Fibonacci Heaps, I

The Fibonacci heap is a data structure implementing the priority queue abstract data type, just like the ordinary
heap but more complicated and asymptotically faster for
some operations. We first introduce binomial trees, which
are special heap-ordered trees, and then describe the Fi-
bonacci heap as a collections of heap-ordered trees.

Binomial Trees. The binomial tree of height \( h \) is a tree
obtained from two binomial trees of height \( h - 1 \) by linking
the root of one to the root of the other. The binomial tree
of height 0 consists of a single node. Binomial trees of
heights up to 4 are shown in Figure 63. The degree of
a node is its number of children. From Table 4 we see
that the degree of the root is equal to the height and the
number of nodes increases exponentially with the height.
A binomial tree with root degree \( h \) thus has \( 2^h \) nodes. The
root has the largest degree of any node in the binomial
tree. This implies that every node in a binomial tree with
\( n \) nodes has degree at most \( \log_2 n \).

| height | 0 | 1 | 2 | 3 | 4 | \ldots | \( h \) |
|--------|---|---|---|---|---|--------|
| degree | 0 | 1 | 2 | 3 | 4 | \ldots | \( h \) |
| size   | 1 | 2 | 4 | 8 | 16 | \ldots | \( 2^h \) |

Table 4: The degree of the root and the number of nodes as a function of the height.

Binomial Heaps. The size of a single binomial tree is a
power of 2. If we need \( n \) nodes, \( n \) not a power of 2, we
can use a small number of binomial trees, choosing them
so that their sizes add up to \( n \). To determine how many
trees we need and of what size, we write \( n = \sum_{i \geq 0} n_i 2^i \),
letting \( n_i \) be the \( (i+1) \)-st bit in the binary notation of
\( n \). To store \( n \) items, we use a binomial tree of size \( 2^i \)
for each \( n_i = 1 \). The total number of binomial trees is
thus the number of 1’s in the binary notation of \( n \), which
is at most \( \log_2 (n + 1) \). The collection is referred to as a
binomial heap. The items in each binomial tree are stored
in heap-order, which means that the item stored in a node
has higher priority (smaller key) than the items stored in
its children. There is no specific relationship between the
items stored in different binomial trees. The item with
minimum key is thus stored in one of the logarithmically
many roots, but it is not clear in which one.

An example is shown in Figure 64 where 1110 = 10112
items are stored in three binomial trees of size 8, 2, and 1.
In order to add a new item to the set, we create a new bino-
mial tree of size 1 and we successively link binomial trees
as dictated by the rules of adding 1 to the binary notation
of \( n \). In the example we get 10112 + 12 = 11002. The
new collection thus consists of two binomial trees of size
8 and 4. The size-8-tree is the old one, and the size-4-tree
is obtained by first linking the two size-1-trees and then
linking the resulting size-2-tree to the old size-2-tree. All
this is illustrated in Figure 64.

Fibonacci Heaps. A Fibonacci heap is a collection of heap-ordered trees. Ideally, we would like it to be a collection of binomial trees, but we need more flexibility to make it as efficient as possible. A Fibonacci heap supports a variety of operations, including the standard priority queue functions. All operations, except for deletions, take constant amortized time. We list all operations together with the amortized time they take.

\[
\begin{align*}
H &= \text{MAKEHEAP} & O(1) \\
\text{INSERT}(H, x) &= O(1) \\
x &= \text{MINIMUM}(H) & O(1) \\
H &= \text{MELD}(H_1, H_2) & O(1) \\
\text{DELETEMIN}(H) &= O(\log_2 n) \\
\text{DECREASE}(H, \nu, \Delta) &= O(\log_2 n) \\
\text{DECREASE}(H, \nu) &= O(1)
\end{align*}
\]

The \text{DECREASE} operation replaces the item with key \(x\) stored in the node \(\nu\) by \(x - \Delta\), with \(\Delta \geq 0\). We will see the use of this operation in computing minimum spanning trees and shortest paths in later sections that deal with graph algorithms.

Implementation. It will be important to understand how exactly the nodes of a Fibonacci heap are connected by pointers. Siblings are organized in doubly-linked cyclic lists, and each node stores a pointer to its parent and a pointer to one of its children, as shown in Figure 65. Besides the pointers, each node stores a key, its degree, and a bit that can be used to mark or unmark the node. The roots of the heap-ordered trees are doubly-linked in a cycle, and there is an explicit pointer to the root that stores the item with the minimum key. Figure 66 illustrates a few basic operations that we can perform on a Fibonacci heap. Given two heap-ordered trees, we link them by making the root with the bigger key the child of the other root. To unlink a heap-ordered tree or subtree, we remove its root from the doubly-linked cycle. Finally, to merge two cycles we cut both open and connect them at their ends. These three operations take only constant time each.

