8 Priority Queues

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

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

Sometimes we also have

- **$S. \text{merge}(S')$**: $S := S \cup S'$; $S' := \emptyset$. 
8 Priority Queues

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

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

Sometimes we also have

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

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

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

Sometimes we also have

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

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

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

Ernst Mayr, Harald Räcke
A Priority Queue $S$ is a dynamic set data structure that supports the following operations:

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

Sometimes we also have

- $S. \text{merge}(S')$: $S := S \cup S'$; $S' := \emptyset$. 

8 Priority Queues
An **addressable Priority Queue** also supports:

- **handle** $S. \text{insert}(x)$: Adds element $x$ to the data-structure, and returns a handle to the object for future reference.
- **$S. \text{delete}(h)$**: Deletes element specified through handle $h$.
- **$S. \text{decrease-key}(h, k)$**: 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 $S.\text{insert}(x)$**: Adds element $x$ to the data-structure, and returns a handle to the object for future reference.
- **$S.\text{delete}(h)$**: Deletes element specified through handle $h$.
- **$S.\text{decrease-key}(h, k)$**: Decreases the key of the element specified by handle $h$ to $k$. Assumes that the key is at least $k$ before the operation.
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.
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.
Algorithm 14 Shortest-Path\((G = (V, E, d), s \in V)\)

1: **Input:** weighted graph \(G = (V, E, d)\); start vertex \(s\);
2: **Output:** key-field of every node contains distance from \(s\);
3: \(S\).build(); // build empty priority queue
4: **for all** \(v \in V \setminus \{s\} \) **do**
5: \(v\).key \(\leftarrow \infty\);
6: \(h_v \leftarrow S\).insert\((v)\);
7: \(s\).key \(\leftarrow 0\); \(S\).insert\((s)\);
8: **while** \(S\).is-empty() \(\neq\) false **do**
9: \(v \leftarrow S\).delete-min();
10: **for all** \(x \in V\) s.t. \((v, x) \in E\) **do**
11: \(\text{if } x\).key \(> v\).key \(+\) \(d(v, x)\) **then**
12: \(S\).decrease-key\((h_x, v\).key \(+\) \(d(v, x)\))
13: \(x\).key \(\leftarrow v\).key \(+\) \(d(v, x)\)
Algorithm 15 Prim-MST\((G = (V, E, d), s \in V)\)

1: Input: weighted graph \(G = (V, E, d)\); start vertex \(s\);
2: Output: pred-fields encode MST;
3: \(S\).build(); // build empty priority queue
4: for all \(v \in V \setminus \{s\}\) do
5: \(v\).key \(\leftarrow \infty\);
6: \(h_v \leftarrow S\).insert\((v)\);
7: \(s\).key \(\leftarrow 0\); \(S\).insert\((s)\);
8: while \(S\).is-empty\((\) = false do
9: \(v \leftarrow S\).delete-min\((\));
10: for all \(x \in V\) s.t. \(\{v, x\} \in E\) do
11: if \(x\).key > \(d(v, x)\) then
12: \(S\).decrease-key\((h_x, d(v, x))\);
13: \(x\).key \(\leftarrow d(v, x)\);
14: \(x\).pred \(\leftarrow v\);
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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary Heap</th>
<th>BST</th>
<th>Binomial Heap</th>
<th>Fibonacci Heap*</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n$</td>
</tr>
<tr>
<td>minimum</td>
<td>1</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>is-empty</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>insert</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>delete</td>
<td>$\log n^{**}$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>delete-min</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>decrease-key</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>merge</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary Heap</th>
<th>BST</th>
<th>Binomial Heap</th>
<th>Fibonacci Heap*</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n$</td>
</tr>
<tr>
<td>minimum</td>
<td>1</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>is-empty</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>insert</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>delete</td>
<td>$\log n^{**}$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>delete-min</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>decrease-key</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>merge</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary Heap</th>
<th>BST</th>
<th>Binomial Heap</th>
<th>Fibonacci Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n$</td>
</tr>
<tr>
<td>minimum</td>
<td>1</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>is-empty</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>insert</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>delete</td>
<td>$\log n^{**}$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>delete-min</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>decrease-key</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>merge</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary Heap</th>
<th>BST</th>
<th>Binomial Heap</th>
<th>Fibonacci Heap*</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n$</td>
</tr>
<tr>
<td>minimum</td>
<td>1</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>is-empty</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>insert</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>delete</td>
<td>$\log n^{**}$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>delete-min</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>decrease-key</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>merge</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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

- Nearly complete binary tree; only the last level is not full, and this one is filled from left to right.
- Heap property: A node's key is not larger than the key of one of its children.
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 $\Theta(1)$.
- **is-empty()**: check whether root-pointer is `null`. Time $\Theta(1)$. 
Binary Heaps

Operations:

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

  - 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 $\Theta(\log n)$.

