

# Fibonacci Heaps

## 1 Basic Heaps

Let  $n$  be the total number of nodes and  $m$  the total number of edges in a graph  $\mathcal{G}$  which is represented by a heap  $h$ . Heaps are an implementation for priority queues which satisfy the following properties:



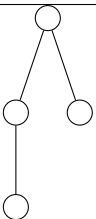
1. Must be a complete binary tree (giving us useful  $\Theta(\log n)$  properties)
2. Must satisfy the Heap Property: for every node  $v$  with children  $c$ ,  $\text{priority}(v) \leq \text{priority}(c)$ 
  - (a) Causes the element of least priority to be the root of the tree

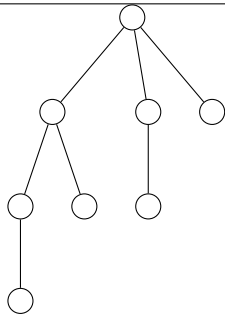
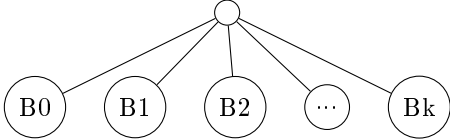
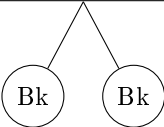
Basics heaps are pretty good, but the algorithm can be improved. A heap must support the following operations, with the noted standard and desired runtimes. Heaps can also be used to implement Prim's and Dijkstra's Algorithms, and the efficiency of the heap affects the runtime of the algorithm.

Operation	Standard runtime	Desired runtime
$\text{insert}(h, v)$	$\Theta(\log n)$	$\Theta(1)$
$\text{findMin}(h)$	$\Theta(1)$	$\Theta(\log n)$
$\text{removeMin}(h)$	$\Theta(\log n)$	$\Theta(\log n)$
$\text{decrementPriority}(h, v)$	$\Theta(\log n)$	$\Theta(1)$
Prim's or Dijkstra's Algorithm	$\Theta(m \log n)$	$\Theta(m + n \log n)$

## 2 Binomial Heaps

### 2.1 Binomial Trees

Binomial Tree Number	Graphic Representation
$B_0$	
$B_1$	
$B_2$	

Binomial Tree Number	Graphic Representation
$B_3$	
$B_{k+1}$	
$B_{k+1}$	

It is evident that  $|B_k| = 2^k$  nodes. We define the rank of a binomial tree as equal to the degree of the root; thus  $rank(B_k) = k$ .

## 2.2 Binomial Heaps

Binomial heaps are similar to log-structured merge trees in that we will combine multiple smaller data structures across various sizes, and merge them together when they get too large. A **binomial heap** is a forest of binomial trees, all satisfying the heap property, with at most one tree per rank. With  $n$  total elements, restricting the number of trees per rank to one forces the heap to contain  $\leq \log n$  trees. Thus, we can find the minimum priority element in  $\Theta(\log n)$  time. To preserve this property when adding new nodes, we define the following operations:

Operation	Worst-case runtime	Description
$\text{meld}(h, h')$	$\Theta(\log n)$	Combines two binomial heaps into a single heap. While there exists a rank $k$ such that we have more than one rank $k$ tree, merge two rank $k$ trees into one $B_{k+1}$ .
$\text{insert}(h, x)$	$\Theta(\log n)$	Create a $B_0$ with value $x$ , then run $\text{meld}$ .
$\text{findMin}(h)$	$\Theta(\log n)$	Check all the roots in the binomial heap and return the root with lowest value.
$\text{removeMin}(h)$	$\Theta(\log n)$	Run $\text{findMin}$ and remove the resultant root of rank $k$ , decomposing it into a new binomial heap of trees $B_0, B_1, \dots, B_{k-1}$ . Then run $\text{meld}$ with the original heap.

However, the binomial heap does not provide us with any advantages over our basic heap...unless we utilize amortized analysis. We must recognize that the worst-case runtime of these operations will occur very infrequently, as half the time when inserting there will be no existing  $B_0$ , a quarter of the time there will be no existing  $B_1$ , etc.

## 2.3 Amortized Analysis

In order to perform amortized analysis on binomial heaps, we must first make a few changes to its structure:

1. Allow multiple trees of same rank
2. Insert just adds a  $B_0$  and does not run meld
3. Meld is called at the beginning of findMin

Now it is trivial that insert runs in  $\Theta(1)$ .

**Theorem 1.** *Meld runs in amortized  $\Theta(1)$  in a modified binomial heap.*

*Proof.* By credit scheme.

Let the **meld credit invariant** be that each binomial tree always has a meld credit.