Potential Function. Consider a sequence of \(n\) operations applied to an initially empty Fibonacci heap. We define \(t_i\) as the number of roots in the root cycle, \(m_i\) as the number of marked nodes, and \(\Phi_i = t_i + 2m_i\) as the potential after the \(i\)-th operation. When we deal with a collection of Fibonacci heaps, we define its potential as the sum of individual potentials. The initial Fibonacci heap is empty, so \(\Phi_0 = 0\). As usual, we let \(c_i\) be the actual cost and \(a_i = c_i + \Phi_i - \Phi_{i-1}\) the amortized cost of the \(i\)-th operation. Since \(\Phi_0 = 0\) and \(\Phi_i \geq 0\), for all \(i\), the actual cost is less than the amortized cost:

\[
\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i = t_n + 2m_n + \sum_{i=1}^{n} c_i.
\]

Transparent Operations. For some of the operations it is fairly easy to compute the amortized cost.

To create an empty Fibonacci heap with \text{MAKEHEAP}, we return a \text{NULL} pointer. This operation does not change the potential and takes amortized and actual cost \(a_i = c_i = 1\).

Function \text{MINIMUM} returns the minimum key and also does not change the potential. It takes amortized and actual cost \(a_i = c_i = 1\).

To meld two Fibonacci heaps \(H_1\) and \(H_2\) into a single Fibonacci heap \(H = \text{MELD}(H_1, H_2)\), we first merge the two root circles and then adjust the pointer to the minimum key. We have \(t_i(H) = t_{i-1}(H_1) + t_{i-1}(H_2)\) and

Figure 66: From left to right we link two trees, we unlink a tree, and we merge two cycles.
\[ m_i(H) = m_{i-1}(H_1) + m_{i-1}(H_2), \]
which implies that there is no change in potential. The amortized and actual cost is therefore \( a_i = c_i = 1. \)

Function \textsc{Insert} adds a key \( x \) to a Fibonacci heap \( H \) by first creating a new Fibonacci heap \( H_1 \) that stores only \( x \) and second melding the two Fibonacci heaps. The number of nodes in the root cycle increases by one, thus \( \Phi_i - \Phi_{i-1} = 1. \) The amortized cost is therefore \( a_i = c_1 + 1 = 2. \)

**DeleteMin.** Next, we consider the somewhat more interesting operation of deleting the minimum key. This is done in four steps.

\begin{itemize}
    \item **Step 1.** Remove the node with minimum key from the root cycle.
    \item **Step 2.** Merge the root cycle with the cycle of children of the removed node.
    \item **Step 3.** As long as there are two roots with the same degree link them.
    \item **Step 4.** Recompute the pointer to the minimum key.
\end{itemize}

For **Step 3**, we use a pointer array \( R \). Initially, \( R[i] = \text{NULL} \) for each \( i \). For each root \( r \) in the root cycle, we execute the following loop:

\[
i = r \rightarrow \text{degree};
\text{while } R[i] \neq \text{NULL} \text{ do}
    \ q = R[i]; \ R[i] = \text{NULL}; \ r = \text{LINK}(r, q); \ i++
\text{endwhile}
R[i] = r.
\]

To analyze the amortized cost for deleting the minimum, let \( D(n) \) be the maximum possible degree of any node in a Fibonacci heap of \( n \) nodes. The actual costs of **Steps** 1 to 4 are 1, 1, \( t_{i-1} \), and at most \( D(n) \), therefore \( c_i \leq D(n) + t_{i-1} + 2 \). Indeed, in **Step 3** we move all roots to the array \( R \) and we have fewer link operations than there are roots because each operation decreases the number of roots by one. The change in potential is \( \Phi_i - \Phi_{i-1} = t_i - t_{i-1} \leq D(n) - t_{i-1} \), because after linking the number of roots in the root cycle is at most the maximum degree, which is at most \( D(n) \). The amortized cost is therefore

\[
a_i = c_i + \Phi_i - \Phi_{i-1}
\leq 2D(n) + 2.
\]

We will prove shortly that the maximum possible degree \( D(n) \) is less than twice the binary logarithm of \( n + 1 \), which implies that function \textsc{DeleteMin} has logarithmic amortized cost.