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 $\Theta(\log n)$.

  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 $\Theta(\log n)$.

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 $\Theta(\log n)$.

  - 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 $\Theta(\log n)$.

  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 \( \Theta(\log n) \).

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

If you go up until the last edge used was a left edge, go right; go left until you reach a null-pointer.
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 $\Theta(\log n)$.

  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 $\Theta(1)$.
▶ **is-empty()**: check whether root-pointer is null. Time $\Theta(1)$.
▶ **insert($k$)**: insert at successor of $x$ and bubble up. Time $\Theta(\log n)$.
▶ **delete($h$)**: swap with $x$ and bubble up or sift-down. Time $\Theta(\log n)$.
We can build a heap in linear time:

$$\sum_{\ell} 2^\ell \cdot (h - \ell) = \sum_i i 2^{h-i} = O(2^h) = O(n)$$
We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i 2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i 2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \sum_i i 2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i 2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i 2^{h-i} = O(2^h) = O(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i 2^{h-i} = O(2^h) = O(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i 2^{h-i} = \Theta(2^h) = \Theta(n)
\]
We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
We can build a heap in linear time:

\[ \sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \sum_{i} i2^{h-i} = \Theta(2^h) = \Theta(n) \]
We can build a heap in linear time:

\[ \sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n) \]
Build Heap

We can build a heap in linear time:

\[ \sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = O(2^h) = O(n) \]
Build Heap

We can build a heap in linear time:

\[ \sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_{i} i2^{h-i} = \Theta(2^h) = \Theta(n) \]
Build Heap

We can build a heap in linear time:

\[ \sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = O(2^h) = O(n) \]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^{\ell} \cdot (h - \ell) = \sum_i i2^{h-i} = O(2^h) = O(n)
\]
Build Heap

We can build a heap in linear time:

\[ \sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_{i} i2^{h-i} = \Theta(2^h) = \Theta(n) \]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i 2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[ \sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i 2^{h-i} = \Theta(2^h) = \Theta(n) \]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i 2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i 2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = O(2^h) = O(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i 2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
Build Heap

We can build a heap in linear time:

\[
\sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n)
\]
We can build a heap in linear time:

\[ \sum_{\text{levels } \ell} 2^\ell \cdot (h - \ell) = \sum_i i2^{h-i} = \Theta(2^h) = \Theta(n) \]
We can build a heap in linear time:

\[ \sum_{\ell} 2^\ell \cdot (h - \ell) = \sum_i i \cdot 2^{h-i} = \Theta(2^h) = \Theta(n) \]
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($x_1, \ldots, x_n$):** Insert elements arbitrarily; then do sift-down operations starting with the lowest layer in the tree. Time $\mathcal{O}(n)$. 
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.
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.
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.
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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary Heap</th>
<th>BST</th>
<th>Binomial Heap</th>
<th>Fibonacci Heap*</th>
</tr>
</thead>
<tbody>
<tr>
<td>build</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$n \log n$</td>
<td>$n$</td>
</tr>
<tr>
<td>minimum</td>
<td>1</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>is-empty</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>insert</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>delete</td>
<td>$\log n^{**}$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>delete-min</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>decrease-key</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
<tr>
<td>merge</td>
<td>$n$</td>
<td>$n \log n$</td>
<td>$\log n$</td>
<td>1</td>
</tr>
</tbody>
</table>
Binomial Trees

\[ B_0 \quad B_1 \quad B_2 \quad B_3 \quad B_4 \]

\[ B_{t-1} \quad B_t \quad B_{t-1} \]

8.2 Binomial Heaps
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 $\ell$.
- 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 $\ell$.
- Deleting the root of $B_k$ gives trees $B_0, B_1, \ldots, B_{k-1}$. 

8.2 Binomial Heaps
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 $\ell$.
- 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 \( \ell \).
- 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 $\ell$.
- Deleting the root of $B_k$ gives trees $B_0, B_1, \ldots, B_{k-1}$.
Deleting the root of $B_5$ leaves sub-trees $B_4$, $B_3$, $B_2$, $B_1$, and $B_0$. 
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 $B_0$. 

