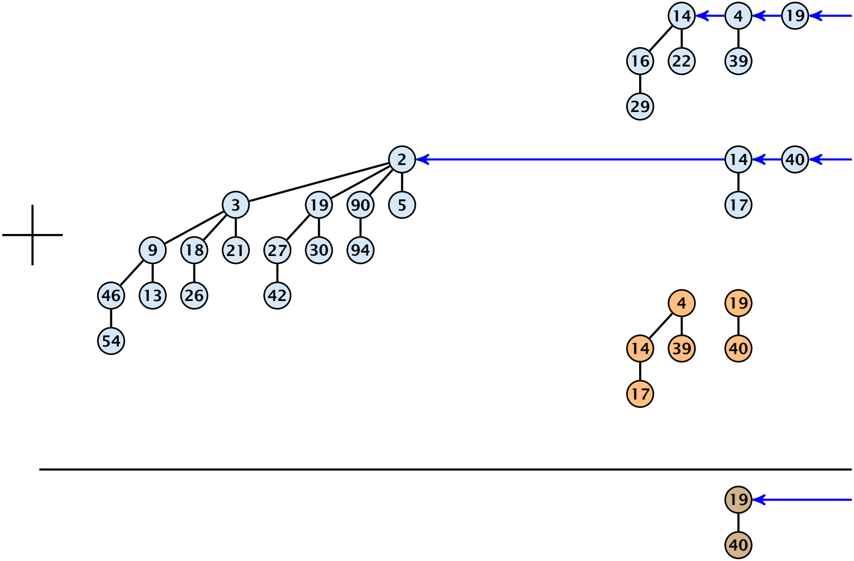


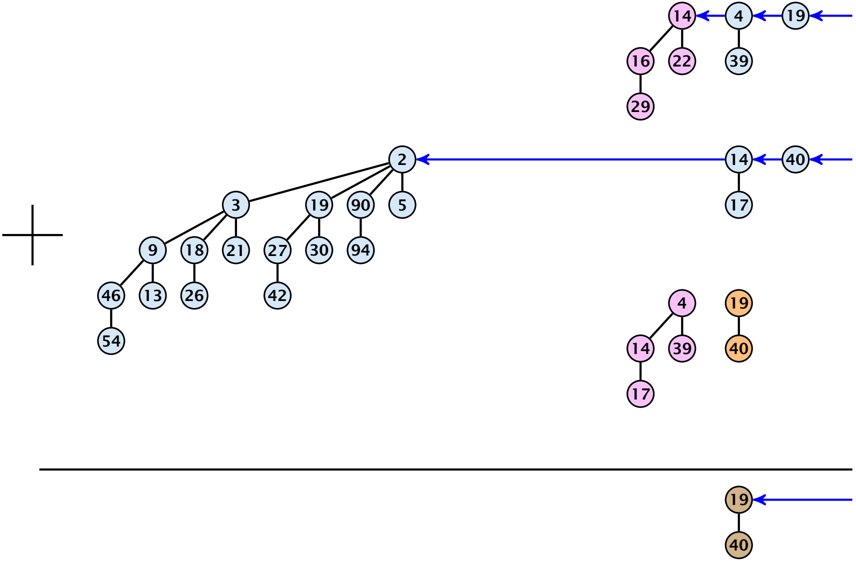
14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
19
54
40


14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
19
54
40

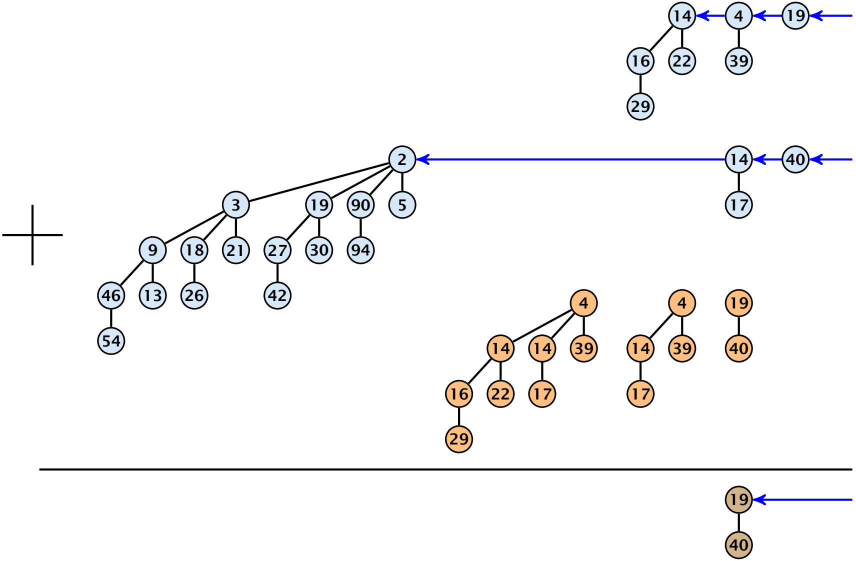


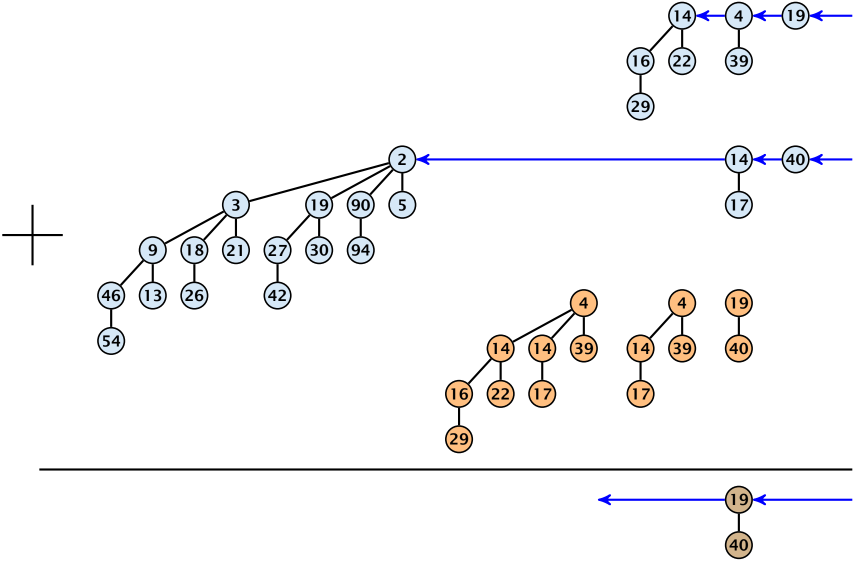
14 4 19
16 22 39
29
2 14 40
3 19 90 5 17
9 18 21 27 30 94
46 13 26 42
54
4 19
14 39 40
17

