ICS 621: Analysis of Algorithms Fall 2019

Lecture 3 — September 20, 2019

Prof. Nodari Sitchinava *Scribe: Anton Nikolaev and Evan Hataishi*

1 Overview

In the last lecture we discussed mergeable priority queues, priority queues which support UNION operation - merging of two priority queues. We first examined the costs of operations using binary heap mergeable PQ. Then we explored a new mergeable PQ data structure, the binomial heap.

In this lecture, we are covering implementations of Lazy Binomial Heaps and Fibonacci Heaps.

2 Heaps Review

	Standard	**Lazy**	**Fibonacci**
• MAKE()	$O(1)$		
• INSERT(Q, x)	$O(1)^*$	$O(1)$	
• MINIMUM(Q)	$O(1)$		
• EXTRACT-MIN(Q)	$O(\log n)$	$O(\log n)^*$	
• DECREASE-KEY(Q, x, k)	$O(\log n)$		$O(1)^*$
• DELETE(Q, x)	$O(\log n)$	$O(\log n)^*$	
• UNION(Q_1, Q_2)	$O(\log n)$	$O(1)$	

Fibonacci heaps have amortized cost of DECREASE-KEY(Q, x, k) of $O(1)$.

Idea: if k is smaller than its parent's value splice it out and join to the root list.

3 Lazy Binomial Heaps

Let's review Lazy Binomial heaps.

Reminder: laziness of binomial heaps lies in relaxing the requirement of having at most one tree of each order in the heap. The invariant of non-decreasing values from root to leafs (the *heap order*) within the trees is still being maintained.

[*] Amortized cost

It allows us to perform UNION(Q_1, Q_2) in $O(1)$ time. If our root list is a circular doubly linked list, UNION(Q_1, Q_2) can be performed by joining the lists and updating the pointer to the minimum by setting it to the mininum of the two heaps. We also will be storing the size of the heap (as $Q.n$), and in UNION(Q_1, Q_2) we will update this size as the sum of $Q_1.n$ and $Q_2.n$.

Algorithm 1

```
function UNION($Q_1, Q_2$)
    $L_1 \leftarrow Q_1.min.left$
    $R_2 \leftarrow Q_2.min.right$
    $L_1.right \leftarrow R_2$
    $R_2.left \leftarrow L_1$
    $Q_2.min.right \leftarrow Q_1.min$
    $Q_1.min.left \leftarrow Q_2.min$
    if $Q_1.min.key < Q_2.min.key$ then
        $Q_2.min \leftarrow Q_1.min$
    $Q_2.n \leftarrow Q_1.n + Q_2.n$
    return $Q_2$
```

INSERT(Q, x) can be seen as a UNION operation on Q and a single element heap formed from x.

MINIMUM(Q) is performed in constant time because we are storing a pointer to the minimum element.

Now let's look at how EXTRACT-MIN(Q) is working, restoring our heap into a compact form, that has at most one tree of each size.

Algorithm 2

```
function EXTRACT-MIN($Q$)
    $v$ = MINIMUM($Q$)
    $c = v.child$
    $L_v = v.left$
    $R_v = v.right$
    if $c \neq$ NIL then
        $L_c = c.left$
        $L_v.right = c$
        $c.left = L_v$
        $R_v.left = L_c$
        $L_c.right = R_v$
    else
        $L_v.right = R_v$
        $R_v.left = L_v$
    $Q.min = R_v$                                   ▷ arbitrary head
    CONSOLIDATE($Q$)
    return $v$
```

Let's figure out how CONSOLIDATE(Q) works:

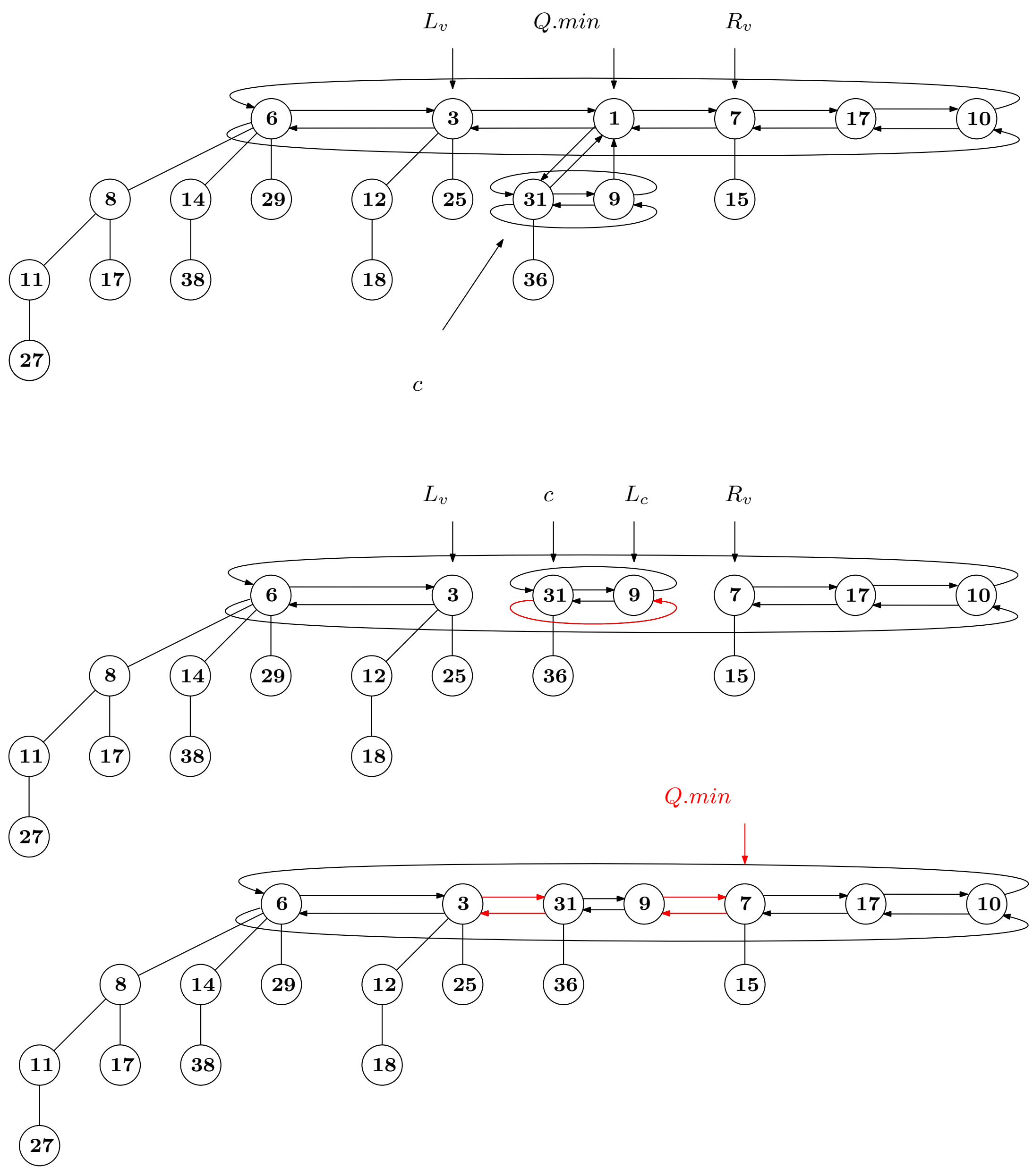

Figure 1: Example of EXTRACT-MIN(Q)

Algorithm 3

```
function CONSOLIDATE(Q)
    Initialize $\log n$-sized array $A$ to NIL
    for each $v$ in root list do
        $d = v.degree$
        while $A[d] \neq$ NIL do
            $v = $ LINK$(v, A[d])$
            $A[d] =$ NIL
            $d = d + 1$
        $A[d] = v$; $v.parent =$ NIL
```

As we can see we end up with the heap of optimal size.

