Binomial and Fibonacci Heaps
A **mergeable priority queue** $Q$ stores a set of elements with keys, and supports the following operations:

- `make-heap()`: create an empty heap.
- `minimum(Q)`: return the element of $Q$ with smallest key.
- `insert(Q, x)`: add $x$ to $Q$.
- `extract-min(Q)`: remove and return the element of $Q$ with smallest key.
- `decrease-key(Q, x, k)`: decrease the key of $x$ to $k$.
- `merge(Q_1, Q_2)`: merge two priority queues into one.
The binomial tree $B_0$ consists of a single node.
The binomial tree $B_k$ consists of two $B_{k-1}$ trees whose roots are linked.
The binomial tree $B_0$ consists of a single node.
The binomial tree $B_k$ consists of two $B_{k-1}$ trees whose roots are linked.
- $B_k$ have $2^k$ nodes.
- The **degree** of $B_k$ (number of children of the root) is $k$.
- Removing the root of $B_k$ gives a forest containing $B_0, \ldots, B_{k-1}$. 

![Diagram of binomial heaps](image)
Properties

- $B_k$ have $2^k$ nodes.
- The **degree** of $B_k$ (number of children of the root) is $k$.
- Removing the root of $B_k$ gives a forest containing $B_0, \ldots, B_{k-1}$.

![Diagram of binomial trees](image-url)
A binomial heap is a forest of binomial trees.

- Nodes of the trees contain the elements of the queue.
- For every node \( x \), the key of \( x \) is greater or equal to the key of the parent of \( x \) (heap property).
- For every \( k \geq 0 \), there is at most one tree of degree \( k \).
Binomial heap

- The roots of the trees are stored in a linked list sorted in increasing order of degrees.
- The heap stores a pointer to the root containing the element with smallest key.
The **linking** of two trees of same degree in a binomial heap is making the root of one tree the leftmost child of the other tree (maintaining the heap order).
The **linking** of two trees of same degree in a binomial heap is making the root of one tree the leftmost child of the other tree (maintaining the heap order).
Merge($H_1, H_2$)

- Merge the list of trees in $H_1$ and $H_2$.
- Scan the trees in increasing order of degrees. When two trees of equal degrees are encountered, link the trees.
- Create a pointer to the element with smallest key.
Merge($H_1$, $H_2$)

- Merge the list of trees in $H_1$ and $H_2$.
- Scan the trees in increasing order of degrees. When two trees of equal degrees are encountered, link the trees.
- Create a pointer to the element with smallest key.
Merge($H_1, H_2$)

- Merge the list of trees in $H_1$ and $H_2$.
- Scan the trees in increasing order of degrees. When two trees of equal degrees are encountered, link the trees.
- Create a pointer to the element with smallest key.
Merge($H_1, H_2$)

- Merge the list of trees in $H_1$ and $H_2$.
- Scan the trees in increasing order of degrees. When two trees of equal degrees are encountered, link the trees.
- Create a pointer to the element with smallest key.

![Diagram showing two binary trees and a link between two nodes]
Merge($H_1$, $H_2$)

- Merge the list of trees in $H_1$ and $H_2$.
- Scan the trees in increasing order of degrees. When two trees of equal degrees are encountered, link the trees.
- Create a pointer to the element with smallest key.

![Diagram of merging trees]

Binomial and Fibonacci Heaps
Binomial heap

- minimum($H$): trivial.
- insert($H, k$): create a new heap with a single tree $B_0$ containing $k$, and merge it with $H$.
- extract-min($H$): Delete the root of the tree containing the smallest key. Merge the forest of the subtrees with $H$.

- decrease-key($H, x, k$): Move $x$ up its tree. Update minimum pointer if needed.
Binomial heap

- minimum\((H)\): trivial.
- insert\((H, k)\): create a new heap with a single tree \(B_0\) containing \(k\), and merge it with \(H\).
- extract-min\((H)\): Delete the root of the tree containing the smallest key. Merge the forest of the subtrees with \(H\).
- decrease-key\((H, x, k)\): Move \(x\) up its tree. Update minimum pointer if needed.
Binomial heap

- `minimum(H)`: trivial.
- `insert(H, k)`: create a new heap with a single tree $B_0$ containing $k$, and merge it with $H$.
- `extract-min(H)`: Delete the root of the tree containing the smallest key. Merge the forest of the subtrees with $H$.