8.2 Binomial Heaps

Ernst Mayr, Harald Räcke

11. Apr. 2018

323/358
The number of nodes on level $\ell$ in tree $B_k$ is therefore

$$\binom{k-1}{\ell-1} + \binom{k-1}{\ell} = \binom{k}{\ell}$$
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 $\ell$-th level contains nodes that have $\ell$ 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 $\ell$-th level contains nodes that have $\ell$ 1’s in their label.
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 $\ell$-th level contains nodes that have $\ell$ 1’s in their label.
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 $\ell$-th level contains nodes that have $\ell$ 1’s in their label.
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 \texttt{x.left} and \texttt{x.right} point to the left and right sibling of \texttt{x} (if \texttt{x} does not have siblings then \texttt{x.left} = \texttt{x.right} = \texttt{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 $x$.left and $x$.right point to the left and right sibling of $x$ (if $x$ does not have siblings then $x$.left = $x$.right = $x$).
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 $x$.left and $x$.right point to the left and right sibling of $x$ (if $x$ does not have siblings then $x$.left = $x$.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 $x.left$ and $x.right$ point to the left and right sibling of $x$ (if $x$ does not have siblings then $x.left = x.right = x$).
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 $T$. 
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 $B_4$. 
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 $B_4$. 
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 $B_4$. 
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 $n$. 
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 $n$. 
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 $n$. 
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 $n = b_d b_{d-1} \ldots b_0$ denote binary representation of $n$.
- The heap contains tree $B_i$ iff $b_i = 1$.
- Hence, at most $\lceil \log n \rceil + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lceil \log n \rceil$.
- 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 $\lceil \log n \rceil + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lceil \log n \rceil$.
- The trees are stored in a single-linked list; ordered by dimension/size.
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.
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 $\lceil \log n \rceil + 1$ 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 $\lceil \log n \rceil + 1$ trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most $\lceil \log n \rceil$.
- The trees are stored in a single-linked list; ordered by dimension/size.
Binomial Heap

Properties of a heap with \( n \) keys:
- Let \( n = b_0 b_1 \ldots b_d \) denote binary representation of \( n \).
- The heap contains tree \( B_i \) iff \( b_i = 1 \).
- Hence, at most \( \lceil \log n \rceil + 1 \) trees.
- The minimum must be contained in one of the roots.
- The height of the largest tree is at most \( \lceil \log n \rceil \).
- 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.
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.
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.
8.2 Binomial Heaps

$S_1. \text{merge}(S_2)$:

- Analogous to binary addition.
- Time is proportional to the number of trees in both heaps.
- Time: $\Theta(\log n)$. 
8.2 Binomial Heaps

\textit{S}_1. \texttt{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

\[ S_1. \text{merge}(S_2): \]

- Analogous to binary addition.
- Time is proportional to the number of trees in both heaps.
- Time: \( O(\log n) \).
8.2 Binomial Heaps

All other operations can be reduced to merge().

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

- Create a new heap \( S' \) that contains just the element \( x \).
- Execute \( S. \text{merge}(S') \).
- Time: \( \Theta(\log n) \).
All other operations can be reduced to merge().

**S. insert(\(x\))**:  
- Create a new heap \(S'\) that contains just the element \(x\).  
- Execute \(S. \text{merge}(S')\).  
- Time: \(O(\log n)\).
8.2 Binomial Heaps

All other operations can be reduced to \texttt{merge}().

\textbf{\texttt{S}. insert}(x):

\begin{itemize}
  \item Create a new heap \texttt{S’} that contains just the element \texttt{x}.
  \item Execute \texttt{S. merge(S’)}.
  \item Time: \(\Theta(\log n)\).
\end{itemize}
8.2 Binomial Heaps

S. minimum():

- Find the minimum key-value among all roots.
- Time: $\Theta(\log n)$. 
8.2 Binomial Heaps

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

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

S. delete-min():

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

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

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

S. delete-min():

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

\textbf{S. delete-min()}: 

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

\textbf{Time}: $O(\log n)$. 