Correctness

We can prove correctness by showing that the invariant is preserved. It comes from proper implementation of LINK$(v, A[d])$. If we pick element that is smaller to be the parent of the two, the invariant will be preserved.

Analysis

We will use the Potential Method with $\Phi_i = t_i =$ number of trees in the root list after the i-th operation.

Let d be the number of children of the $Q.min$. Observe that the number of children in a binomial tree is at most $\log n$ (and we never add children to any of the original binomial trees, only potentially remove them). Therefore, $d \leq \log n$.

- Actual cost $c_i \leq O(1) + (t_{i-1} + d) + \log n \leq O(1) + t_{i-1} + 2\log n$
- Change in potential is $\Delta\Phi_i = t_i - t_{i-1}$

Then the amortized cost is defined by

$$\hat{c_i} = c_i + \Delta\Phi_i \leq O(1) + 2\log n + t_i \leq O(1) + 3\log n = O(\log n),$$

since the number of trees in the root list after consolidation is at most $\log n$, i.e., $t_i \leq \log n$.

DELETE(Q, x) and DECREASE-KEY(Q, x, k) stay the same as in standard Binomial heap.

4 Fibonacci Heaps

Fibonacci heaps are an extension of lazy binomial heaps. They follow the same "lazy" approach and enforce the same runtime complexity for each of the lazy binomial heap's operations. However, Fibonacci heaps provide an optimization to support a more efficient DECREASE-KEY operation.

4.1 Motivation

A heap with a more efficient Decrease-Key operation has applications to classical graph problems. In the case of Dijkstra's single-source shortest paths algorithm, Fibonacci heaps are used to obtain a reduced runtime complexity.

Let m be the number of edges and n be the number of vertices in a graph. Then Dijksta's algorithm runs in $O(m \cdot T_{\text{Decrease-Key}} + n \cdot T_{\text{Extract-Min}})$, where $T_{\text{Decrease-Key}}$ is the time complexity of each Decrease-Key operation, which is called for each edge, and $T_{\text{Extract-Min}}$ is the time complexity of each Extract-Min operation, which is called for each node. If the heap is implemented using a Binary Heap or as a Binomial Heap, the runtime of Dijkstra's algorithm becomes $O(m \log n + n \log n)$. By implementing the heap as a Fibonacci Heap with $O(1)$ amortized Decrease-Key operation, the runtime of Dijkstra's algorithm becomes $O(m + n \log n)$ which is strictly better for dense graphs (with $m = \omega(n)$ edges).

4.2 Properties

In general, Fibonacci heaps can be implemented with the same data structure as lazy binomial heaps with the addition of a **marked** property for nodes:

1. Each tree is heap-ordered
2. The root list may contain an arbitrary number of trees
3. Sibling lists are doubly-linked circular lists
4. Each node v has properties:
   - $v.parent$: pointer to its parent
   - $v.child$: pointer to one (arbitrary) child
   - $v.degree$: number of children
   - $v.marked$: a boolean representing whether or not v has lost a child since becoming a child of another node

4.3 Functions

Fibonacci heaps are able to achieve an $O(1)$ amortized cost for Decrease-Key (Algorithm 4) by introducing a new function Cut (Algorithm 5), as well as making a minor modification during consolidation. Lazy binomial heaps implement Decrease-Key by modifying a given node's key value then restoring heap order up the tree, which incurs a cost of $O(\log n)$ in the worst case. Fibonacci heaps avoid this cost by modifying the node's key and cutting out the selected node, along with its entire subtree, and adding it to the root list. Since the node becomes a new root, heap order is guaranteed to be maintained.

