19 Fibonacci Heaps

19.1 Structure of Fibonacci heaps

There is no exercise in this section.

19.2 Mergeable-heap operations

19.2-1

> Show the Fibonacci heap that results from calling $\text{FIB-HEAP-EXTRACT-MIN}$ on the Fibonacci heap shown in Figure 19.4(m).

(Omit!)

19.3 Decreasing a key and deleting a node

19.3-1

> Suppose that a root x in a Fibonacci heap is marked. Explain how x came to be a marked root. Argue that it doesn't matter to the analysis that x is marked, even though it is not a root that was first linked to another node and then lost one child.

x came to be a marked root because at some point it had been a marked child of $H.min$ which had been removed in $\text{FIB-HEAP-EXTRACT-MIN}$ operation. See figure 19.4 for an example, where the node with key 18 became a marked root. It doesn't add the potential for having to do any more actual work for it to be marked. This is because the only time that markedness is checked is in line 3 of cascading cut. This however is only ever run on nodes whose parent is non NIL. Since every root has NIL as it parent, line 3 of cascading cut will never be run on this marked root. It will still cause the potential function to be larger than needed, but that extra computation that was paid in to get the potential function higher will never be used up later.

19.3-2

> Justify the $O(1)$ amortized time of $\text{FIB-HEAP-DECREASE-KEY}$ as an average cost per operation by using aggregate analysis.

Recall that the actual cost of $\text{FIB-HEAP-DECREASE-KEY}$ is $O(c)$, where c is the number of calls made to CASCADING-CUT. If c_i is the number of calls made on the ith key decrease, then the total time of n calls to $\text{FIB-HEAPDECREASE-KEY}$ is $\sum_{i=1}^n O(c_i)$.

Next observe that every call to CASCADING-CUT moves a node to the root, and every call to a root node takes $O(1)$. Since no roots ever become children during the course of these calls, we must have that $\sum_{i=1}^n c_i = O(n)$. Therefore the aggregate cost is $O(n)$, so the average, or amortized, cost is $O(1)$.

19.4 Bounding the maximum degree

19.4-1

> Professor Pinocchio claims that the height of an n-node Fibonacci heap is $O(\lg n)$. Show that the professor is mistaken by exhibiting, for any positive integer n, a sequence of Fibonacci-heap operations that creates a Fibonacci heap consisting of just one tree that is a linear chain of n nodes.

- **Initialize:** insert 3 numbers then extract-min.

- **Iteration:** insert 3 numbers, in which at least two numbers are less than the root of chain, then extract-min. The smallest newly inserted number will be extracted and the remaining two numbers will form a heap whose degree of root is 1, and since the root of the heap is less than the old chain, the chain will be merged into the newly created heap. Finally we should delete the node which contains the largest number of the 3 inserted numbers.

19.4-2

> Suppose we generalize the cascading-cut rule to cut a node x from its parent as soon as it loses its kth child, for some integer constant k. (The rule in Section 19.3 uses $k = 2$.) For what values of k is $D(n) = O(\lg n)$?

Following the proof of lemma 19.1, if x is any node if a Fibonacci heap, $x.\text{degree} = m$, and x has children $y_1, y_2, \dots, y_m$, then $y_1.\text{degree} \ge 0$ and $y_i.\text{degree} \ge i - k$. Thus, if s_m denotes the fewest nodes possible in a node of degree m, then we have $s_0 = 1, s_1 = 2, \dots, s_{k-1} = k$ and in general, $s_m = k + \sum_{i=0}^{m-k} s_i$. Thus, the difference between s_m and s_{m-1} is s_{m-k}.

Let $\{f_m\}$ be the sequence such that $f_m = m + 1$ for $0 \le m < k$ and $f_m = f_{m-1} + f_{m-k}$ for $m \ge k$.

