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

An addressable Priority Queue also supports:

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

Dijkstra's Shortest Path Algorithm

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: \( S.\text{build(); // build empty priority queue} \)
4: **for all** \( v \in V \setminus \{s\} \) **do**
5: \( v.\text{key} \leftarrow \infty; \)
6: \( h_v \leftarrow S.\text{insert}(v); \)
7: \( s.\text{key} \leftarrow 0; S.\text{insert}(s); \)
8: **while** \( S.\text{is-empty}() = \text{false} \) **do**
9: \( v \leftarrow S.\text{delete-min}(); \)
10: **for all** \( x \in V \) s.t. \( \{v, x\} \in E \) **do**
11: \( \text{if } x.\text{key} > v.\text{key} + d(v, x) \) **then**
12: \( S.\text{decrease-key}(h_x, v.\text{key} + d(v, x)); \)
13: \( x.\text{key} \leftarrow v.\text{key} + d(v, x); \)

Prim's Minimum Spanning Tree Algorithm

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: \( S.\text{build(); // build empty priority queue} \)
4: **for all** \( v \in V \setminus \{s\} \) **do**
5: \( v.\text{key} \leftarrow \infty; \)
6: \( h_v \leftarrow S.\text{insert}(v); \)
7: \( s.\text{key} \leftarrow 0; S.\text{insert}(s); \)
8: **while** \( S.\text{is-empty}() = \text{false} \) **do**
9: \( v \leftarrow S.\text{delete-min}(); \)
10: **for all** \( x \in V \) s.t. \( \{v, x\} \in E \) **do**
11: \( \text{if } x.\text{key} > d(v, x) \) **then**
12: \( S.\text{decrease-key}(h_x, d(v, x)); \)
13: \( x.\text{key} \leftarrow d(v, x); \)
14: \( x.\text{pred} \leftarrow v; \)
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?

8 Priority Queues

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

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

<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>$n \log n$</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Fibonacci heaps only give an amortized guarantee. The standard version of binary heaps is not addressable. Hence, it does not support a delete.

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 $O(1)$.
- is-empty(): check whether root-pointer is null. Time $O(1)$.

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

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).

Build Heap

We can build a heap in linear time:

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

8.1 Binary Heaps

Ernst Mayr, Harald Räcke 321

8.2 Binomial Heaps

Ernst Mayr, Harald Räcke 322

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

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 $B_0$.

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.
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 \( x.\text{left} = x.\text{right} = x \)).

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 \( B_4 \).

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 \( \lfloor \log n \rfloor + 1 \) 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.

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

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

**S. minimum():**
- Find the minimum key-value among all roots.
- Time: $O(\log n)$.

**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 $O(\log n)$ trees).
- Compute $S$.merge($S'$).
- Time: $O(\log n)$.

**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

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

\[ \quad \text{ Execute } S. \text{ decrease-key}(h, -\infty). \]
\[ \quad \text{ Execute } S. \text{ delete-min}. \]
\[ \quad \text{ Time: } O(\log n). \]

8.3 Fibonacci Heaps

Collection of trees that fulfill the heap property.

Structure is much more relaxed than binomial heaps.

The potential function:

\[ t(S) \] denotes the number of trees in the heap.
\[ m(S) \] denotes the number of marked nodes.

We use the potential function \( \Phi(S) = t(S) + 2m(S) \).

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

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

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

---

8.3 Fibonacci Heaps

S. minimum()

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

---

8.3 Fibonacci Heaps

S. merge($S'$)

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

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

Running time:

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

---

8.3 Fibonacci Heaps

S. insert($x$)

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

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

Running time:

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

\[ D(\text{min}) \] is the number of children of the node that stores the minimum.

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

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

**Consolidate:**

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

**Consolidate root-list so that no roots have the same degree.**

Time \( t \cdot \Theta(1) \) (see next slide).
8.3 Fibonacci Heaps

Consolidate:

current

min → 7

now 0 1 2 3

23 26 29 35 40 46 52

now 0 1 2 3

23 26 29 35 40 46 52

now 0 1 2 3

23 26 29 35 40 46 52

now 0 1 2 3
8.3 Fibonacci Heaps

**Consolidate:**

- 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 $D_n \leq \log n$.

---

**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.

**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 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.
Delete node

\( H. \, \text{delete}(x) : \)
- decrease value of \( x \) to \(-\infty\).
- delete-min.

Amortized cost: \( O(D_n) \)
- \( O(1) \) for decrease-key.
- \( O(D_n) \) for delete-min.

8.3 Fibonacci Heaps

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

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

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

8.3 Fibonacci Heaps

Proof
- When \( y_1 \) 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 \( y_1 \) has lost at most one child.
- Therefore, \( \text{degree}(y_i) \geq i - 2 \).

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:

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

\[ \phi = \frac{1}{2} (1 + \sqrt{5}) \text{ denotes the golden ratio.} \]
Note that \( \phi^2 = 1 + \phi. \)

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.

Prioriry Queues

Bibliography


Binary heaps are covered in [CLRS90] in combination with the heapsort algorithm in Chapter 6. Fibonacci heaps are covered in detail in Chapter 19. Problem 19.2 in this chapter introduces Binomial heaps.

Chapter 6 in [MS08] covers Priority Queues. Chapter 6.2.2 discusses Fibonacci heaps. Binomial heaps are dealt with in Exercise 6.11.