Algorithm 4

```
1: function Fib-Decrease-Key($Q, x, k$)          ▷ Assert $k < v.key$
2:     $v.key \leftarrow k$
3:     if $v.parent \neq NIL$ and $v.key < v.parent.key$ then
4:         Cut($Q, v, v.parent$)
```

If the cut out node's parent was already marked before this operation, i.e. it previously had a child

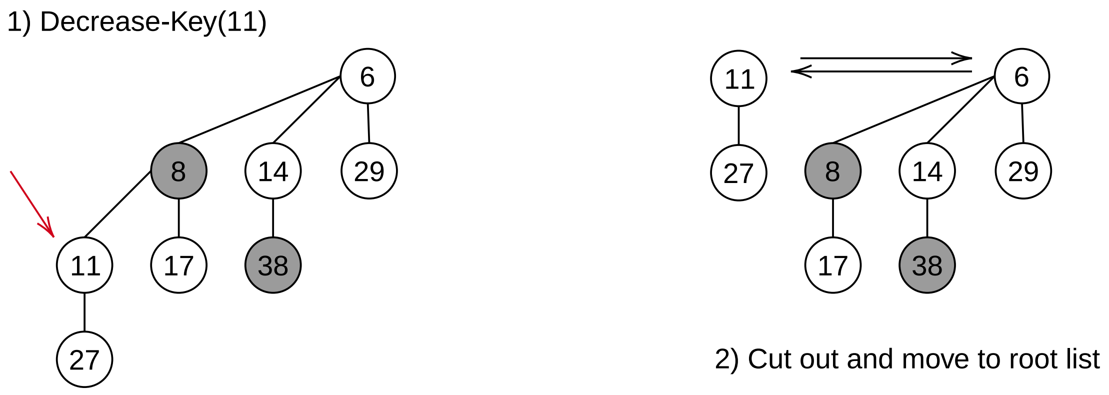

Figure 2: Calling Decrease-Key on Node 11

cut away, we call Cut on the parent as well. This is done recursively until we find a parent that has not yet been marked or reach the root of the tree.

Algorithm 5

```
1: function Cut($Q, v, p$)                                  ▷ $p$ is the parent of $v$
2:     Remove $v$ from child list of $p$
3:     Add $v$ to the root list of l$Q$
4:     $v.marked$ = FALSE; $v.parent = NIL$                 ▷ Un-mark $v$
5:     if $p.parent \neq NIL$ then
6:         if $p.marked$ = FALSE then
7:             $p.marked$ = TRUE                            ▷ $p$ just lost a child, so mark it
8:         else                                             ▷ $p$ just lost the second child
9:             Cut($Q, p, p.parent$)                        ▷ so add it to the root list too
```

However, after performing many cuts, the heap may become filled with many small and thin trees. Therefore, the number of children cut from any node is limited to two before it is cut itself and added to the root list. This is done by ”marking” nodes that have children cut away. Also, it is important to note that the binomial tree structure may no longer be preserved in the process of cutting away arbitrary children. Therefore, it is no longer guaranteed that Fibonacci heaps are made up of binomial trees.

Figures 2 and 3 show an example of a Fibonacci heap that calls Decrease-Key on the node 11. In Figure 2, node 11 and its subtree are cut out and added to the root list. Continuing in 3, 11’s parent (Node 8) was already marked so Cut is called again. When Node 8 is added to the root list, it becomes unmarked. Node 8’s parent (Node 6) just lost a child, so it becomes marked.

All other lazy binomial heap operations are preserved except for linking (Algorithm 6) within consolidation. The only difference is that when a root becomes the child of another root, the root that becomes a child becomes unmarked.

