ICS 621: Analysis of Algorithms	Fall 2021

Lecture 3 — September 17, 2021

Prof. Nodari Sitchinava *Scribe: In Woo Park, Darlene Agbayani, Michael Rogers*

1 Overview

Previously, we covered mergeable priority queues, starting with a brief overview of the priority queue abstract data type (ADT), and reviews of both Binary and Binomial heaps. Next, we analyzed each corresponding ADT function used in Binomial heap operations. We then introduced Lazy Binomial heaps, which improve upon the traditional Binomial heaps by delaying the work normally carried out during the Union(Q) operation, thus reducing the Union(Q_1, Q_2) operation to a constant factor. Finally, for homework, we designed and analyzed an efficient algorithm to perform Extract-Min(Q).

In this lecture, we will begin by reviewing the Lazy Binomial Heap operations, including an in-depth analysis of Extract-Min(Q), which was introduced in the homework. Understanding the algorithms behind Lazy Binomial heaps are a great segue into today's primary topic, Fibonacci heaps. We will see how Fibonacci heaps can be used to reduce runtimes even further, and also discuss a well-known application of this data structure, Dijkstra's Algorithm.

2 Mergeable Heaps Continued

	Binomial	**Lazy Binomial**	**Fibonacci**
Make()	$O(1)$	$O(1)$	$O(1)$
Insert(Q, x)	$O(1)^*$	$O(1)$	$O(1)$
Minimum(Q)	$O(1)$	$O(1)$	$O(1)$
Extract-Min(Q)	$O(\log n)$	$O(\log n)^*$	$O(\log n)^*$
Decrease-Key(Q, x, k)	$O(\log n)$	$O(\log n)$	$O(1)^*$
Delete(Q, x)	$O(\log n)$	$O(\log n)^*$	$O(\log n)$
Union(Q_1, Q_2)	$O(\log n)$	$O(1)$	$O(1)$

Table 1: Running times for operations of the three implementations of the priority queue. The running times marked with * are amortized.

The key difference between the Lazy Binomial Heap and Fibonacci Heap implementations is the amortized cost of Decrease-Key(Q, x, k) becomes constant time, while still maintaining the advantages of Lazy Binomial over standard Binomial heaps, i.e., the exchange from worst-case $O(\log n)$ to amortized cost $O(\log n)$ for calls made to Extract-Min(Q) and Delete(Q, x), with the faster, worst-case $O(1)$ cost of calls to Union(Q_1, Q_2).

3 Lazy Binomial Heaps

Let's review Lazy Binomial heaps.

Reminder: The 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 leaves (the *heap order*) within the trees is still being maintained.

3.1 Lazy Union(Q_1, Q_2)

Relaxing the invariant that there be at most one binomial tree of each order 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 minimum 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 $Q_2.n$. Moreover, since the sibling lists are *circular*, we no longer need to keep a pointer to the head node. Instead, we can pick any arbitrary node in the root list as a starting point, and since we already maintain a pointer to the minimum element, it is easy enough to just start with that node.

Algorithm 1

UNION(Q_1, Q_2)
1 $L_1 = Q_1.min.left$
2 $R_2 = Q_2.min.right$
3 $L_1.right = R_2$
4 $Q_2.min.right = Q_1.min$
5 $Q_1.min.left = Q_2.min$
6 **if** $Q_1.min.key < Q_2.min.key$
7 $\quad$ $Q_2.min = Q_1.min$
8 $\quad$ **return** Q_2

Observe that this implementation of Union takes only $O(1)$ time in the worst case. However, it no longer preserves the invariant that the list is ordered by the sizes of trees. Moreover, it is also no longer true that the binomial heap will have at most one binomial tree of each size. In fact, the root lists can contain arbitrarily many trees of the same size. We will restore these invariants during the EXTRACT-MIN operation.