![Binomial heap diagram]

- `decrease-key(H, x, k)`: Move $x$ up its tree. Update minimum pointer if needed.
Binomial heap

- **minimum**($H$): trivial.
- **insert**($H, k$): create a new heap with a single tree $B_0$ containing $k$, and merge it with $H$.
- **extract-min**($H$): Delete the root of the tree containing the smallest key. Merge the forest of the subtrees with $H$.
- **decrease-key**($H, x, k$): Move $x$ up its tree. Update minimum pointer if needed.
Binomial heap

- **minimum**(*H*): trivial.
- **insert**(*H*, *k*): create a new heap with a single tree *B*₀ containing *k*, and merge it with *H*.
- **extract-min**(*H*): Delete the root of the tree containing the smallest key. Merge the forest of the subtrees with *H*.

```
7
3
6
9
3 2
5
8
5 4
```

- **decrease-key**(*H*, *x*, *k*): Move *x* up its tree. Update minimum pointer if needed.
Binomial heap

- minimum($H$): trivial.
- insert($H$, $k$): create a new heap with a single tree $B_0$ containing $k$, and merge it with $H$.
- extract-min($H$): Delete the root of the tree containing the smallest key. Merge the forest of the subtrees with $H$.
- decrease-key($H$, $x$, $k$): Move $x$ up its tree. Update minimum pointer if needed.
Merging two heaps is linear in the number of trees in heaps (each linking decreases the number of trees by 1).

In a binomial heaps with $n$ elements, the number of trees is $\leq \lceil \log_2 n \rceil$.

A minimum operation takes $O(1)$ time. Other operations take $O(\log n)$ time.
A Fibonacci heap is a forest of trees. Each node contains an element. The heap stores a **minimum pointer** (pointer to element with smallest key).

- For every node $x$, the key at $x$ is greater or equal to the key at the parent of $x$.
- A node $x$ is **marked** if it has lost a child.
Amortized analysis

We will analyse the amortized time complexity of each operation using the accounting method.

We keep the following invariants:

- Each tree stores one dollar.
- Each marked node stores two dollars.
The operation is trivial.

Analysis:
- Cost: $1.
- Charge: $1.
- We use the $1 of the charge to pay the cost.
Add a new tree with one unmarked node containing $k$ to $H$. Update the minimum pointer if needed.

Analysis:
- Cost: $1$.
- Charge: $2$.
- We use $1$ from the charge pay the cost. The second dollar is placed in the new tree.
merge($H_1$, $H_2$)

- Concatenate the lists of $H_1$ and $H_2$.
- Compute the minimum pointer.

Cost: $1$.
Charge: $1$.
We use the dollar of the charge to pay the cost.
merge($H_1, H_2$)

- Concatenate the lists of $H_1$ and $H_2$.
- Compute the minimum pointer.

Cost: $1$.
Charge: $1$.

We use the dollar of the charge to pay the cost.
Let $x$ be the node containing the smallest key. Remove $x$, and add the subtrees of the children of $x$ to the list of trees of $H$. We need to scan the new trees in order to update the minimum pointer, so the time of extract-min may be large. We may as well do some maintenance work to reduce the number of trees.
Let $x$ be the node containing the smallest key.
Remove $x$, and add the subtrees of the children of $x$ to the list of trees of $H$.
We need to scan the new trees in order to update the minimum pointer, so the time of extract-min may be large. We may as well do some maintenance work to reduce the number of trees.
Sort the trees according to their degrees.
Scan the trees in increasing order of degrees. When two trees of equal degrees are encountered, link the trees. Unmark the root that becomes a child if it was marked.
Create a pointer to the element with smallest key.
Sort the trees according to their degrees.
Scan the trees in increasing order of degrees. When two trees of equal degrees are encountered, link the trees. Unmark the root that becomes a child if it was marked.
Create a pointer to the element with smallest key.
Sort the trees according to their degrees.

Scan the trees in increasing order of degrees. When two trees of equal degrees are encountered, link the trees. Unmark the root that becomes a child if it was marked.