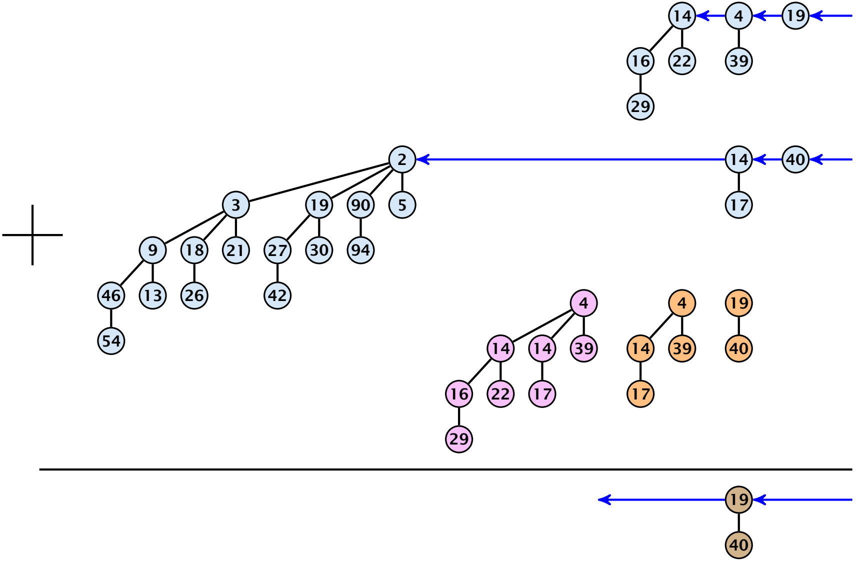

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
19
14
39
40
17
19
40



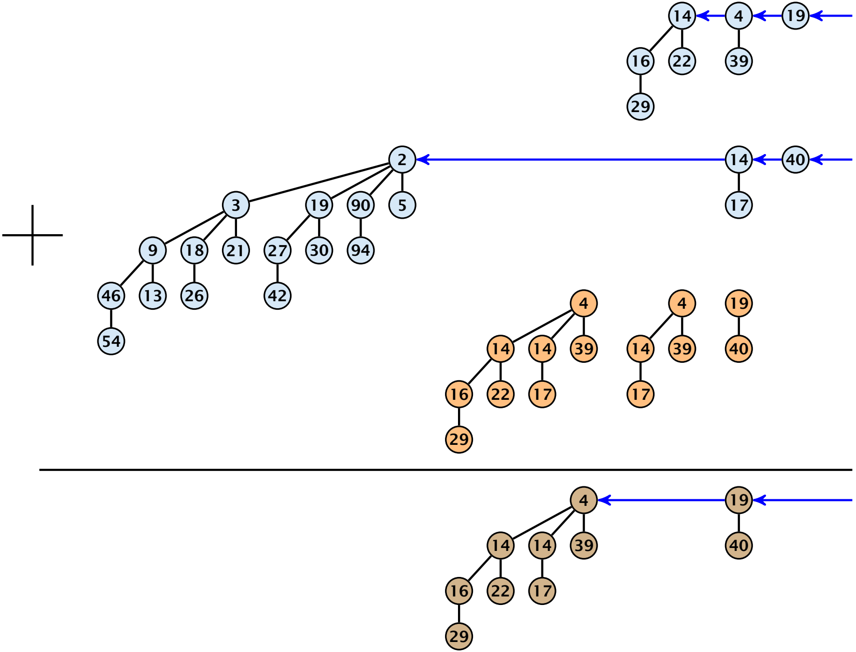
14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
4
19
54
14
39
40
17
19
40