3.2 Insert(Q, x)

On the bright side, we can also implement INSERT(Q, x) in $O(1)$ worst-case time:

Algorithm 2

INSERT(Q, x)
1 $Q' =$ **new** BINOMIALHEAP(x)
2 **return** UNION(Q, Q')

3.3 Extract-Min(Q)

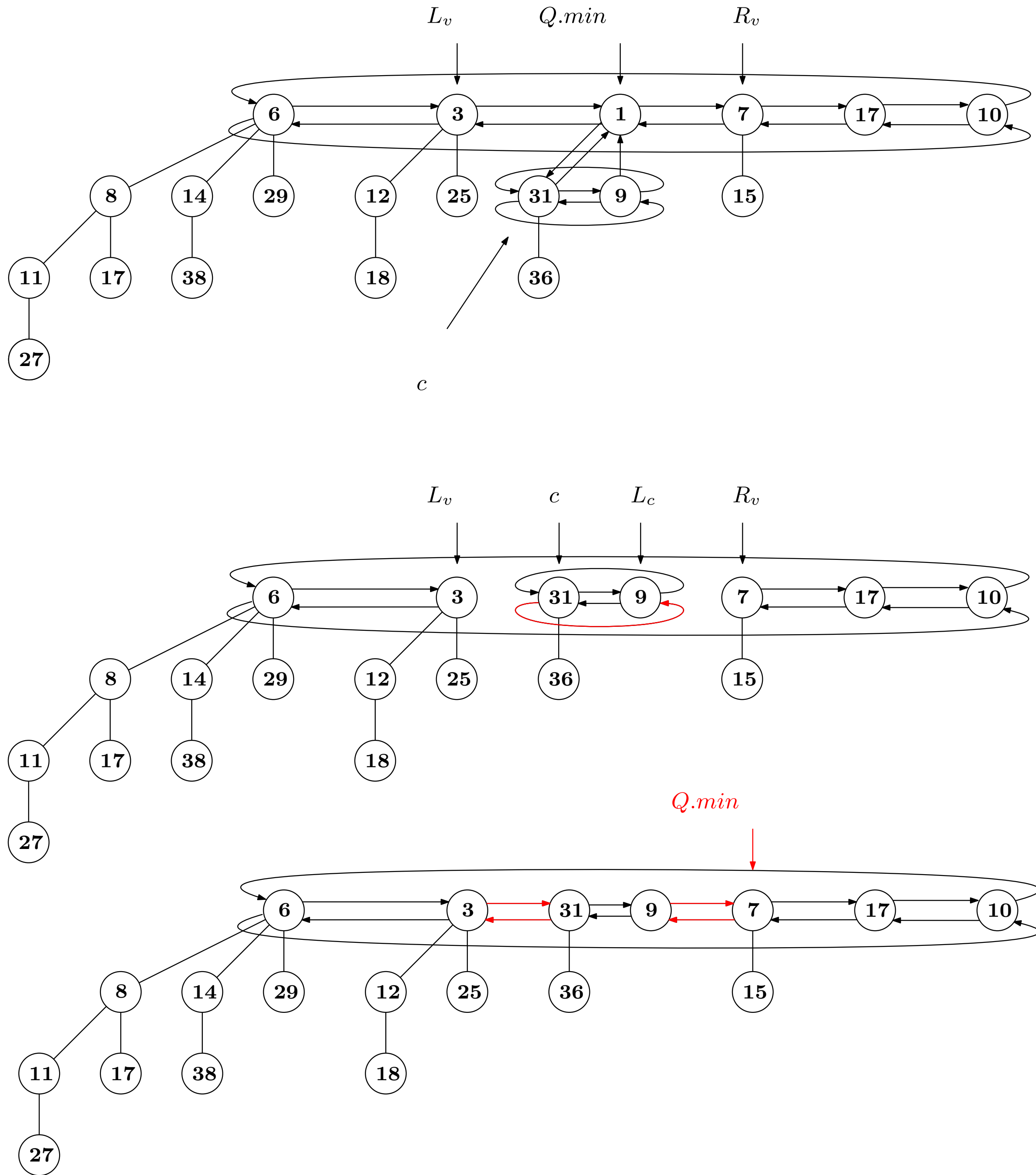


Figure 1: An example of EXTRACT-MIN(Q)

Extract-Min(Q), as shown in Figure 1, is one of the more interesting operations, and is where most of the important work happens in the Lazy Binomial Heap. We begin by finding the node v that stores the minimum key, removing the tree rooted at v from the root list and splicing the children of v into the root list. Since the list of children is stored as a circular list, we can perform the splicing in constant time.

Following the splicing, we once again have a collection of binomial trees in arbitrary order with no guarantees of how many of each order. At this point we perform CONSOLIDATE(Q) operation which consolidates all the trees of the root list, so there is at most one of each order and which finds the new minimum node among the remaining nodes of the root list.