8.2 Binomial Heaps

\textbf{S. delete-min():}

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

8.2 Binomial Heaps 11. Apr. 2018
Ernst Mayr, Harald Räcke
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: \( \Theta(\log n) \) since the trees have height \( \Theta(\log n) \).
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: $O(\log n)$ since the trees have height $O(\log n)$. 
8.2 Binomial Heaps

\textit{S. decrease-key(\textit{handle} \textit{h})}: 

\begin{itemize}
  \item Decrease the key of the element pointed to by \textit{h}.
  \item Bubble the element up in the tree until the heap property is fulfilled.
  \item Time: $O(\log n)$ since the trees have height $O(\log n)$.
\end{itemize}
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: $O(\log n)$ since the trees have height $O(\log n)$. 
8.2 Binomial Heaps

\textit{S. delete(handle $h$)}:

\begin{itemize}
  \item Execute $S. \text{decrease-key}(h, -\infty)$.
  \item Execute $S. \text{delete-min}()$.
  \item Time: $\mathcal{O}(\log n)$.
\end{itemize}
8.2 Binomial Heaps

$S. \text{delete}(\text{handle } h)$:

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

\( \textit{S. delete(handle } h) : \)

- Execute \( \textit{S. decrease-key}(h, -\infty) \).
- Execute \( \textit{S. delete-min}() \).

\( \text{Time: } \Theta(\log n) \).
8.2 Binomial Heaps

$S$. delete(handle $h$):

- Execute $S$. decrease-key($h$, $-\infty$).
- Execute $S$. delete-min().
- Time: $\mathcal{O}(\log n)$. 
8.3 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 $x.\text{degree}$. Note that this can be updated in constant time when adding a child to $x$.

▶ Every node stores a boolean value $x.\text{marked}$ 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 $\Phi(S) = 5 + 2 \cdot 3 = 11$. 
We assume that one unit of potential can pay for a constant amount of work, where the constant is chosen “big enough” (to take care of the constants that occur).

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

S. minimum()

- Access through the min-pointer.
- Actual cost $\Theta(1)$.
- No change in potential.
- Amortized cost $\Theta(1)$. 
8.3 Fibonacci Heaps

S. merge($S'$)

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

Running time:

- Actual cost $O(1)$.
- No change in potential.
- Hence, amortized cost is $O(1)$. 
8.3 Fibonacci Heaps

S. merge($S'$)

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

Running time:

- Actual cost $O(1)$. 
8.3 Fibonacci Heaps

S. merge($S'$)

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

Running time:

▶ Actual cost $O(1)$.
▶ No change in potential.
8.3 Fibonacci Heaps

S. merge($S'$)

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

Running time:

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

\textbf{S. insert}(x)

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

\[
\begin{array}{c}
\text{min} \\
7 \\
\text{min} \\
3 \\
\text{min} \\
23 \\
\text{min} \\
24 \\
\text{min} \\
17 \\
\end{array}
\]

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. insert($x$)

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

\textbf{S. insert}(x)

\begin{itemize}
\item Create a new tree containing \(x\).
\item Insert \(x\) into the root-list.
\item Update min-pointer, if necessary.
\end{itemize}

\textbf{Running time:}

\begin{itemize}
\item Actual cost \(\mathcal{O}(1)\).
\item Change in potential is \(+1\).
\item Amortized cost is \(c + \mathcal{O}(1) = \mathcal{O}(1)\).
\end{itemize}
8.3 Fibonacci Heaps

\( S. \text{delete-min}(x) \)

\[ \text{Time} = \mathcal{O}(\log n) \]

\[ \text{Consolidate root-list so that no roots have the same degree. Time } t \cdot \mathcal{O}(1) \text{ (see next slide).} \]
8.3 Fibonacci Heaps

S. delete-min(x)

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

S. delete-min(x)

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

![Diagram of Fibonacci heap with nodes and edges indicating tree structure.](image)
8.3 Fibonacci Heaps

S. delete-min(x)

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

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

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

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

![Diagram of Fibonacci Heaps]

- **min**: 7
- **18**: 39, 41
- **52**: 44
- **23**: 24
- **17**: 30
8.3 Fibonacci Heaps

Consolidate:
Consolidate:
8.3 Fibonacci Heaps

Consolidate:

![Diagram showing consolidation process in a Fibonacci heap](image-url)
8.3 Fibonacci Heaps

Consolidate:

![Diagram of Fibonacci Heaps]

- **current**
- **min**

The diagram illustrates the consolidation process in a Fibonacci heap, with nodes representing elements and arrows indicating connections. The current element is highlighted, and the minimum element is indicated by the label 'min'.
8.3 Fibonacci Heaps

Consolidate:
8.3 Fibonacci Heaps

Consolidate:

![Diagram of Fibonacci Heaps]