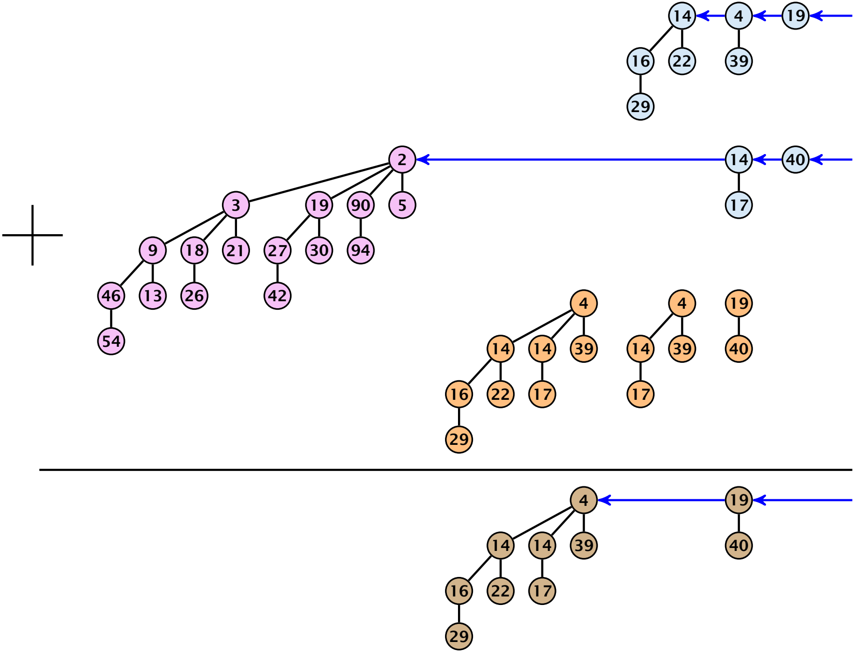

14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
19
40


14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
19
40

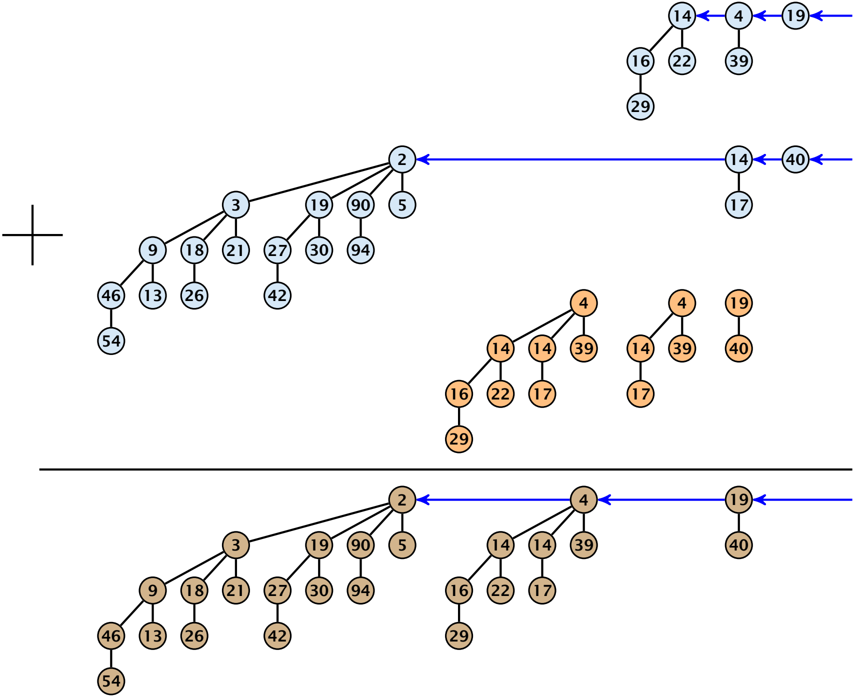


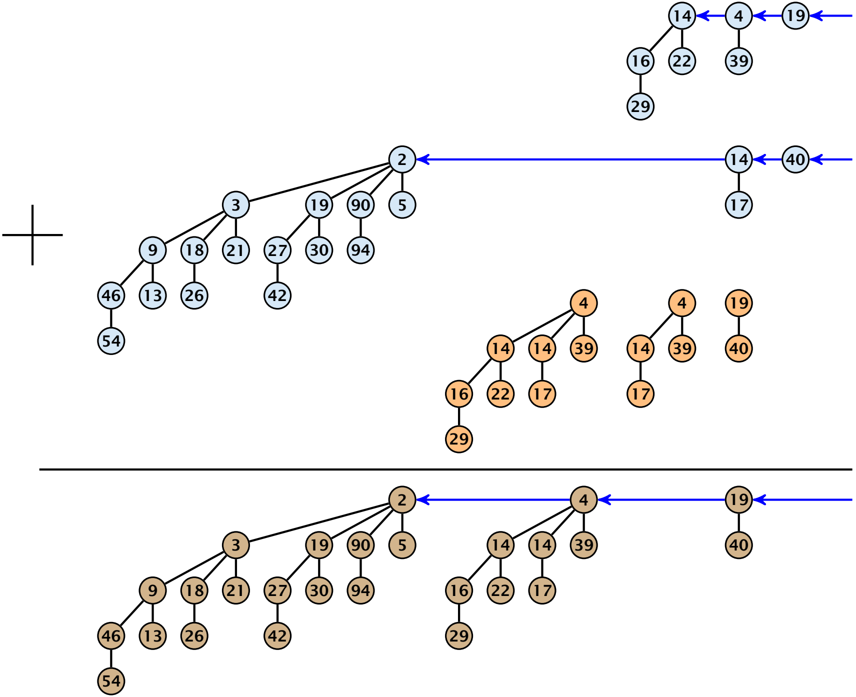
14 4 19
16 22 39
29
2 14 40
3 19 90 5 17
9 18 21 27 30 94
46 13 26 42
54
4 4 19
14 14 39 14 39 40
16 22 17 17
29
19
40


14 4 19
16 22 39
29
2 14 40
3 19 90 5 17
9 18 21 27 30 94
46 13 26 42
54
4 4 19
14 14 39 14 39 40
16 22 17 17
29
4 19
14 14 39 40
16 22 17
29



14
4
19
16
22
39
29
2
14
40
3
19
90
5
17
9
18
21
27
30
94
46
13
26
42
54
4
4
19
14
14
39
14
39
40
16
22
17
17
29
4
19
14
14
39
40
16
22
17
29


14 4 19
16 22 39
29
2 14 40
3 19 90 5 17
9 18 21 27 30 94
46 13 26 42
54
4 4 19
14 14 39 14 39 40
16 22 17 17
29
2 4 19
3 19 90 5 14 14 39 40
9 18 21 27 30 94 16 22 17
46 13 26 42 29
54


14 4 19
16 22 39
29
2 14 40
3 19 90 5 17
9 18 21 27 30 94
46 13 26 42
54
4 4 19
14 14 39 14 39 40
16 22 17 17
29
2 4 19
3 19 90 5 14 14 39 40
9 18 21 27 30 94 16 22 17
46 13 26 42 29
54

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

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

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

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

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.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.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*.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.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.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.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.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.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: $\mathcal{O}(\log n)$.

8.2 Binomial Heaps

S.delete(handle h):

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

8.2 Binomial Heaps

S.delete(handle h):

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

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 1

A data structure with operations $\text{op}_1(), \dots, \text{op}_k()$ has amortized running times $t_1, \dots, t_k$ 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 $\sum_i k_i \cdot t_i(n)$.

Potential Method

Introduce a potential for the data structure.

Potential Method

Introduce a potential for the data structure.

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

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

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

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 $\Phi(D_i) \geq \Phi(D_0)$.

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

$$\sum_{i=1}^{k} c_i$$

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

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

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

- $S.\mathbf{push}()$
- $S.\mathbf{pop}()$
- $S.\mathbf{multipop}(k)$: 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:

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

Example: Stack

Stack

- $S.\mathbf{push}()$
- $S.\mathbf{pop}()$
- $S.\mathbf{multipop}(k)$: 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:

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

Example: Stack

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

Example: Stack

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

Amortized cost:

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

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

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

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

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

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

Example: Stack

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

Amortized cost:

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

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

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

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

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

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

Example: Stack

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

Amortized cost:

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

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

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

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

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

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

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

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

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:

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

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

Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k operations, and one operation.

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

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:

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

- Changing bit from 1 to 0:

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

- 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 $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \leq 2$.

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 $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \le 2$.

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:

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

- Changing bit from 1 to 0:

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

- 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 $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \leq 2$.

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:

$$\hat{C}_{0\to1} = C_{0\to1} + \Delta\Phi = 1 + 1 \le 2 \ .$$

- Changing bit from 1 to 0:

$$\hat{C}_{1\to0} = C_{1\to0} + \Delta\Phi = 1 - 1 \le 0 \ .$$

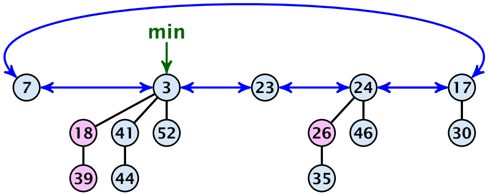
- Increment: Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to0} + \hat{C}_{0\to1} \le 2$.

8.3 Fibonacci Heaps

Collection of trees that fulfill the heap property.

Structure is much more relaxed than binomial heaps.



8.3 Fibonacci Heaps

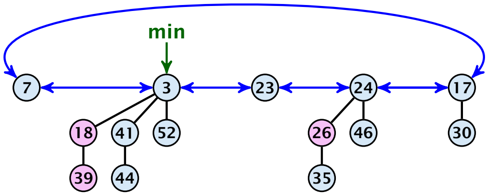
Additional implementation details:

- Every node x stores its degree in a field $x.\mathrm{degree}$. Note that this can be updated in constant time when adding a child to x.
- Every node stores a boolean value $x.\mathrm{marked}$ 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

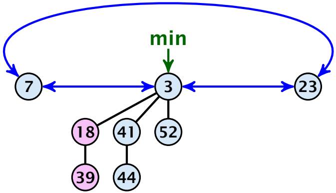
$S.\textbf{minimum}()$

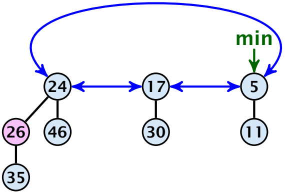
- Access through the min-pointer.
- Actual cost $\mathcal{O}(1)$.
- No change in potential.
- Amortized cost $\mathcal{O}(1)$.