Algorithm 3

```
EXTRACT-MIN(Q)
 1  v = MINIMUM(Q)
 2  c = v.child
 3  L_v = v.left
 4  R_v = v.right
 5  if c ≠ NIL
 6       L_c = c.left
 7       L_v.right = c
 8       c.left = L_v
 9       R_v.left = L_c
10       L_c.right = R_v
11  else
12       L_v.right = R_v
13       R_v.left = L_v
14  Q.min = R_v                                   ▷ arbitrary head
15  CONSOLIDATE(Q)
16  return v
```

Figure 1, specifically the top most heap, shows all the pointers maintained by EXTRACT-MIN(Q). The second heap shows how **lines 1-14** of the EXTRACT-MIN(Q) pseudocode removes the minimum node 1 by removing 1's pointers to its children and vice versa. To maintain the lazy binomial heap, we simply move 1's children up to the root list and move their pointers to respect that this data structure is a doubly linked circular list.

MINIMUM(Q) simply calls return $Q.min$ where $Q.min$ contains a pointer to the current index. This function takes $O(1)$ time.

Algorithm 4

```
MINIMUM(Q)
1  return Q.min
```

Now that we have successfully removed the minimum node from the lazy binomial heap, we need to keep track of a new minimum node so that the MINIMUM(Q) function has runtime of $O(1)$. Therefore, after we remove the minimum node, we set R_v as our initial minimum pointer, and at each iteration of the **for** loop in CONSOLIDATE(Q) (lines 10-14), we compare $A[i].key$ with our minimum pointer, and update the minimum pointer if $A[i].key$ is ever less than $Q.min$.

EXTRACT-MIN(Q) calls the function CONSOLIDATE(Q) in order to cleanup the binomial heap and maintain the property that there are at most one tree of each size. We process each root v such that the degree of v is used as the index of the array A, and A stores a pointer to that node if it is

Algorithm 5

```
CONSOLIDATE(Q)
1   Initialize log n-sized array A to NIL
2   for each v in root list
3       d = v.degree
4       while A[d] ≠ NIL
5           v = LINK(v, A[d])
6           A[d] = NIL
7           d = d + 1
8       A[d] = v; v.parent = NIL
9   min = +∞
10  for i = 0 to log n − 1 do
11      if A[i] ≠ NIL then
12          Add A[i] to the root list
13          if A[i].key < min then
14              Q.min ← A[i]; min = A[i].key
```

empty. If A is not empty, we need to link the two subtrees together to maintain the binomial heap property by using the function LINK$(v, A[d])$.

Algorithm 6

```
LINK(v, w)
1   if w.key < v.key
2       SWAP(v, w)                          ▷ make sure v is smaller
3   Add w to the child list of v
4   v.degree = v.degree + 1
```

LINK$(v, A[d])$ removes v from the root list of A and makes it the child of $A[v.degree]$ where $A[v.degree]$ has a pointer to a root node of the same degree. The function ends with A having a pointer to the new subtree with increased degree and subsequently continuing the LINK$(v, A[d])$ function if A already has a pointer to a older subtree of the same degree.

EXTRACT-MIN(Q) ends with returning v the minimum node, but in addition, we cleaned up the lazy binomial heap by combining root nodes of the same degree until we are left with a binomial heap with at most one tree of each size.

Proving Algorithm Correctness

The invariant for the **for** loop on lines 2-8 of CONSOLIDATE(Q) is that during each iteration of the loop, A points to a subset of trees that are unique in size. Before the first iteration of the loop (Initialization), $A[d]$ is empty so the invariant is satisfied. To show that the Maintenance condition is satisfied, CONSOLIDATE(Q) calls the **while** loop which needs its own invariant and we'll address it in the next paragraph. During termination we are left with the useful property that after the last iteration, all trees in the subset are unique in size.