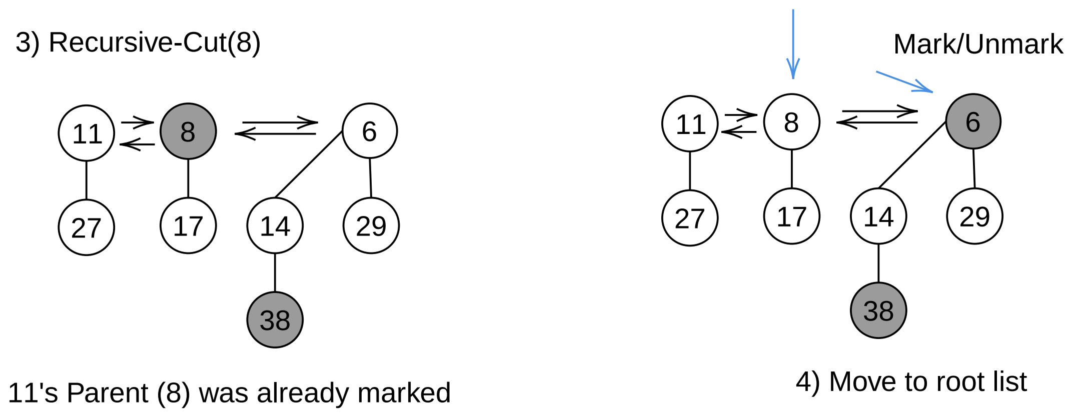

Figure 3: CUT on a parent node that was previously marked

Algorithm 6

```
1: function LINK($v, w$)
2:     if $w.key < v.key$ then
3:         SWAP($v, w$)                         ▷ make sure $v$ is smaller
4:     Add $w$ to the child list of $v$
5:     $v.degree = v.degree + 1$
6:     $w.marked =$ FALSE        ▷ Previous root gets un-marked after becoming a new child
```

4.4 Runtime Analysis of DECREASE-KEY

Let t_i be the number of root list trees and m_i be the number of marked nodes after the i-th operation on the Fibonacci heap. Let k be a constant (we will determine it later). We can define the potential function for Fibonacci heap to be

$$\Phi_i = k(t_i + 2m_i).$$

Let's analyze the DECREASE-KEY operation. If t' trees were added to the root list during all the cuts of a DECREASE-KEY operation, then there must have been at least $t' - 1$ marked nodes that became unmarked after being added to the root list (the original node on which CUT was called might not have been marked to begin with). Also, if all nodes are cut up to the root, the root stays as a root, but will still get marked (hence a "+1" in the equation below). We can therefore bound the new number of marked nodes as:

$$m_i \le m_{i-1} - (t' - 1) + 1.$$

The change in potential after the DECREASE-KEY operation is:

$$\begin{aligned}\Delta\Phi_i &= k(t_i + 2m_i) - k(t_{i-1} + 2m_{i-1}),\\ &= k(t_i - t_{i-1}) + 2k(m_i - m_{i-1}).\end{aligned}$$

An upper bound for $m_i - m_{i-1}$ is given by the above equation for m_i:

$$\begin{aligned}m_i &\le m_{i-1} - (t' - 1) + 1,\\ m_i - m_{i-1} &\le 2 - t'.\end{aligned}$$

By also substituting t' for $t_i - t_{i-1}$, the change in potential can be simplified:

$$\begin{aligned}\Delta\Phi_i &= k(t_i - t_{i-1}) + 2k(m_i - m_{i-1}),\\ &\le k \cdot t' + 2k(2 - t')\\ &\le k(t' + 2(2 - t')),\\ &\le k(4 - t').\end{aligned}$$

The number of cut out nodes is the bound for the actual cost c_i of the Decrease-Key operation. Therefore, the actual cost c_i is $O(t')$. More precisely, $c_i \le \bar{k} \cdot t'$ for some constant $\bar{k}$. By setting $k = \bar{k}$, we get

$$\begin{aligned}\hat{c}_i &= c_i + \Delta\Phi,\\ &\le \bar{k} \cdot t' + k(4 - t'),\\ &\le 4\bar{k},\\ &= O(1).\end{aligned}$$

I.e., the amortized cost of Decrease-Key operation is $O(1)$.

