15-750: Graduate Algorithms January 23, 2017

Lecture 3: Fibonacci Heaps

Lecturer: Gary Miller *Scribe(s): Gabriele Farina, Rui Silva*

1 Motivation

Fibonacci Heaps [Fredman and Tarjan, 1987] can be viewed as an extension of Binomial Heaps. In particular, they support the DECREASE-KEY operation in $O(1)$ amortized time, while preserving the complexity of all other operations. Table 1 provides a comparative summary of the complexity of each operation supported by Binomial and Fibonacci Heaps.

	Binomial Heaps (lazy)	Fibonacci Heaps
MAKE-HEAP	$O(1)$	$O(1)$
FIND-MIN	$O(1)$	$O(1)$
INSERT	$O(1)$	$O(1)$
DELETE-MIN	$O(\log n)$	$O(\log n)$
MELD	$O(1)$	$O(1)$
DELETE	$O(\log n)$	$O(\log n)$
DECREASE-KEY	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(1)}$

Table 1: Amortized complexity of operations in Binomial and Fibonacci Heaps.

As seen in the previous lecture, the logarithmic cost of DECREASE-KEY in Binomial Heaps is related to the fact that decreasing the key of a node may lead to a violation of the min-heap ordering (unless performed on a root), which is fixed by "bubbling up" the modified node in the heap, until the min-heap ordering is satisfied again. Since the trees have at most logarithmic height, in the worst-case (*i.e.* when it is necessary to bubble up a leaf up to the root), this operation takes $O(\log n)$ time. Fibonacci Heaps completely avoid this inconvenience, by allowing links to be cut in a controlled way.

By supporting the DECREASE-KEY operation in $O(1)$ time, Fibonacci Heaps become clearly attractive for all those applications making extensive use of this operation. In particular, historically they allowed an improvement in the runtime of the Dijkstra's single-source shortest path algorithm, bringing its time complexity down to $O(m + n \log n)$.

2 Overview

Fibonacci Heaps are an extension of Binomial Heaps, and share many common points with Binomial Heaps. To start, a Fibonacci Heap is also represented as a linked-list of heap-ordered trees (also known as a *forest*). Furthermore, like in Binomial Heaps, each node v in a Fibonacci Heap has associated a value $\textsc{rank}(v)$, representing the number of children of v; the rank of a tree T, $\textsc{rank}(T)$, is equal to the rank of the root of T, and again we only link trees with equal rank. Each node has a pointer to its parent (or NIL if the node is a root), and a list of pointers to its children. Finally, a global pointer (the MIN-PTR) at each instant points to one of trees in the forest containing the minimum element of the collection (such value must be stored at the root, since the trees are all heap-ordered).

What sets Fibonacci Heaps apart is the fact that nodes can be unlinked (or *cut*) away from their parents in amortized constant time. While the operation of cutting trees is not hard to implement *per se*, it is important to consider that it poses a threat to the efficiency of the other operations, especially DELETE-MIN. Indeed, after a number of cuts, the forest risks to be composed of scraggly trees with high rank, making the implementation of DELETE-MIN we saw in the last lecture inefficient. In order to guarantee that the trees in the forest are "bushy", the cuts need to be done in a controlled way.

3 Cuts

As aforementioned, Fibonacci Heaps introduce a new function, called CUT. When CUT is called on the non-root vertex v, the link from v to its parent gets deleted, splitting the tree. As we will show in a moment, it is possible to implement CUT in constant time. However, let's first stop to appreciate how a constant-time implementation of CUT allows us to implement $\textsc{decrease-key}(v, \delta)$ in constant time. As we recalled earlier, the problem with $\textsc{decrease-key}(v, \delta)$ is that when v is not the root of the tree, a decrement in its value might break the heap order. However, by first running $\textsc{cut}(v)$, we are guaranteed that v is the root of a tree in the heap, and changing the value of v surely does not break the heap order.