⟨ Correct invariant is presented on page 517 of CLRS ⟩ The invariant we can place for each iteration

of the **while** loop on lines 4-7 is that, if $A[d+1]$ is non-empty, $A[d]$ must continue linking with v. During initialization, $A[d]$ already has a pointer to a subtree with the same degree, therefore we compare $v.key$ and $A[d].key$ values to determine which gets merged as a child and which stays in the root node. Once chosen, the subtree with the higher key value is linked as the leftmost child of the root node. The root node is then incremented in degree. During maintenance, our previous $A[d]$, which had successfully linked during initialization, may encounter that $A[d+1]$ also already has a pointer stored to a subtree, therefore we continue the linking process until $A[d+n]$ root nodes are checked and linked. During termination, $A[d+1]$ is finally empty, therefore, no more links are required, the new subtree is incremented in degree, and stored at $A[d+1]$ and the loop is terminated. We are left with the useful property that the subtree we created is unique in size.

Finally, we iterate over all the keys stored at the roots and update the $Q.min$ to point to the smallest.

3.4 Amortized Cost

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 is $c_i \leq O(1) + (t_{i-1} + d) + \log n \leq O(1) + t_{i-1} + 2\log n$, where $O(1)$ sets the pointer, $(t_{i-1} + d)$ represents the runtime of consolidation, and $\log n$ is the initialization of the array and updates the pointer. The bound on d is $\log n - 1$ because every tree is valid, therefore the number of children is at most $\log n$.
- The change in potential is $\Delta\Phi_i = t_i - t_{i-1}$ Where t_i is the number of trees after the ith operation, and t_{i-1} is the number of trees before the ith operation.

Then the amortized cost is defined by

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

because 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.

Student Question

The for loop iterates for each v in the root list. I thought that would be k. Where does that disappear? I thought k would be the number of roots before we consolidate?

Notice that we define it not as k but rather t_{i-1} where t_{i-1} is the number of trees in the root list before consolidation. Think of it as k_i, before k was the order of a binomial tree, where the number of elements at the root list was k. k doesn't disappear, but rather becomes t_{i-1} based on how we defined it in our analysis.

Student Question
How come the root list is the only one considered? What about the while loop?

Among all nodes, the for and $while$ loop will process twice. Some nodes will be processed twice such as the minimum node 3 in Figure 1. Although our minimum node is processed the most, there aren't many nodes that will be processed that many times. The number of times for and $while$ iterates will be 2 - (number of nodes). Once processed and subsequently removed, it is never visited again. Among all nodes, it will be processed 2 times. Across k items in the root list, the total number of processed nodes will be $2k$ or t_{i-1}.