4.5 Runtime Analysis of Extract-Min

Fibonacci heaps implement Extract-Min the exact same was as lazy binomial heaps do. From previous analysis, we have shown that Extract-Min has an $O(\log n)$ amortized time complexity due to the fact that a binomial heap with n nodes will have at most $\log n$ binomial trees after consolidation. Therefore, we must also make sure that there exists a similarly reasonable bound within Fibonacci heaps.

Let F_k represent the k-th Fibonacci number. The following lemma, which is proven in the CLRS textbook, bounds the size of the subtree rooted at every node v in the Fibonacci heap:

Lemma 1. *Let $k = v.degree$. Then for every node v in the Fibonacci heap, SIZE(v) $\ge F_{k+2} \ge \phi^k$, where $\phi = \frac{1+\sqrt{5}}{2}$ and SIZE(v) represents the number of nodes in the subtree rooted at v.*

Then it follows that the degree of each node is bounded by:

Corollary 2. $k \le \log_\phi n$

Observe that $\log_\phi n \approx 1.44042 \cdot \log_2 n \le 2 \cdot \log_2 n$ for any $n \ge 1$.

Let t_i be the number of root list trees and m_i be the number of marked nodes after the i-th operation, and let d_{i-1} be the number of children of $Q.min$ before the i-th operation. Let $\bar{k}$ be the constant hidden in the big-O notation, as defined earlier. Any node has at most $\log_\phi n \le 2\log n$ children, so $d_{i-1} \le 2\log n$ as well. Recall that the potential function of a Fibonacci heap is defined as

$$\Phi_i = \bar{k}(t_i - 2m).$$

Then the change in potential after the Extract-Min operation is:

$$\begin{aligned}\Delta\Phi_i &= \bar{k}(t_i + 2m_i) - \bar{k}(t_{i-1} + 2m_{i-1}),\\ &= \bar{k}(t_i - t_{i-1}) + 2\bar{k}(m_i - m_{i-1}).\end{aligned}$$

During the consolidation step, nodes that become children of other nodes are unmarked so $m_i < m_{i-1}$, i.e. $(m_i - m_{i-1}) \leq 0$, and the change in potential becomes

$$\begin{aligned} \Delta\Phi_i &= \bar{k}(t_i - t_{i-1}) + 2\bar{k}(m_i - m_{i-1}), \\ &\leq \bar{k}(t_i - t_{i-1}). \end{aligned}$$

The actual cost of Extract-Min is the sum of attaching $Q.min$'s children list to the root list ($O(1)$), consolidation of the now $t_{i-1} - 1 + d_{i-1} \leq t_{i-1} + 2\log n$ root nodes, and finding the minimum amongst the new (at most) $\log n$ root nodes:

$$\begin{aligned} c_i &\leq O(1) + t_{i-1} + d_{i-1} + \log n, \\ &\leq O(1) + t_{i-1} + 3\log n. \end{aligned}$$

Plugging in for the amortized runtime:

$$\begin{aligned} \hat{c}_i &= c_i + \Delta\Phi_i, \\ &\leq O(1) + t_{i-1} + 3\log n + \bar{k}(t_i - t_{i-1}), \\ &\leq O(1) + 3\log n + \bar{k} \cdot t_i - t_{i-1}(\bar{k} - 1). \end{aligned}$$

Since $\bar{k} \geq 1$, $t_{i-1}(\bar{k} - 1) \leq 0$. Also, $t_i \leq \log n$ after consolidation, since there are at most $\log n$ root nodes. The above equation can be further reduced:

$$\begin{aligned} \hat{c}_i &\leq O(1) + 2\log n + \bar{k} \cdot t_i - t_{i-1}(\bar{k} - 1), \\ &\leq O(1) + 2\log n + \bar{k} \cdot t_i, \\ &\leq O(1) + 2\log n + \bar{k}\log n, \\ &= O(\log n). \end{aligned}$$

I.e. the amortized cost of Extract-Min in the Fibonacci Heap is $O(\log n)$.