In order to avoid the potential problem of having extremely sparse trees of high rank, we limit the number of cuts among the children of any vertex to two. The rational behind this choice is that by doing this, we can guarantee that each tree of rank k has at least c^k nodes, for a suitable value of c (reasonably, $c < 2$, i.e. we get a slightly worse base for the exponent than vanilla Binomial Heaps). We defer this analysis to Section 4.2.

In order to keep track of the number of cut children of a node, we instrument every node v with a boolean flag $\textsc{marked}(v)$. When $\textsc{marked}(v)$ is FALSE, no child of v has been cut; when it is TRUE, exactly one child has been cut. In order to maintain our invariant, whenever $\textsc{marked}(v)$ is TRUE and the second child of v gets cut, v gets recursively cut as well, and its MARKED attribute reset. Figure 1 shows a sample forest with only one tree; the numbers inside the nodes represent the values stored in the heap, while gray nodes with bold text represent nodes for which MARKED is TRUE.

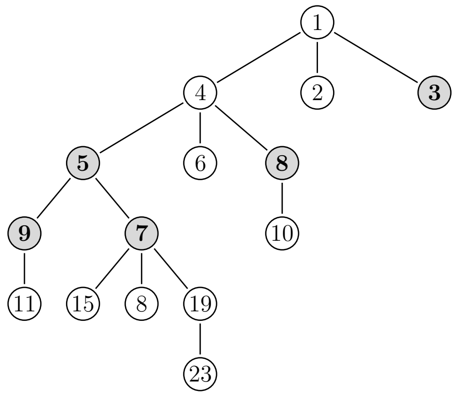


Figure 1: Sample forest with only one tree. The numbers inside the nodes represent the values stored in the heap, while gray nodes with bold text represent nodes for which MARKED is TRUE.

Algorithm 1 shows the pseudocode for the algorithm described above, while Figure 2 presents a step-by-step simulation of the effect of CUT(19) on the tree of Figure 1.

Algorithm 1 Pseudocode for the CUT operation.

```
1:  procedure CUT($v$)
2:      if PARENT($v$) $\neq$ NIL then                              ▷ $v$ is not the root of the tree
3:          $p \leftarrow$ PARENT($v$)
4:          Remove the link from $p$ to $v$; add the new tree rooted in $v$ to the list of trees
5:          MARKED($v$) $\leftarrow$ FALSE
6:          RANK($p$) $\leftarrow$ RANK($p$) $- 1$
7:          if MARKED($p$) then                                      ▷ $p$ had already lost one child
8:              CUT($p$)
9:          else
10:             MARKED($p$) $\leftarrow$ TRUE
11:         end if
12:     end if
13: end procedure
```

3.1 Interplay with the other operations

Apart from CUT, all the other operations are agnostic to the presence of marks. This implies that they can be mostly borrowed from Binomial Heaps without changes. The only exceptions are:

- INSERT(v), which also needs to initialize MARKED(v) to FALSE;
- DECREASE-KEY(v, δ), which is now re-implemented to execute CUT(v) before changing the value of the (now) root v.

4 Analysis

Most of the (amortized) analysis done for Binomial Heap transfers unchanged to Fibonacci Heaps. In particular, this is true for MAKE-HEAP, LINK, CONSOLIDATE, FIND-MIN, MELD and INSERT. On the