8.3 Fibonacci Heaps

$S.\,\mathrm{merge}(S')$

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

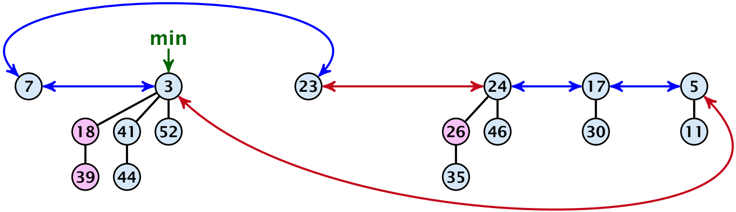




8.3 Fibonacci Heaps

$S.\text{merge}(S')$

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



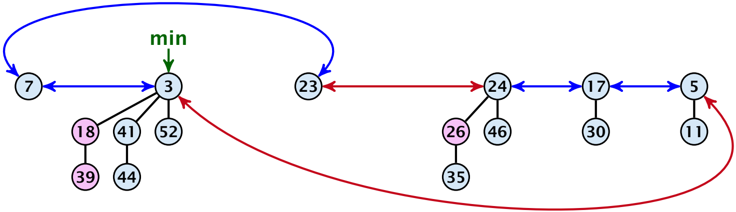
Running time:

- Actual cost $\mathcal{O}(1)$.

8.3 Fibonacci Heaps

$S.\mathrm{merge}(S')$

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



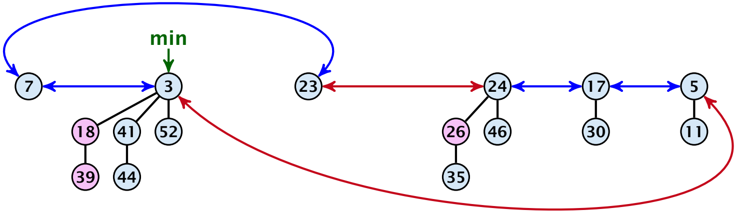
Running time:

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

8.3 Fibonacci Heaps

$S.\,\mathrm{merge}(S')$

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



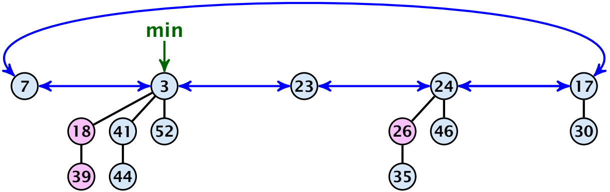
Running time:

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

8.3 Fibonacci Heaps

$S.\text{insert}(x)$

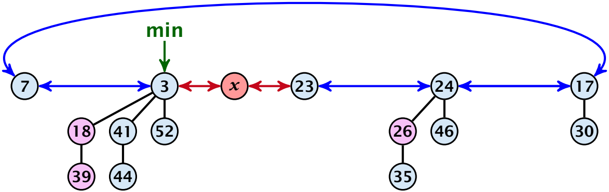
- Create a new tree containing x.
- Insert x into the root-list.
- Update min-pointer, if necessary.



8.3 Fibonacci Heaps

S. insert(x)

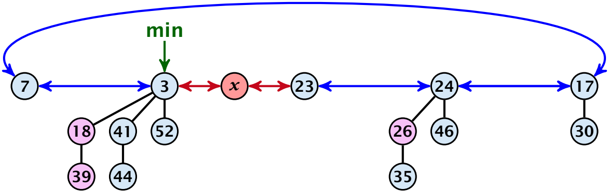
- Create a new tree containing x.
- Insert x into the root-list.
- Update min-pointer, if necessary.



8.3 Fibonacci Heaps

$S.$ insert(x)

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

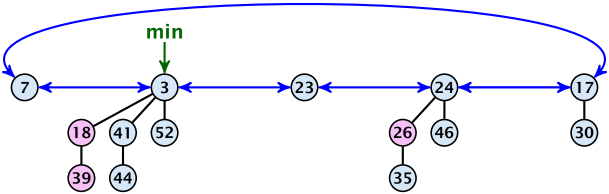


Running time:

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

8.3 Fibonacci Heaps

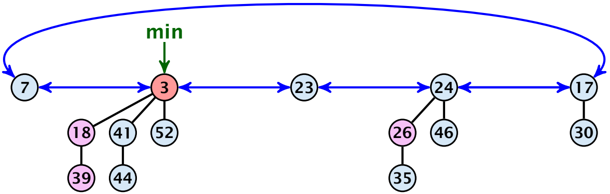
S. delete-min(x)



8.3 Fibonacci Heaps

S. delete-min(x)

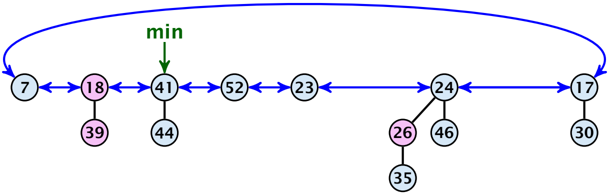
- Delete minimum; add child-trees to heap; time: $D(\min) \cdot \mathcal{O}(1)$.



8.3 Fibonacci Heaps

S. delete-min(x)

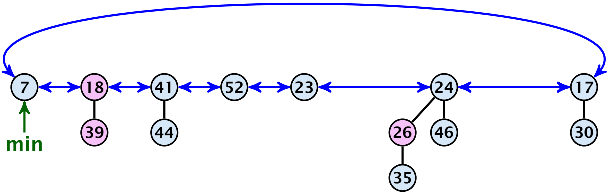
- Delete minimum; add child-trees to heap; time: $D(\min) \cdot \mathcal{O}(1)$.
- Update min-pointer; time: $(t + D(\min)) \cdot \mathcal{O}(1)$.



8.3 Fibonacci Heaps

S. delete-min(x)

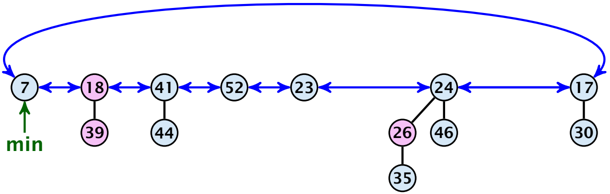
- Delete minimum; add child-trees to heap; time: $D(\min) \cdot \mathcal{O}(1)$.
- Update min-pointer; time: $(t + D(\min)) \cdot \mathcal{O}(1)$.