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 7) by introducing a new function CUT (Algorithm 8), 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 7

```
FIB-DECREASE-KEY(Q, x, k)                          ▷ Assert k < v.key
1  v.key = k
2  if v.parent ≠ NIL and v.key < v.parent.key
3      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 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 8

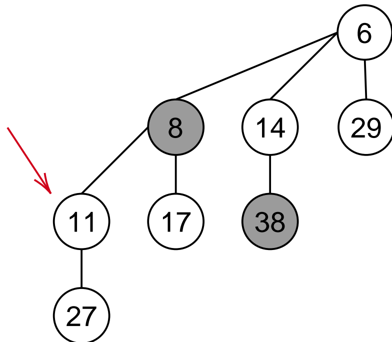
```
CUT(Q, v, p)                                       ▷ p is the parent of v
1  Remove v from child list of p
2  Add v to the root list of Q
3  v.marked = FALSE; v.parent = NIL                ▷ Unmark v
4  if p.parent ≠ NIL
5      if p.marked = FALSE
6          p.marked = TRUE                         ▷ p just lost a child, so mark it
7      else                                        ▷ p just lost the second child
8          CUT(Q, p, p.parent)                     ▷ so add it to the root list too
```

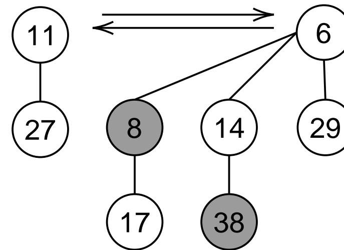
However, after performing many cuts, the heap may become filled with many small and thin trees. Therefore, the number of children cut from any node is limited to two before it is cut itself and added to the root list. This is done by "marking" nodes that have children cut away. Also, it is important 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 Figure 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 9) within consolidation. The only difference is that when a root becomes the child of another root, the root that becomes a child becomes unmarked.

1) Fib-Decrease-Key(11)

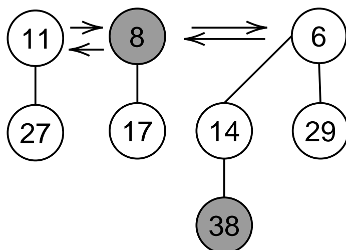




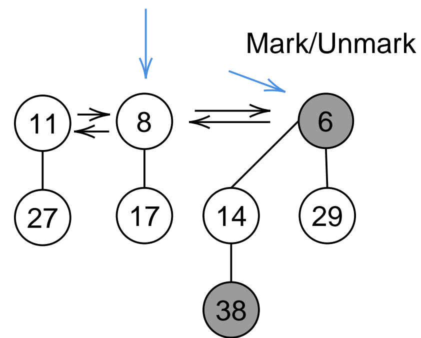
2) Cut out and move to root list

Figure 2: Calling Fib-Decrease-Key on Node 11

3) Recursive-Cut(8)



11's Parent (8) was already marked



4) Move to root list

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

Algorithm 9

Link(v, w)

1 **if** $w.key < v.key$

2 $\quad$ swap(v, w) $\qquad$ ▷ make sure v is smaller

3 Add w to the child list of v

4 $v.degree = v.degree + 1$

5 $w.marked = \text{False}$ $\qquad$ ▷ Previous root gets unmarked 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:

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

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:

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

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

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

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, $c_i \leq \bar{k} \cdot t'$ for some constant $\bar{k}$. By setting $k = \bar{k}$, we get

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

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 way 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 $d = v.degree$. Then for every node v in the Fibonacci heap, SIZE(v) $\geq F_{k+2} \geq \phi^d$, 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. $d \leq \log_\phi n$

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

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 $\log_\phi n \leq 2 \log n$ children, so $d_{i-1} \leq 2 \log n$ 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 $O(\log n)$.

5 Application: Single Source Shortest Paths

You may recall the Single Source Shortest Paths problem from undergraduate algorithms: Given a graph with n vertices and m edges, each edge has a weight, and given a start vertex s, we want to find the shortest paths from s to every other vertex. The shortest path is defined as the sum of weights of the edges needed to get there.

Dijkstra's Algorithm is often taught in undergraduate algorithms which works for non-negative weights, and typically uses a priority queue with n INSERT calls, and a call to EXTRACT-MIN on each node (total n times), and at most m calls to DECREASE-KEY, where m is the number of edges and DECREASE-KEY is called for every edge.

We are taught it runs in $O(m + n \log n)$ time, but without Fibonacci heaps it is not the case. Analyzing and comparing the runtimes of operations across several mergeable heaps (see Table 2), you will notice Binary heaps and Binomial heaps both result in $O(m \log n)$ time, whereas the total runtime for the Fibonacci heap is $O(m + n \log n)$ because even when the number of calls to DECREASE-KEY is large, the *amortized* cost is still constant. This means that if m is larger than n by more than a constant factor (dense graphs), the Fibonacci heaps perform asymptotically better than the other two.

Operation	Running times			Count
	Binary Heap	Binomial Heap	Fibonacci Heap	
INSERT	$O(1)$	$O(1)^*$	$O(1)$	n
EXTRACT-MIN	$O(\log n)$	$O(\log n)$	$O(\log n)^*$	n
DECREASE-KEY	$O(\log n)$	$O(\log n)$	$O(1)^*$	m
Total	$O(n + n \log n + m \log n)$ $= O(m \log n)$	$O(n + n \log n + m \log n)$ $= O(m \log n)$	$O(n + m + n \log n)$ $= O(m + n \log n)$	

Table 2: Running times for Dijkstra's Algorithm operations of the three implementations of the priority queue. The running times marked with * are amortized.

Student Question
Are the total costs amortized or worst-case costs? They appear to be amortized because they are determined from amortized costs.

This was the question I wanted to ask of you following the lecture, is it actually Amortized Cost, or Worst-case Cost?

Recall, the definition of amortized cost is if I take n operations, my total time will be n times the total cost of each operation in the worst case. No matter how many operations I do, say n, my total runtime of n operations will be n times the amortized cost of each operation, in the worst case. Therefore, if we are doing n and m calls, the total cost will be n times the amortized cost of each operation plus m times the amortized cost of each of those operations, overall, in the worst-case, as long as n and m are large enough. This is **not** to say that this is the worst-case runtime of an individual operation, some will run shorter and others longer, but the total overall is the worst-case.