If $F(x)$ is the generating function for f_m then we have $F(x) = \frac{1-x^k}{(1-x)(1-x-x^k)}$. Let α be a root of $x^k = x^{k-1} + 1$. We'll show by induction that $f_{m+k} \ge \alpha^m$. For the base cases:

$$\begin{aligned}
f_k &= k + 1 \ge 1 = \alpha^0 \\
f_{k+1} &= k + 3 \ge \alpha^1 \\
&\vdots \\
f_{k+k} &= k + \frac{(k+1)(k+2)}{2} = k + k + 1 + \frac{k(k+1)}{2} \ge 2k + 1 + \alpha^{k-1} \ge \alpha^k.
\end{aligned}$$

In general, we have

$$f_{m+k} = f_{m+k-1} + f_m \ge \alpha^{m-1} + \alpha^{m-k} = \alpha^{m-k}(\alpha^{k-1} + 1) = \alpha^m.$$

Next we show that $f_{m+k} = k + \sum_{i=0}^{m} f_i$. The base case is clear, since $f_k = f_0 + k = k + 1$. For the induction step, we have

$$f_{m+k} = f_{m-1-k} + f_m = k \sum_{i=0}^{m-1} f_i + f_m = k + \sum_{i=0}^{m} f_i.$$

Observe that $s_i \ge f_{i+k}$ for $0 \le i < k$. Again, by induction, for $m \ge k$ we have

$$s_m = k + \sum_{i=0}^{m-k} s_i \ge k + \sum_{i=0}^{m-k} f_{i+k} \ge k + \sum_{i=0}^{m} f_i = f_{m+k}.$$

So in general, $s_m \ge f_{m+k}$. Putting it all together, we have

$$\begin{aligned}
\text{size}(x) &\ge s_m \\
&\ge k + \sum_{i=k}^{m} s_{i-k} \\
&\ge k + \sum_{i=k}^{m} f_i \\
&\ge f_{m+k} \\
&\ge \alpha^m.
\end{aligned}$$

Taking logs on both sides, we have

$$\log_\alpha n \ge m.$$

In other words, provided that α is a constant, we have a logarithmic bound on the maximum degree.

Problem 19-1 Alternative implementation of deletion

Professor Pisano has proposed the following variant of the FIB-HEAP-DELETE procedure, claiming that it runs faster when the node being deleted is not the node pointed to by $H.min$.

```
PISANO-DELETE(H, x)
    if x == H.min
        FIB-HEAP-EXTRACT-MIN(H)
    else y = x.p
        if y ≠ NIL
            CUT(H, x, y)
            CASCADING-CUT(H, y)
        add x's child list to the root list of H
        remove x from the root list of H
```

a. The professor's claim that this procedure runs faster is based partly on the assumption that line 7 can be performed in $O(1)$ actual time. What is wrong with this assumption?

b. Give a good upper bound on the actual time of PISANO-DELETE when x is not $H.min$. Your bound should be in terms of $x.degree$ and the number c of calls to the CASCADING-CUT procedure.

c. Suppose that we call PISANO-DELETE(H, x), and let H' be the Fibonacci heap that results. Assuming that node x is not a root, bound the potential of H' in terms of $x.degree$, c, $t(H)$, and $m(H)$.

d. Conclude that the amortized time for PISANO-DELETE is asymptotically no better than for FIB-HEAP-DELETE, evenwhen $x \ne H.min$.

a. It can take actual time proportional to the number of children that x had because for each child, when placing it in the root list, their parent pointer needs to be updated to be NIL instead of x.

b. Line 7 takes actual time bounded by $x.degree$ since updating each of the children of x only takes constant time. So, if c is the number of cascading cuts that are done, the actual cost is $O(c + x.degree)$.

c. We examine the number of trees in the root list $t(H)$ and the number of marked nodes $m(H)$ of the resulting Fibonacci heap H' to upper-bound its potential. The number of trees $t(H)$ increases by the number of children x had ($= x.degree$), due to line 7 of PISANO-DELETE(H, x). The number of marked nodes in H' is calculated as follows. The first $c - 1$ recursive calls out of the c calls to CASCADING-CUT unmarks a marked node (line 4 of CUT invoked by line 5 of CASCADING-CUT). The final cth call to CASCADING-CUT marks an unmarked node (line 4 of CASCADING-CUT), and therefore, the total change in marked nodes is $-(c-1)+1 = -c+2$. Therefore, the potential of H' is

$$\Phi(H') \le t(H) + x.degree + 2(m(H) - c + 2).$$

d. The asymptotic time is

$$\Theta(x.degree) = \Theta(\lg(n)),$$

which is the same asyptotic time that was required for the original deletion method.