Create a pointer to the element with smallest key.
Sort the trees according to their degrees.
Scan the trees in increasing order of degrees. When two trees of equal degrees are encountered, link the trees. Unmark the root that becomes a child if it was marked.
Create a pointer to the element with smallest key.
extract-min($H$)

- Sort the trees according to their degrees.
- Scan the trees in increasing order of degrees. When two trees of equal degrees are encountered, link the trees. Unmark the root that becomes a child if it was marked.
- Create a pointer to the element with smallest key.
Analysis of extract-min

- **Cost**: \( k + D(n) \) where
  - \( k \) = number of trees in the heap before the extract-min
  - \( D(n) = 1.45 \log n \) is an upper bound on the degree of a tree in a Fibonacci heap

The sort involves \( \leq k + D(n) \) trees with keys from \( \{0, \ldots, D(n)\} \).

- **Charge**: \( 2D(n) + 1 \).
- We use \( D(n) \) dollars of the charge to pay for \( D(n) \) term in the cost.
- We use the dollars in the trees (before the extract-min) to pay for the \( k \) term in the cost.
- We use \( \leq D(n) + 1 \) dollars of the charge to place dollars in the trees after the linking stage is completed.
decrease-key($H, x, k$)

- If $k$ is smaller than the key of the parent of $x$, cut the edge between $x$ and its parent and unmark $x$.
- Continue cutting the marked ancestors of $x$, until reaching an unmarked ancestor $y$. Unmark the cut nodes, and mark $y$.

```
decrease-key(H,x,4)
```
If $k$ is smaller than the key of the parent of $x$, cut the edge between $x$ and its parent and unmark $x$.

Continue cutting the marked ancestors of $x$, until reaching an unmarked ancestor $y$. Unmark the cut nodes, and mark $y$.

decrease-key(H, x, 4)
decrease-key($H, x, k$)

- If $k$ is smaller than the key of the parent of $x$, cut the edge between $x$ and its parent and unmark $x$.
- Continue cutting the marked ancestors of $x$, until reaching an unmarked ancestor $y$. Unmark the cut nodes, and mark $y$. 

![Diagram of a tree with nodes labeled and colored yellow indicating marked nodes.](image-url)
decrease-key($H, x, k$)

- If $k$ is smaller than the key of the parent of $x$, cut the edge between $x$ and its parent and unmark $x$.
- Continue cutting the marked ancestors of $x$, until reaching an unmarked ancestor $y$. Unmark the cut nodes, and mark $y$. 

![Tree diagram]
decrease-key($H, x, k$)

- If $k$ is smaller than the key of the parent of $x$, cut the edge between $x$ and its parent and unmark $x$.
- Continue cutting the marked ancestors of $x$, until reaching an unmarked ancestor $y$. Unmark the cut nodes, and mark $y$. 

![Diagram](image)
Analysis of decrease-key

- Cost: $1 + k$, where $k$ is the number of marked nodes encountered.
- Charge: $4$.
- We use $1$ of the charge to pay for the handling of $x$.
- If $x$ was cut, we put $1$ of the charge in the new tree that contains $x$.
- For each ancestor $z$ of $x$ which is cut, we use $1$ of the two dollars stored at $z$ to pay for the handling of $z$. We put the second dollar in $z$ in the new tree whose root is $z$.
- If a node $y$ was marked, we put $2$ from the charge in $y$. 
Recall that

- A node can become a child of another node only by linking during extract-min operations.
- A node can lose children only during decrease-key operations.
- When a node loses a child, it becomes marked.
- When a marked node loses a child, it is cut from its parent.
Bounding $D(n)$

**Lemma**
Let $x$ be a node, and let $y_1, \ldots, y_k$ be the children of $x$ from right to left. Then, the degree of $y_i$ is $\geq \max(i - 2, 0)$.

**Proof.**
Suppose that $i \geq 3$. When $y_i$ was linked to $x$, $\deg(y_i) = \deg(x) \geq i - 1$. Since then, $y_i$ has lost at most one child (it would have been cut from $x$ if it had lost two children).
Corollary

A node $x$ with degree $k$ has at least $\phi^k$ descendants, where $\phi = (1 + \sqrt{5})/2$.

Proof.