- Min: 7
- Current: 18
- Values: 7, 18, 52, 24, 46, 26, 35, 23, 17, 30, 18, 39, 41, 44, 52, 18, 39, 41, 44, 18, 39, 41, 44, 24, 46, 26, 35, 7, 52, 17, 30
- Array: [0, 1, 2, 3]
- Elements: current

Ernst Mayr, Harald Räcke
8.3 Fibonacci Heaps

Consolidate:

[Diagram showing a Fibonacci heap with nodes and links, including a min and current pointer.]
8.3 Fibonacci Heaps

Consolidate:
8.3 Fibonacci Heaps

Consolidate:
8.3 Fibonacci Heaps

Consolidate:

```
min    current
   7   18   23   24   17
  52  41  39   46
  44  26  46
  35
```

```
8.3 Fibonacci Heaps

Consolidate:
8.3 Fibonacci Heaps

Consolidate:
8.3 Fibonacci Heaps

Consolidate:

```plaintext
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
min -> current
7  18  23  24  17
52 41 39 26 30
44 46 35
```

Ernst Mayr, Harald Räcke

11. Apr. 2018

347/358
8.3 Fibonacci Heaps

Consolidate:

![Diagram of Fibonacci Heaps]

- Current node: 7
- Minimum node: 35
- Nodes: 7, 18, 23, 17, 52, 24, 46, 26, 44, 39
- Structure: Tree with branches and pointers

0 1 2 3

Nodes are connected by arrows indicating the heap structure.
8.3 Fibonacci Heaps

Consolidate:
8.3 Fibonacci Heaps

Consolidate:

\[
\begin{array}{c}
\text{current} \\
\text{min}
\end{array}
\]
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 $O(1) \cdot (D_n + t)$.
  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 $O(1) \cdot (D_n + t)$.

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()

- $t' \leq D_n + 1$ as degrees are different after consolidating.
- Therefore $\Delta \Phi \leq D_n + 1 - t$;
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 $O(1) \cdot (D_n + t)$. 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.
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 $O(1) \cdot (D_n + t)$.
  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
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 $O(1) \cdot (D_n + t)$.

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

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

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

$$c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1) \leq (c_1 + c)D_n + (c_1 - c)t + c \leq 2c(D_n + 1)$$
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 $O(1) \cdot (D_n + t)$.
  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

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

$$\leq (c_1 + c)D_n + (c_1 - c)t + c \leq 2c(D_n + 1) \leq O(D_n)$$
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 $O(1) \cdot (D_n + t)$.
  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
  \[ c_1 \cdot (D_n + t) - c \cdot (t - D_n - 1) \leq (c_1 + c)D_n + (c_1 - c)t + c \leq 2c(D_n + 1) \leq O(D_n) \]
  for $c \geq c_1$.
8.3 Fibonacci Heaps

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

If we do not have delete or decrease-key operations then $D_n \leq \log n$. 
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 $D_n \leq \log n$. 
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: \texttt{decrease-key}(\texttt{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.
Case 1: decrease-key does not violate heap-property

- Just decrease the key-value of element referenced by $h$. Nothing else to do.
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: \texttt{decrease-key}(\texttt{handle} \ h, \nu)

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).
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\((\text{handle } h, \nu)\)

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.
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\((\text{handle } h, v)\)

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.
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.
**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\((\text{handle } h, v)\)

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\((\text{handle } h, v)\)

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

- Decrease key-value of element \(x\) reference by \(h\).
- Cut the parent edge of \(x\), and make \(x\) into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.
Fibonacci Heaps: decrease-key(handle \( h, v \))

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

- Decrease key-value of element \( x \) reference by \( h \).
- Cut the parent edge of \( x \), and make \( x \) into a root.
- Adjust min-pointers, if necessary.
- Execute the following:
  
  \[
  p \leftarrow \text{parent}[x]; \\
  \text{while} \; (p \; \text{is marked}) \; \\
  pp \leftarrow \text{parent}[p]; \\
  \text{cut of } p; \; \text{make it into a root}; \; \text{unmark it}; \\
  p \leftarrow pp; \\
  \text{if } p \; \text{is unmarked and not a root mark it};