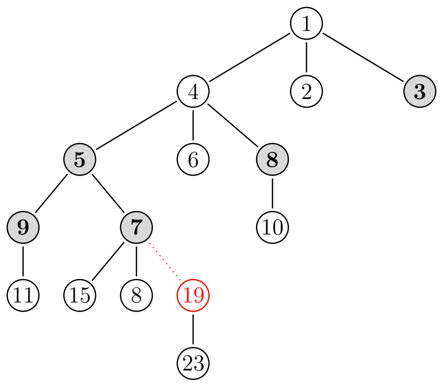
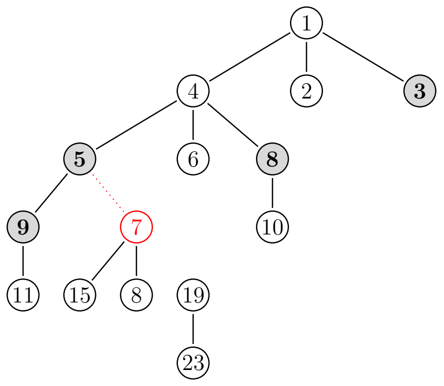
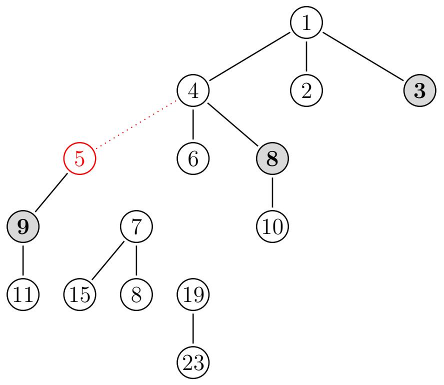
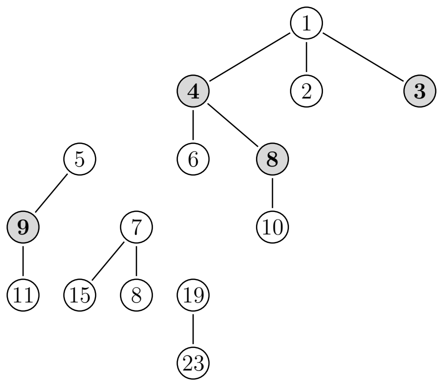
CUT(19)		The link connecting node 19 to its parent 7 is deleted. Node 7 was marked, so we recursively cut 7 as well.
CUT(7)		The link connecting node 7 to its parent 5 is deleted. Node 7 is unmarked. Node 5 was marked, so we recursively cut 5 as well.
CUT(5)		The link connecting node 5 to its parent 4 is deleted. Node 5 is unmarked. Node 4 was not marked, so we mark it and return.
End		The call to CUT(19) returns. The heap now contains 4 trees, among which 4 nodes are marked.

Figure 2: Step by step simulation of the effect of CUT(19) on the tree of Figure 1.

contrary, extra care has to be put into the analysis of DELETE-MIN (which heavily depends on having a logarithmic bound on the maximum rank of any tree), DECREASE-KEY (whose implementation differs from Binomial Heaps), and of course the new operation, CUT.

4.1 Analysis of CUT and DECREASE-KEY

As discussed previously, the ability of performing CUT operations in (amortized) constant time is what sets Fibonacci Heaps apart, allowing for an efficient implementation of the DECREASE-KEY operation. Here, we prove this amortized bound, using the potential method.

Each call to CUT performs a constant amount of work, and potentially cascades, recursively calling CUT on the parent of the node. Let c be the number of calls to CUT (including the first one) needed before stopping. For instance, in the example of Figure 2, $c = 3$ as CUT(19) results in a call to CUT(7) and CUT(5) as well.

Consider now the potential function

$$\Phi(H) = T(H) + 2\,M(H), \tag{1}$$

where $T(H)$ and $M(H)$ are respectively the number of trees and of marked nodes in the heap H. As an example, the potential function of the heap in the End stage of Figure 2 is given by $\Phi(H) = 4 + 2 \times 4 = 12$.

Each cut creates a new tree, provoking a change in potential of $\Delta_{\text{n trees}} = +c$. Furthermore, each cut, except maybe the first one, clears a marked node, provoking a successive change in potential $\Delta_{\text{cleared marks}} \leq -2(c-1)$. Finally, a node might[1] be marked when the recursion stops, provoking a change in potential $\Delta_{\text{new mark}} \leq +2$. Therefore, the overall change in potential after a call to CUT makes $c-1$ additional recursive calls is at most

$$\begin{aligned}\Delta\Phi(H) &= \Delta_{\text{n trees}} + \Delta_{\text{cleared marks}} + \Delta_{\text{new mark}}\\ &\leq c - 2(c-1) + 2\\ &= -c + 4.\end{aligned}$$

The amortized cost per CUT is then $c + 4 - c = O(1)$.