Let $s_k$ be the minimum size of a subtree of a node of degree $k$ in a Fibonacci heap. Clearly, $s_0 = 1$ and $s_1 = 2$. By the lemma, $s_k \geq 1 + \sum_{i=1}^{k} s_{\max(i-2,0)} = 2 + \sum_{i=2}^{k} s_{i-2}$.
Bounding $D(n)$

Corollary

A node $x$ with degree $k$ has at least $\phi^k$ descendants, where $\phi = (1 + \sqrt{5})/2$.

Proof.

Let $F_0, F_1, \ldots$ be Fibonacci number ($F_0 = 0$, $F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$).

We will show the following:

1. $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$.
2. $F_{k+2} \geq \phi^k$.
3. $s_k \geq F_{k+2}$. 
Corollary

A node $x$ with degree $k$ has at least $\phi^k$ descendants, where $\phi = (1 + \sqrt{5})/2$.

Proof.

We show that $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$ by induction:

$$F_{k+2} = F_k + F_{k+1} = F_k + \left(1 + \sum_{i=0}^{k-1} F_i \right) = 1 + \sum_{i=0}^{k} F_i.$$
Corollary

A node $x$ with degree $k$ has at least $\phi^k$ descendants, where $\phi = (1 + \sqrt{5})/2$.

Proof.

We show that $F_{k+2} \geq \phi^k$ by induction:

$$F_{k+2} = F_{k+1} + F_k \geq \phi^{k-1} + \phi^{k-2} = \phi^{k-2}(\phi + 1) = \phi^{k-2} \cdot \phi^2 = \phi^k.$$
Corollary

A node $x$ with degree $k$ has at least $\phi^k$ descendants, where $\phi = (1 + \sqrt{5})/2$.

Proof.

We show that $s_k \geq F_{k+2}$ by induction:

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2}.$$
Corollary

A node $x$ with degree $k$ has at least $\phi^k$ descendants, where $
\phi = (1 + \sqrt{5})/2$.

Corollary

$D(n) = \log_\phi n \leq 1.45 \log n$. 
### Summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Binary heap</th>
<th>Binomial heap</th>
<th>Fibonacci heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst case</td>
<td>Worst case</td>
<td></td>
<td>Amortized</td>
</tr>
<tr>
<td>make-heap</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>minimum</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>extract-min</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>decrease-key</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>merge</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

- Fibonacci heap implements some of the priority queue operations faster than binary heap.
- In some applications, decrease-key is the most frequent operation (e.g., Dijkstra’s algorithm). Thus, a Fibonacci heap is very efficient for such applications.
**Dijkstra’s algorithm**

\[ \text{DIJKSTRA}(G, w, s) \]

1. **INITIALIZE** \((G, s)\)
2. Build priority queue \(Q\) on vertices (keys are \(d\) values)
3. **while** \(Q\) is not empty
   4. \(u \leftarrow \text{EXTRACTMIN}(Q)\)
   5. **foreach** \(v \in G.\text{Adj}[u]\)
      6. **if** \(u.d + w(u, v) < v.d\)
      7. **DECREASEKEY** \((Q, v, u.d + w(u, v))\)

Diagram:

```
  a 10  b
  |   1   |
 a  |     | b
|   10 |
  s 2 3 4 6
  |   5  |
 c 5 2 7
    |   7  |
     d
```
Dijkstra’s algorithm

\textbf{DIJKSTRA}(G, w, s)

1. \textbf{INITIALIZE}(G, s)
2. Build priority queue \( Q \) on vertices (keys are \( d \) values)
3. \textbf{while} \( Q \) is not empty
4. \( u \leftarrow \text{EXTRACTMIN}(Q) \)
5. \textbf{foreach} \( v \in G.\text{Adj}[u] \)
6. \textbf{if} \( u.d + w(u, v) < v.d \)
7. \( \text{DECREASEKEY}(Q, v, u.d + w(u, v)) \)

The algorithm performs:
- \(|V| \) insert operations and \(|V| \) extract-min operations.
- \( \leq |E| \) decrease-key operations.

The total time is
- \( O((V + E) \log V) \) when using a binary heap.
- \( O(E + V \log V) \) when using Fibonacci heap.