Operation	Credit Description
insert( $h, x$ )	Create a $B_0$ with value $x$ , then give that tree a meld credit.
removeMin( $h$ )	Already does $\log n$ work to find the correct root, so do another $\log n$ to give each tree a meld credit.
meld( $h, h'$ )	2 $B_k$ merge, so spend 1 of their meld credits for the work done during merging and pass the remaining onto the $B_{k+1}$ .

Thus, the extra work done during the insert and removeMin operations pay off during meld, keeping it amortized  $\Theta(1)$ .  $\square$

Any sequence of  $a$  inserts and  $b$  removeMins, starting from an empty heap, is  $\mathcal{O}(a + b \log n)$ .

## 3 Fibonacci Heaps

### 3.1 DecrementPriority

We have the insert runtime that we want, but we still haven't implemented all our desired operations yet. In order to develop a decrementPriority function, we will have to use a Fibonacci heap.

We introduce a function decrementPriority( $h, x, \delta$ ) which decreases the cost of  $x$  in  $h$  by  $\delta$ . Using an array or hashtable, we can look up  $x$  in  $\Theta(1)$ . However, swapping  $x$  with its parents takes  $\Theta(\log n)$  like in a basic heap. We want decrementPriority to run in amortized  $\Theta(1)$ , so we will instead delink  $x$  and all its children from  $h$ , creating a new tree. We will also spend a unit of extra time to give that tree a credit.




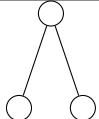
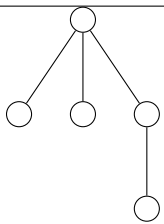
To prevent trees from becoming too flat, as soon as the second child of a node  $v$  is delinked,  $v$  itself (along with remaining children) is also delinked and forms a new tree (which may trigger further delinkage at the parents of  $v$ , etc). We introduce a new type of credit: the delinking credit. When delinking  $x$ , give a delinking credit to its parent  $v$ . If  $v$  loses a second child, thus getting delinked itself, the delinking credit pays for it. Also, we give  $x$  and  $v$  a meld credit, so the meld credit invariant holds. Thus decrementPriority is constant work plus three credits.

Let the **delinking credit invariant** be that each node that has lost a child has both a meld and cutting credit. This proves that decrementPriority is amortized  $\Theta(1)$  like we wanted.

### 3.2 FindMin

Now we have the decrementPriority runtime we want, but we have to prove that findMin is still  $\Theta(\log n)$ . To do this, we'll show that the number of nodes in a tree grows exponentially in its rank  $k$  similar to binomial heaps (wherein  $|B_k| = 2^k$ ). This forces there to be at most  $\log n$  trees, letting findMin run in  $\Theta(\log n)$ .

Let the  $k^{th}$  **Fibonacci tree**  $F_k$  equal  $B_k$  with the smallest number of nodes possible after the delinkages of decrementPriority. To create  $F_k$ , we start with  $B_k$  and remove its  $B_{k-1}$  subtree, then recursively transform the rest of its subtrees into their respective  $F_k$ . Then, a **Fibonacci heap** is a forest of Fibonacci trees.

Fibonacci Tree Number	Graphic Representation
$F_0$	
$F_1$	
$F_2$	
$F_3$	
$F_4$	

$$|F_{k+2}| = 1 + \sum_{i=0}^k |F_i|$$

$$= 1 + F_0 + F_1 + \cdots + F_k$$

$$F_{k+3} = 1 + F_0 + F_1 + \cdots + F_k + F_{k+1}$$

$$F_{k+3} - F_{k+2} = F_{k+1}$$

Thus  $F_{k+3} = F_{k+2} + F_{k+1}$ , and since  $|F_0| = |F_1| = 1$ , this is the definition of Fibonacci numbers. Because of that, we can say  $|F_k| \geq \phi^{k-2}$  where  $\phi \approx 1.618$  is the golden ratio.

This proves that Fibonacci trees grow exponentially in their rank. There are at most  $\log_{\phi} n$  different ranks, so findMin must take  $\Theta(\log n)$  time.

## 4 Matroids

A **matroid** is an abstract set-theoretical construct which generalizes the scenarios in which greedy algorithms work. A matroid is a set system  $(E, \mathcal{I})$  such that  $\mathcal{I} \subseteq 2^E$ , with the following properties:

- $\emptyset \in \mathcal{I}$
- If  $S \in \mathcal{I}, S' \in S$ , then  $S' \in \mathcal{I}$
- If  $S, S' \in \mathcal{I}, |S| < |S'|$ , then there exists an  $e \in S' \setminus S$  such that  $S \cup \{e\} \in \mathcal{I}$

The last of these is called the **exchange property**, and is the most important for the correctness of greedy algorithms.