Conveniently, this also proves that DECREASE-KEY(v) runs in amortized constant time, as it just consists of a call to CUT(v) and an access to memory to decrease the value of the (now) root v.

4.2 Analysis of DELETE-MIN

The logarithmic performance of DELETE-MIN (and therefore also DELETE) in Binomial Heaps is directly related to the fact that the maximum rank of any tree in a heap of size n is at most logarithmic in n (equivalently, the number of nodes in a tree is at least exponential in the rank of that tree). Theorem 4.2, the central result for this section, proves that the same property holds for Fibonacci heaps. We begin with a lemma.

Lemma 4.1. *Let x be any vertex, and let $y_1, \ldots, y_m$ be the children of x, arranged in the order in which they were linked into x. Then* RANK$(y_i) \geq i - 2$.

Proof. This is a direct consequence of our policy of only linking trees having equal rank. When y_i was linked into x, x had already $i-1$ children. Hence, at that time, RANK$(y_i) = i - 1$. Since then, y_i might have lost one child, but not two, or otherwise y_i itself would have been cut from the tree. Hence, RANK$(y_i) \geq i - 2$. □

[1]It depends on whether the recursion stops before reaching the root or not.

Theorem 4.2. *Let T be a tree in a Fibonacci Heap. The size of T is exponential in* $\textsc{rank}(T)$.

Proof. The conclusion follows easily from Lemma 4.1. Let S_r be a lower bound on the size of any tree having rank r. Clearly, $S_0 = 1$, so that we can focus on the case $r \geq 1$. By definition of rank, the root of such a tree T has r children $y_1, \ldots, y_r$. Using Lemma 4.1 on the y_i's, we know that the subtree rooted in y_i has size at least S_{i-2} (when $i = 1$, this means that the subtree rooted in y_1 has size at least $S_0 = 1$ — i.e. node y_1 itself). Therefore, the size s of the tree T is bounded by

$$s \geq 1 + S_0 + \sum_{i=2}^{r} S_{i-2} = 2 + \sum_{i=2}^{r} S_{i-2} = 2 + \sum_{i=0}^{r-2} S_i,$$

and hence we can let

$$S_r = 2 + \sum_{i=0}^{r-2} S_i. \tag{2}$$

Equation 2 defines a recurrence relation, whose first terms are

$$\begin{aligned}
&S_0 = 1,\\
&S_1 = 2,\\
&S_2 = 2 + 1 = 3,\\
&S_3 = 2 + 1 + 2 = 5,\\
&S_4 = 2 + 1 + 2 + 3 = 8,\\
&\ldots
\end{aligned}$$

This sequence of integers is known as the "Fibonacci sequence" (hence the name of the data structure). It is immediate to show that Equation 2 can also be rewritten as

$$S_n = S_{n-1} + S_{n-2} \quad \forall n \geq 2, \qquad S_0 = 1, S_1 = 2.$$

In order to conclude the proof, we show that $S_r = \Omega(c^r)$ for some $c > 1$. For the time being, we prove that by induction, by explicitly providing, "out of the blue", the constant c; see Section 5 for a more insightful proof.

Let ϕ be the only positive root of the quadratic equation $x^2 = x + 1$; notice that such root is smaller than 2. We prove by induction that $S_r \geq \phi^r$ for all r. The base cases are readily checked, as $S_0 = 1 \geq 1$ and $S_1 = 2 \geq \phi$. Now, suppose the claim holds for $r = 1, \ldots, \bar{r} - 1$; we prove that it holds for $\bar{r}$ as well. Indeed,

$$\begin{aligned}
S_{\bar{r}} &= S_{\bar{r}-1} + S_{\bar{r}-2}\\
&\geq \phi^{\bar{r}-1} + \phi^{\bar{r}-2}\\
&= \phi^{\bar{r}-2}(\phi + 1) = \phi^{\bar{r}},
\end{aligned}$$

where we used the fact that $\phi + 1 = \phi^2$ by construction. This concludes the proof. □

Theorem 4.2 immediately implies that in a heap containing n values, the rank of any tree in the forest is $O(\log_\phi n) = O(\log n)$. Therefore, the amortized cost of DELETE-MIN (and hence, also of DELETE) is $O(\log n)$, as claimed.