8.3 Fibonacci Heaps

***S*. delete-min(*x*)**

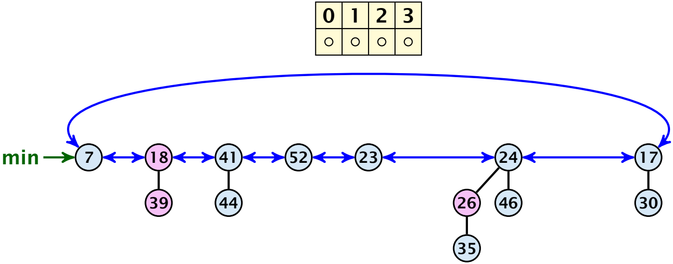
- Delete minimum; add child-trees to heap; time: $D(\min) \cdot \mathcal{O}(1)$.
- Update min-pointer; time: $(t + D(\min)) \cdot \mathcal{O}(1)$.



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

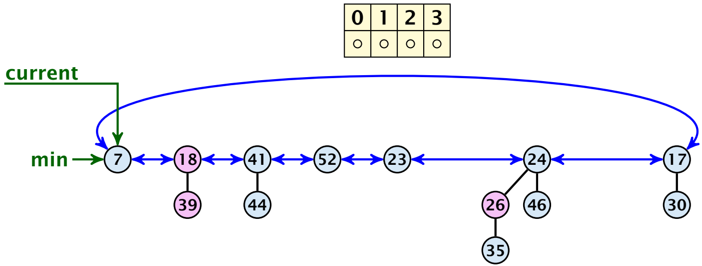
8.3 Fibonacci Heaps

Consolidate:



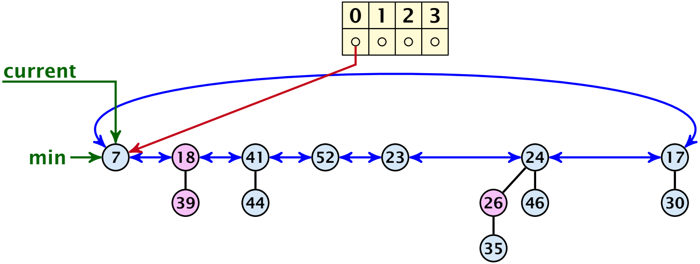
8.3 Fibonacci Heaps

Consolidate:



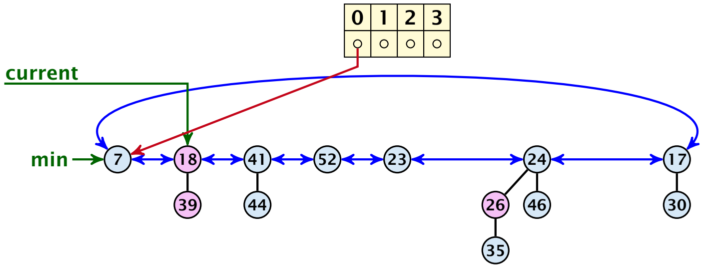
8.3 Fibonacci Heaps

Consolidate:



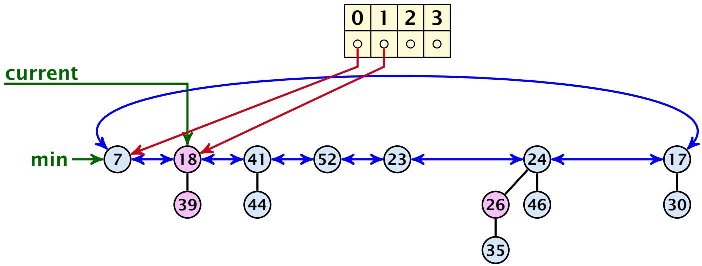
8.3 Fibonacci Heaps

Consolidate:



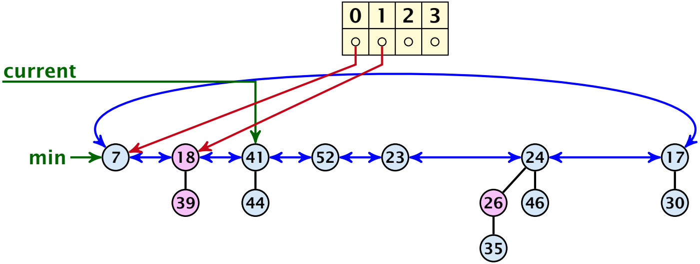
8.3 Fibonacci Heaps

Consolidate:



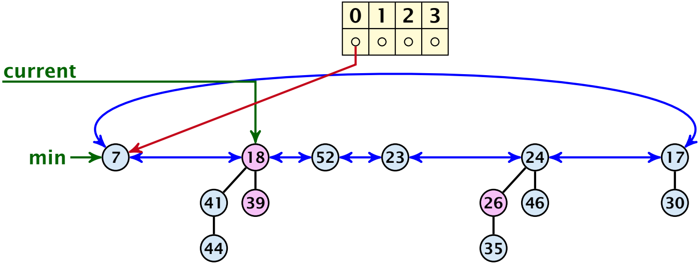
8.3 Fibonacci Heaps

Consolidate:



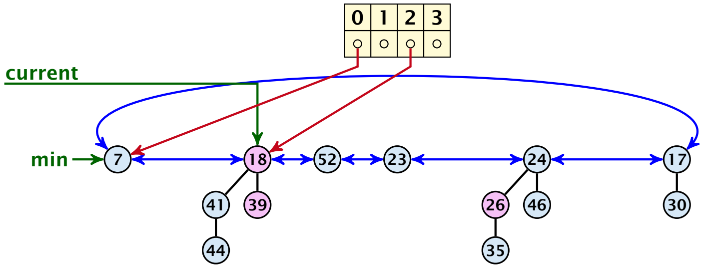
8.3 Fibonacci Heaps

Consolidate:



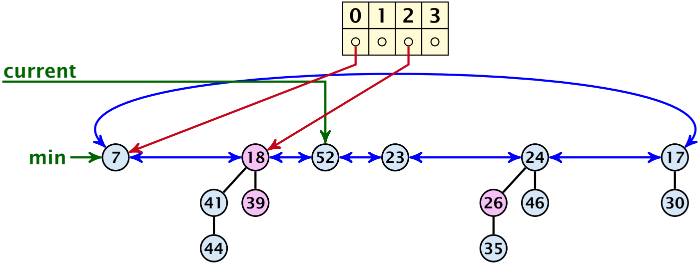
8.3 Fibonacci Heaps

Consolidate:



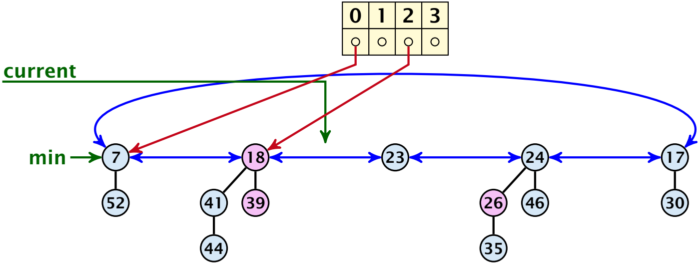
8.3 Fibonacci Heaps

Consolidate:



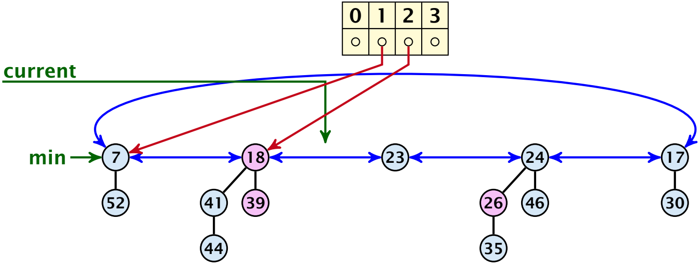
8.3 Fibonacci Heaps

Consolidate:



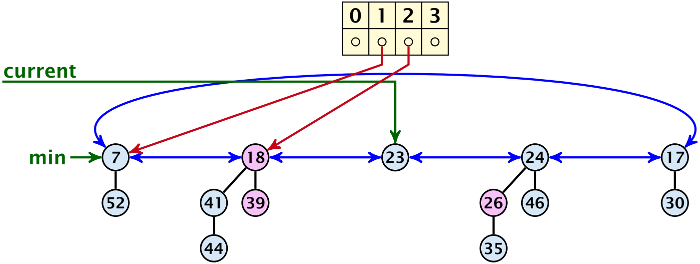
8.3 Fibonacci Heaps

Consolidate:



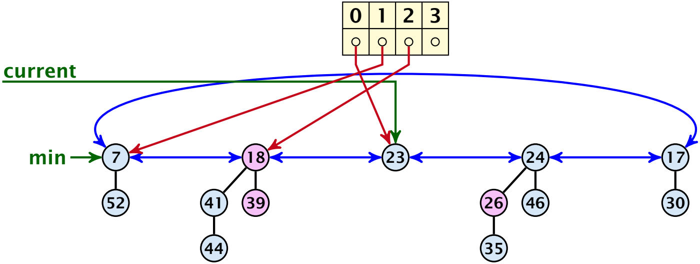
8.3 Fibonacci Heaps

Consolidate:



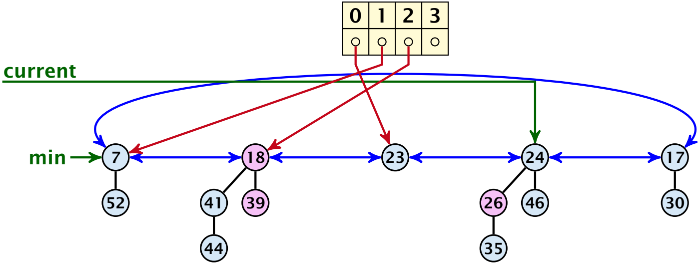
8.3 Fibonacci Heaps

Consolidate:



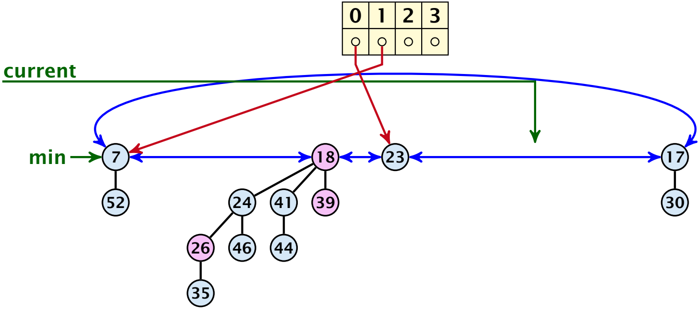
8.3 Fibonacci Heaps

Consolidate:



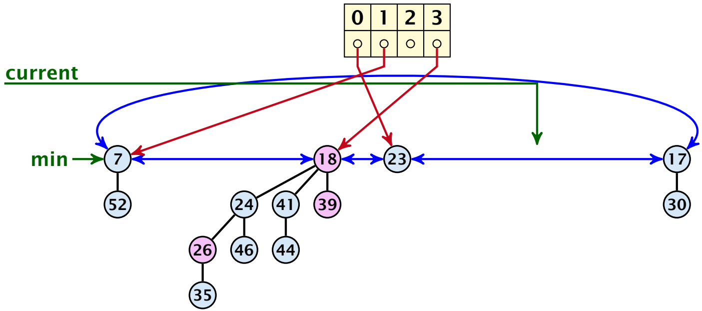
8.3 Fibonacci Heaps

Consolidate:



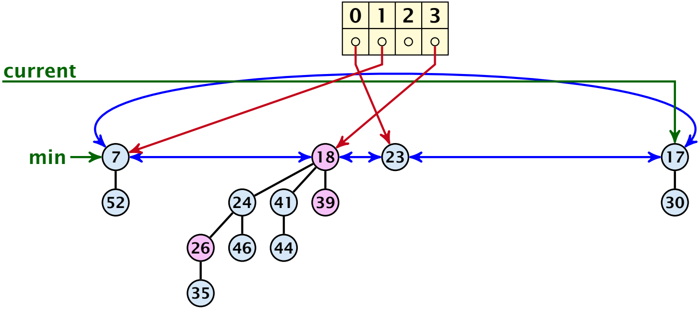
8.3 Fibonacci Heaps

Consolidate:



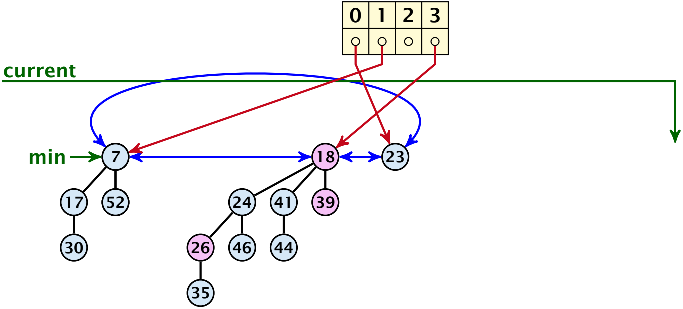
8.3 Fibonacci Heaps

Consolidate:



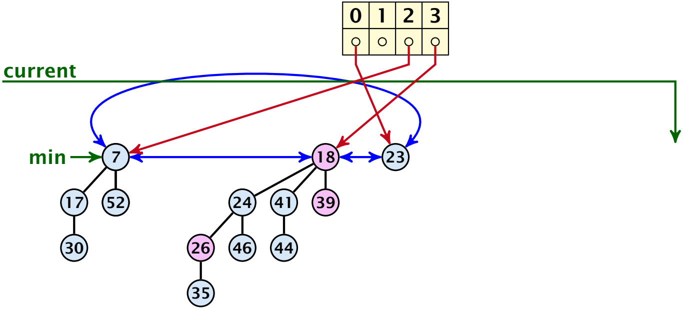
8.3 Fibonacci Heaps

Consolidate:



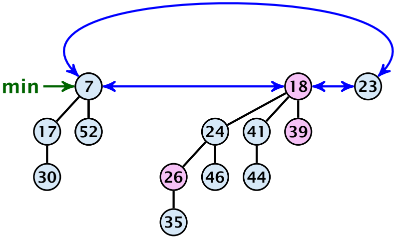
8.3 Fibonacci Heaps

Consolidate:



8.3 Fibonacci Heaps

Consolidate:



8.3 Fibonacci Heaps

Actual cost for delete-min()

- At most $D_n + t$ elements in root-list before consolidate.