Problem 19-2 Binomial trees and binomial heaps

The ***binomial tree*** B_k is an ordered tree (see Section B.5.2) defined recursively. As shown in Figure 19.6(a), the binomial tree B_0 consists of a single node. The binomial tree B_k consists of two binomial trees B_{k-1} that are linked together so that the root of one is the leftmost child of the root of the other. Figure 19.6(b) shows the binomial trees B_0 through B_4.

a. Show that for the binomial tree B_k,

1. there are 2^k nodes,
2. the height of the tree is k,
3. there are exactly $\binom{k}{i}$ nodes at depth i for $i = 0, 1, \ldots, k$, and

4. the root has degree k, which is greater than that of any other node; moreover, as Figure 19.6(c) shows, if we number the children of the root from left to right by $k-1, k-2, \ldots, 0$, then child i is the root of a subtree B_i.

A ***binomial heap*** H is a set of binomial trees that satisfies the following properties:

1. Each node has a key (like a Fibonacci heap).
2. Each binomial tree in H obeys the min-heap property.
3. For any nonnegative integer k, there is at most one binomial tree in H whose root has degree k.

b. Suppose that a binomial heap H has a total of n nodes. Discuss the relationship between the binomial trees that H contains and the binary representation of n. Conclude that H consists of at most $\lfloor \lg n \rfloor + 1$ binomial trees.

Suppose that we represent a binomial heap as follows. The left-child, right-sibling scheme of Section 10.4 represents each binomial tree within a binomial heap. Each node contains its key; pointers to its parent, to its leftmost child, and to the sibling immediately to its right (these pointers are NIL when appropriate); and its degree (as in Fibonacci heaps, how many children it has). The roots form a singly linked root list, ordered by the degrees of the roots (from low to high), and we access the binomial heap by a pointer to the first node on the root list.

c. Complete the description of how to represent a binomial heap (i.e., name the attributes, describe when attributes have the value NIL, and define how the root list is organized), and show how to implement the same seven operations on binomial heaps as this chapter implemented on Fibonacci heaps. Each operation should run in $O(\lg n)$ worst-case time, where n is the number of nodes in the binomial heap (or in the case of the UNION operation, in the two binomial heaps that are being united). The MAKE-HEAP operation should take constant time.

d. Suppose that we were to implement only the mergeable-heap operations on a Fibonacci heap (i.e., we do not implement the DECREASE-KEY or DELETE operations). How would the trees in a Fibonacci heap resemble those in a binomial heap? How would they differ? Show that the maximum degree in an n-node Fibonacci heap would be at most $\lfloor \lg n \rfloor$.

e. Professor McGee has devised a new data structure based on Fibonacci heaps. A McGee heap has the same structure as a Fibonacci heap and supports just the mergeable-heap operations. The implementations of the operations are the same as for Fibonacci heaps, except that insertion and union consolidate the root list as their last step. What are the worst-case running times of operations on McGee heaps?

a.

1. B_k consists of two binomial trees B_{k-1} .
2. The height of one B_{k-1} is increased by 1.
3. For $i = 0$, $\binom{k}{0} = 1$ and only root is at depth 0. Suppose in B_{k-1} , the number of nodes at depth i is $\binom{k-1}{i}$, in B_k, the number of nodes at depth i is $\binom{k-1}{i} + \binom{k-1}{i-1} = \binom{k}{i}$.
4. The degree of the root increase by 1.

b. Let $n.b$ denote the binary expansion of n. The fact that we can have at most one of each binomial tree corresponds to the fact that we can have at most 1 as any digit of $n.b$. Since each binomial tree has a size which is a power of 2, the binomial trees required to represent n nodes are uniquely determined. We include B_k if and only if the kth position of $n.b$ is 1. Since the binary representation of n has at most $\lfloor \lg n \rfloor + 1$ digits, this also bounds the number of trees which can be used to represent n nodes.

c. Given a node x, let $x.key$, $x.p$, $x.c$, and $x.s$ represent the attributes key, parent, left-most child, and sibling to the right, respectively. The pointer attributes have value NIL when no such node exists. The root list will be stored in a singly linked list.