References

[Fredman and Tarjan, 1987] Fredman, M. L. and Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615. 1

5 Appendix: Fibonacci numbers

We give an alternative proof of the asymptotics of Fibonacci numbers, whose definition is reported in Equation 3:

$$S_0 = 1, S_1 = 2, \qquad S_n = S_{n-1} + S_{n-2} \quad \forall\, n \geq 2. \tag{3}$$

Equation 3 can be regarded as a discrete version of a linear ordinary differential equation (ODE) of the second order. Like with higher-order linear ODEs, we can always reduce Equation 3 to first order by increasing the dimension of the space:

$$\mathbf{F}_0 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \qquad \mathbf{F}_n = \mathbf{M}\mathbf{F}_{n-1}, \text{ where } \mathbf{M} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \qquad \forall n \geq 1. \tag{4}$$

Notice that for all n, $\mathbf{F}_n = (S_{n+1}, S_n)^\top$.

We now show how to analytically compute a closed formula for $\mathbf{F}_n$. Notice that we have

$$\mathbf{F}_n = \mathbf{M}^n \mathbf{F}_0.$$

Since $\mathbf{M}$ is a symmetric matrix, the spectral theorem guarantees that it is (orthogonally) diagonalizable. In this case we have

$$\mathbf{M} = \mathbf{P}\mathbf{\Lambda}\mathbf{P}^{-1} = \mathbf{P}\mathbf{\Lambda}\mathbf{P}^\top, \text{ where } \mathbf{P} = \frac{1}{\sqrt{\phi^2+1}} \begin{pmatrix} \phi & -1 \\ 1 & \phi \end{pmatrix}, \quad \mathbf{\Lambda} = \begin{pmatrix} \phi & 0 \\ 0 & -\phi^{-1} \end{pmatrix},$$

and $\phi \approx 1.618$ is the only positive eigenvalue of $\mathbf{M}$, i.e. the only positive root of the quadratic equation $x^2 = x + 1$. This representation is particularly useful, because it allows us to write

$$\mathbf{M}^n = \underbrace{\left(\mathbf{P}\mathbf{\Lambda}\cancel{\mathbf{P}^{-1}}\right)\left(\cancel{\mathbf{P}}\mathbf{\Lambda}\cancel{\mathbf{P}^{-1}}\right)\cdots\left(\cancel{\mathbf{P}}\mathbf{\Lambda}\mathbf{P}^{-1}\right)}_{n \text{ times}} = \mathbf{P}\mathbf{\Lambda}^n\mathbf{P}^{-1} = \mathbf{P}\mathbf{\Lambda}^n\mathbf{P}^\top.$$

However, $\mathbf{\Lambda}$ is a diagonal matrix, and therefore

$$\mathbf{\Lambda}^n = \begin{pmatrix} \phi^n & 0 \\ 0 & (-\phi^{-1})^n \end{pmatrix}.$$

Wrapping up, we find that

$$\mathbf{F}_n = \mathbf{M}^n\mathbf{F}_0 = \mathbf{P} \begin{pmatrix} \phi^n & 0 \\ 0 & (-\phi^{-1})^n \end{pmatrix} \mathbf{P}^\top \mathbf{F}_0.$$

This immediately proves that the terms in $\mathbf{F}_n$ (i.e. the Fibonacci numbers) grow as $\Theta(\phi^n)$. Besides, it also gives us a closed formula for computing the n-th Fibonacci number: by expanding the matrix products, we find

$$\begin{aligned} S_n &= \frac{1+\phi}{\sqrt{5}} \cdot \phi^n - \frac{2-\phi}{\sqrt{5}} \cdot (-\phi^{-1})^n \\ &\approx 1.17 \cdot \phi^n - 0.17 \cdot (-\phi^{-1})^n. \end{aligned}$$

Notice that, as unlikely as it might look, the expression above evaluates to an integer for all $n \in \mathbb{N}$.