  \]
Fibonacci Heaps: decrease-key (handle $h, v$)

**Actual cost:**
- Constant cost for decreasing the value.
- Constant cost for each of $\ell$ 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.
- $\Delta \Phi \leq \ell + 2$.
- Amortized cost is at most $c_2 (\ell + 1) + c (4 - \ell) \leq (c_2 - c) \ell + 4c + c_2 = O(1)$, if $c_2 \geq c$. 

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

Actual cost:
- Constant cost for decreasing the value.
- Constant cost for each of $\ell$ 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.
- Amortized cost is at most $O(1)$.
Fibonacci Heaps: decrease-key(handle \(h, v\))

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of \(\ell\) 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.
- \(\Delta\Phi \leq \ell + 2\) (\(-\ell + 2\) is a correction term).
- Amortized cost is at most

\[
\text{Amortized cost} = c_2 (\ell + 1) + c (4 - \ell) \\
\leq (c_2 - c) \ell + 4c + c_2 = O(1),
\]
Fibonacci Heaps: decrease-key(handle $h$, $v$)

Actual cost:
- Constant cost for decreasing the value.
- Constant cost for each of $\ell$ 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.
- Amortized cost is at most $c_2 (\ell + 1) + c(4 - \ell) \leq (c_2 - c)\ell + 4c + c_2 = O(1)$, if $c \geq c_2$. 

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

**Actual cost:**

▷ Constant cost for decreasing the value.
▷ Constant cost for each of $\ell$ 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 constant times the number of operations, meaning
$\Phi(t') \leq \Phi(t)$. 

8.3 Fibonacci Heaps

Ernst Mayr, Harald Räcke
Fibonacci Heaps: \texttt{decrease-key}(\textit{handle} \ h, v)

**Actual cost:**

- Constant cost for decreasing the value.
- Constant cost for each of $\ell$ 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 $c_2 \cdot (\ell + 1) + c(4 - \ell)$.
Fibonacci Heaps: \text{decrease-key}(\text{handle } h, v)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of \( \ell \) 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
Fibonacci Heaps: decrease-key(handle $h$, $v$)

Actual cost:

- Constant cost for decreasing the value.
- Constant cost for each of $\ell$ 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

$$c_2(\ell+1) + c(4-\ell) \leq (c_2 - c)\ell + 4c + c_2 = \Theta(1),$$

if $c \geq c_2$. 
Fibonacci Heaps: \textit{decrease-key}\((\text{handle } h, v)\)

**Actual cost:**

- Constant cost for decreasing the value.
- Constant cost for each of \(\ell\) 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

\[
c_2(\ell + 1) + c(4 - \ell) \leq (c_2 - c)\ell + 4c + c_2 = \Theta(1),
\]

if \(c \geq c_2\).
Fibonacci Heaps: \texttt{decrease-key}(\text{handle } h, v)

**Actual cost:**
- Constant cost for decreasing the value.
- Constant cost for each of $\ell$ 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
\[
c_2(\ell + 1) + c(4 - \ell) \leq (c_2 - c)\ell + 4c + c_2 = \Theta(1),
\] 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 $\ell$ 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 $c_2(\ell + 1) + c(4 - \ell) \leq (c_2 - c)\ell + 4c + c_2 = \Theta(1)$, if $c \geq c_2$. 
Delete node

\[ H. \text{delete}(x): \]

- decrease value of \( x \) to \(-\infty\).
- delete-min.

Amortized cost: \( \Theta(D_n) \)

- \( \Theta(1) \) for decrease-key.
- \( \Theta(D_n) \) for delete-min.
Lemma 1

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

Proof

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

Proof

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

Proof

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

Let $s_k$ be the minimum possible size of a sub-tree rooted at a node of degree $k$ that can occur in a Fibonacci heap.
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 $y_1, \ldots, y_k$ be its children.

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

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

$$\geq 2 + \sum_{i=2}^{k} s_{i-2}$$
8.3 Fibonacci Heaps

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

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

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

$$\geq 2 + \sum_{i=2}^{k} s_{i-2}$$

$$= 2 + \sum_{i=0}^{k-2} s_i$$
8.3 Fibonacci Heaps

Definition 2
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.
k=0: \quad 1 = F_0 \geq \Phi^0 = 1
k=1: \quad 2 = F_1 \geq \Phi^1 \approx 1.61
k-2,k-1 \rightarrow k: \quad F_k = F_{k-1} + F_{k-2} \geq \Phi^{k-1} + \Phi^{k-2} = \Phi^{k-2}(\Phi + 1) = \Phi^k

k=2: \quad 3 = F_2 = 2 + 1 = 2 + F_0
k-1 \rightarrow k: \quad F_k = F_{k-1} + F_{k-2} = 2 + \sum_{i=0}^{k-3} F_i + F_{k-2} = 2 + \sum_{i=0}^{k-2} F_i