- **MAKE-HEAP** initialize an empty list for the root list and return a pointer to the head of the list, which contains NIL. This takes constant time. To insert: Let x be a node with key k, to be inserted. Scan the root list to find the first m such that B_m is not one of the trees in the binomial heap. If there is no B_0, simply create a single root node x. Otherwise, union $x, B_0, B_1, \ldots, B_{m-1}$ into a B_m tree. Remove all root nodes of the unioned trees from the root list, and update it with the new root. Since each join operation is logarithmic in the height of the tree, the total time is $O(\lg n)$. MINIMUM just scans the root list and returns the minimum in $O(\lg n)$, since the root list has size at most $O(\lg n)$.

- **EXTRACT-MIN:** finds and deletes the minimum, then splits the tree Bm which contained the minimum into its component binomial trees $B_0, B_1, \ldots, B_{m-1}$ in $O(\lg n)$ time. Finally, it unions each of these with any existing trees of the same size in $O(\lg n)$ time.
- **UNION:** suppose we have two binomial heaps consisting of trees $B_{i_1}, B_{i_2}, \ldots, B_{i_k}$ and $B_{j_1}, B_{j_2}, \ldots, B_{j_m}$ respectively. Simply union orresponding trees of the same size between the two heaps, then do another check and join any newly created trees which have caused additional duplicates. Note: we will perform at most one union on any fixed size of binomial tree so the total running time is still logarithmic in n, where we assume that n is sum of the sizes of the trees which we are unioning.
- **DECREASE-KEY:** simply swap the node whose key was decreased up the tree until it satisfies the min-heap property. This method requires that we swap the node with its parent along with all their satellite data in a brute-force manner to avoid updating p attributes of the siblings of the node. When the data stored in each node is large, we may want to update p instead, which, however, will increase the running time bound to $O(\lg^2 n)$.
- **DELETE:** note that every binomial tree consists of two copies of a smaller binomial tree, so we can write the procedure recursively. If the tree is a single node, simply delete it. If we wish to delete from B_k, first split the tree into its constituent copies of B_{k-1}, and recursively call delete on the copy of B_{k-1} which contains x. If this results in two binomial trees of the same size, simply union them.

d. The Fibonacci heap will look like a binomial heap, except that multiple copies of a given binomial tree will be allowed. Since the only trees which will appear are binomial trees and B_k has $2k$ nodes, we must have $2k \le n$, which implies $k \le \lfloor \lg n \rfloor$. Since the largest root of any binomial tree occurs at the root, and on B_k it is degree k, this also bounds the largest degree of a node.

e. INSERT and UNION will no longer have amortized $O(1)$ running time because CONSOLIDATE has runtime $O(\lg n)$. Even if no nodes are consolidated, the runtime is dominated by the check that all degrees are distinct.

Since calling UNION on a heap and a single node is the same as insertion, it must also have runtime $O(\lg n)$. The other operations remain unchanged.