8.3 Fibonacci Heaps

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 $c_1 \cdot (D_n + t)$.

8.3 Fibonacci Heaps

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.

8.3 Fibonacci Heaps

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()

- $t' \le D_n + 1$ as degrees are different after consolidating.
- Therefore $\Delta\Phi \le D_n + 1 - t$;

8.3 Fibonacci Heaps

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()

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

8.3 Fibonacci Heaps

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()

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

8.3 Fibonacci Heaps

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()

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

  $$c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1)$$

8.3 Fibonacci Heaps

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()

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

$$
\begin{aligned}
c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1) \\
\le (c_1 + c)D_n + (c_1 - c)t + c
\end{aligned}
$$

8.3 Fibonacci Heaps

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()

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

$$
\begin{aligned}
& c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1) \\
& \qquad \le (c_1 + c)D_n + (c_1 - c)t + c \le 2c(D_n + 1)
\end{aligned}
$$

8.3 Fibonacci Heaps

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

$$c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1)$$

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

8.3 Fibonacci Heaps

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()

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

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

for $c \ge c_1$.

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.

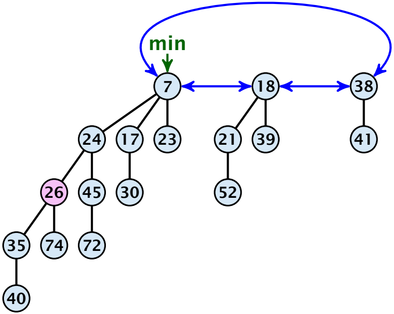
If we do not have delete or decrease-key operations then $D_n \leq \log n$.

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 $D_n \leq \log n$.

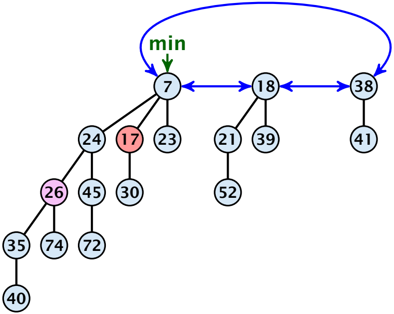
Fibonacci Heaps: decrease-key(handle h, v)



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

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

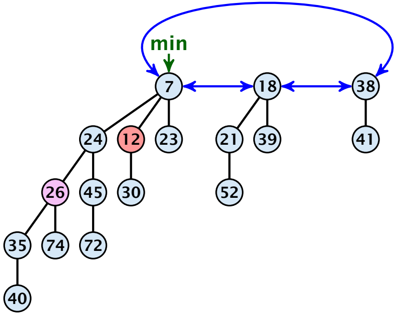
Fibonacci Heaps: decrease-key(handle h, v)



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

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

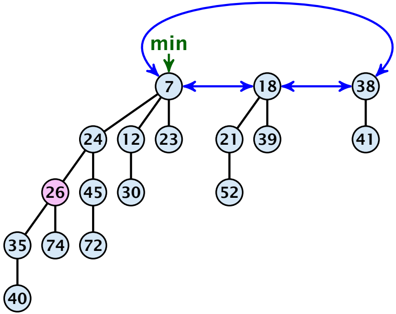
Fibonacci Heaps: decrease-key(handle h, v)



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

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

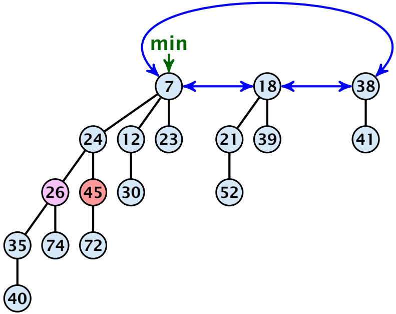
Fibonacci Heaps: decrease-key(handle h, v)



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

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

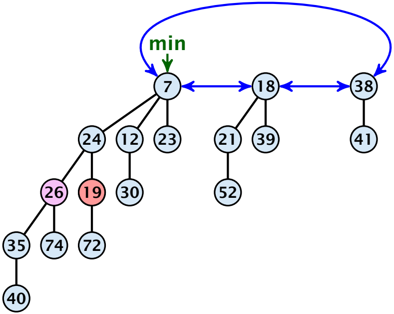
Fibonacci Heaps: decrease-key(handle h, v)



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).

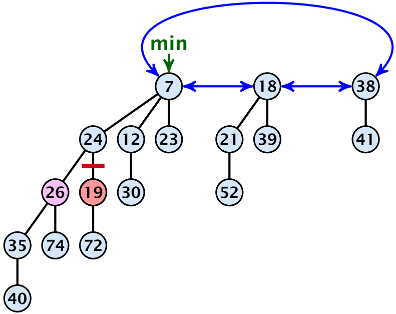
Fibonacci Heaps: decrease-key(handle h, v)



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).

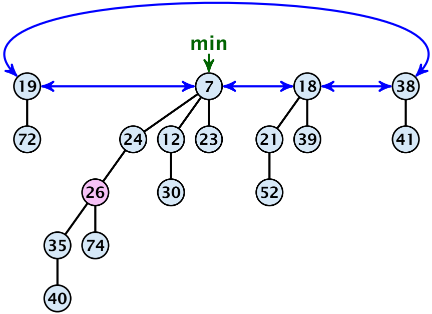
Fibonacci Heaps: decrease-key(handle h, v)



Case 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).

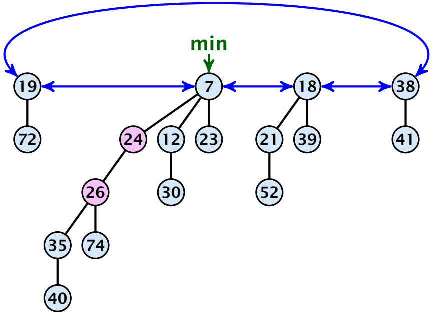
Fibonacci Heaps: decrease-key(handle h, v)



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).