Problem 19-3 More Fibonacci-heap operations

> We wish to augment a Fibonacci heap H to support two new operations without changing the amortized running time of any other Fibonacci-heap operations.
>
> **a.** The operation $\text{FIB-HEAP-CHANGE-KEY}(H, x, k)$ changes the key of node x to the value k. Give an efficient implementation of $\text{FIB-HEAP-CHANGE-KEY}$, and analyze the amortized running time of your implementation for the cases in which k is greater than, less than, or equal to $x.key$.
>
> **b.** Give an efficient implementation of $\text{FIB-HEAP-PRUNE}(H, r)$, which deletes $q = \min(r, H.n)$ nodes from H. You may choose any q nodes to delete. Analyze the amortized running time of your implementation. (Hint: You may need to modify the data structure and potential function.)

a. If $k < x.key$ just run the decrease key procedure. If $k > x.key$, delete the current value x and insert x again with a new key. For the first case, the amortized time is $O(1)$, and for the last case the amortized time is $O(\lg n)$.

b. Suppose that we also had an additional cost to the potential function that was proportional to the size of the structure. This would only increase when we do an insertion, and then only by a constant amount, so there aren't any worries concerning this increased potential function raising the amortized cost of any operations. Once we've made this modification, to the potential function, we also modify the heap itself by having a doubly linked list along all of the leaf nodes in the heap.

To prune we then pick any leaf node, remove it from it's parent's child list, and remove it from the list of leaves. We repeat this $\min(r, H.n)$ times. This causes the potential to drop by an amount proportional to r which is on the order of the actual cost of what just happened since the deletions from the linked list take only constant amounts of time each. So, the amortized time is constant.

Problem 19-4 2-3-4 heaps

Chapter 18 introduced the 2-3-4 tree, in which every internal node (other than possibly the root) has two, three, or four children and all leaves have the same depth. In this problem, we shall implement ***2-3-4 heaps***, which support the mergeable-heap operations.

The 2-3-4 heaps differ from 2-3-4 trees in the following ways. In 2-3-4 heaps, only leaves store keys, and each leaf x stores exactly one key in the attribute $x.key$. The keys in the leaves may appear in any order. Each internal node x contains a value $x.small$ that is equal to the smallest key stored in any leaf in the subtree rooted at x. The root r contains an attribute $r.height$ that gives the height of the tree. Finally, 2-3-4 heaps are designed to be kept in main memory, so that disk reads and writes are not needed.

Implement the following 2-3-4 heap operations. In parts (a)–(e), each operation should run in $O(\lg n)$ time on a 2-3-4 heap with n elements. The UNION operation in part (f) should run in $O(\lg n)$ time, where n is the number of elements in the two input heaps.

a. MINIMUM, which returns a pointer to the leaf with the smallest key.

b. DECREASE-KEY, which decreases the key of a given leaf x to a given value $k \le x.key$.

c. INSERT, which inserts leaf x with key k.

d. DELETE, which deletes a given leaf x.

e. EXTRACT-MIN, which extracts the leaf with the smallest key.

f. UNION, which unites two 2-3-4 heaps, returning a single 2-3-4 heap and destroying the input heaps.

a. Traverse a path from root to leaf as follows: At a given node, examine the attribute $x.small$ in each child-node of the current node. Proceed to the child node which minimizes this attribute. If the children of the current node are leaves, then simply return a pointer to the child node with smallest key. Since the height of the tree is $O(\lg n)$ and the number of children of any node is at most 4, this has runtime $O(\lg n)$.

b. Decrease the key of x, then traverse the simple path from x to the root by following the parent pointers. At each node y encountered, check the attribute $y.small$. If $k < y.small$, set $y.small = k$. Otherwise do nothing and continue on the path.

c. Insert works the same as in a B-tree, except that at each node it is assumed that the node to be inserted is 'smaller' than every key stored at that node, so the runtime is inherited. If the root is split, we update the height of the tree. When we reach the final node before the leaves, simply insert the new node as the leftmost child of that node.

d. As with B-TREE-DELETE, we'll want to ensure that the tree satisfies the properties of being a 2-3-4 tree after deletion, so we'll need to check that we're never deleting a leaf which only has a single sibling. This is handled in much the same way as in chapter 18. We can imagine that dummy keys are stored in all the internal nodes, and carry out the deletion process in exactly the same way as done in exercise 18.3-2, with the added requirement that we update the height stored in the root if we merge the root with its child nodes.

e. EXTRACT-MIN simply locates the minimum as done in part (a), then deletes it as in part (d).

f. This can be done by implementing the join operation, as in Problem 18-2 (b).