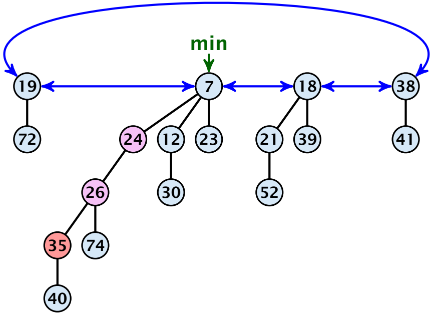
Fibonacci Heaps: decrease-key(handle h, v)



Case 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).

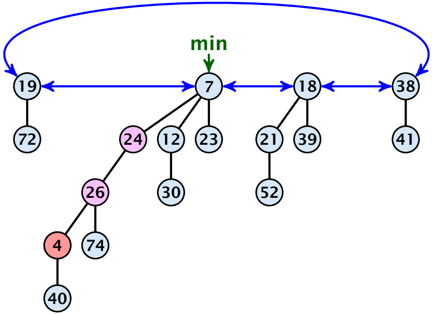
Fibonacci Heaps: decrease-key(handle h, v)



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.

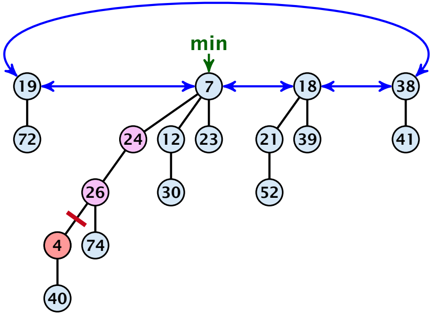
Fibonacci Heaps: decrease-key(handle h, v)



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.

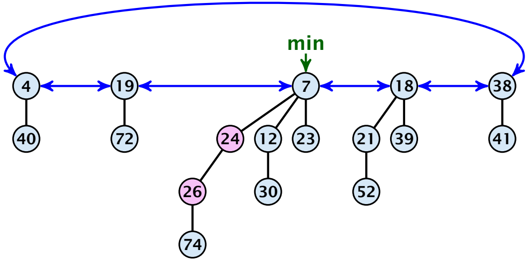
Fibonacci Heaps: decrease-key(handle h, v)



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.
- Continue cutting the parent until you arrive at an unmarked node.

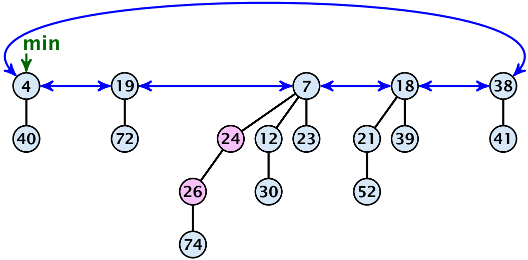
Fibonacci Heaps: decrease-key(handle h, v)



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.

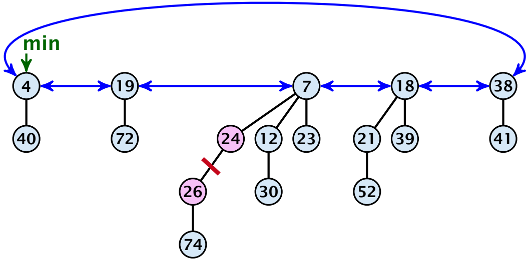
Fibonacci Heaps: decrease-key(handle h, v)



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.
- Continue cutting the parent until you arrive at an unmarked node.

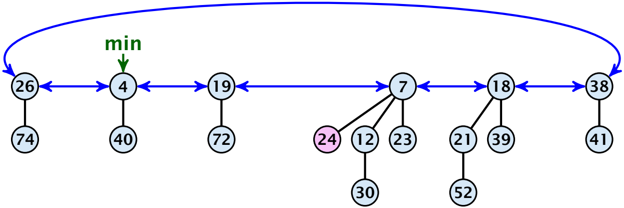
Fibonacci Heaps: decrease-key(handle h, v)



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.

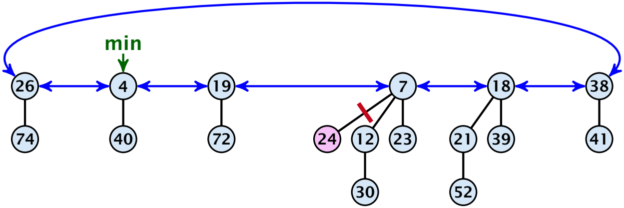
Fibonacci Heaps: decrease-key(handle h, v)



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.
- Continue cutting the parent until you arrive at an unmarked node.

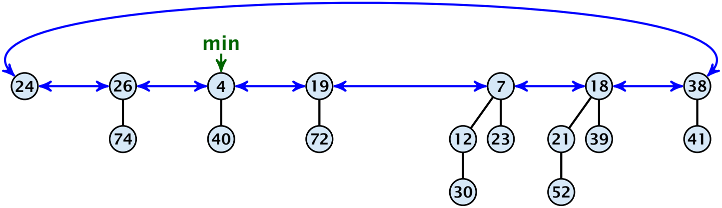
Fibonacci Heaps: decrease-key(handle h, v)



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.
- 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:

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

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:

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

if $c \ge c_2$.

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

if $c \ge c_2$.

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

if $c \ge c_2$.

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

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

  if $c \ge c_2$.

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.
- $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut unmarks a node; the last cut may mark a node.
- $\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

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

if $c \geq c_2$.

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.
- $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut unmarks a node; the last cut may mark a node.
- $\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

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

  if $c \geq c_2$.

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.
- $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut unmarks a node; the last cut may mark a node.
- $\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

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

  if $c \geq c_2$.

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.
- $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut unmarks a node; the last cut may mark a node.
- $\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

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

  if $c \geq c_2$.

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

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

if $c \ge c_2$.

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.
- $m' \leq m - (\ell - 1) + 1 = m - \ell + 2$, since all but the first cut unmarks a node; the last cut may mark a node.
- $\Delta\Phi \leq \ell + 2(-\ell + 2) = 4 - \ell$
- Amortized cost is at most

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

  if $c \geq c_2$.

Delete node

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

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

Amortized cost: $\mathcal{O}(D_n)$

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

8.3 Fibonacci Heaps

Lemma 2
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*

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

8.3 Fibonacci Heaps

Proof

- When y_i was linked to x, at least $y_1, \dots, y_{i-1}$ 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

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 $\mathrm{degree}(x) \geq i - 1$, and therefore also $\mathrm{degree}(y_i) \geq i - 1$ as the algorithm links nodes of equal degree only.
- Since, then y_i has lost at most one child.
- Therefore, $\mathrm{degree}(y_i) \geq i - 2$.

8.3 Fibonacci Heaps

Proof

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

8.3 Fibonacci Heaps

Proof

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

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.

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

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 $s_1 = 2$.

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 $y_1, \dots, y_k$ be its children.

$$s_k = 2 + \sum_{i=2}^{k} \mathrm{size}(y_i)$$

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.

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

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.

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

8.3 Fibonacci Heaps

Definition 3

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.