# Assignment 9

14CH3FP18

20 March 2016

**Abstract**

Simulation of reading large chunks of data using Fibonacci Heaps

## Part I
# Introduction

Fibonacci heaps are interesting the!oretically because they have asymptotically good runtime guarantees for many operations. In particular, **insert, peek, and decrease-key** all run in amortized $O(1)$ time. **dequeueMin** and **delete** each run in amortized $O(lgn)$ time. This allows algorithms that rely heavily on **decrease-key** to gain significant performance boosts. For example, Dijkstra's algorithm for single-source shortest paths can be shown to run in $O(m+nlgn)$ using a Fibonacci heap, compared to $O(mlgn)$ using a standard binary or binomial heap. Internally, a Fibonacci heap is represented as a circular, doubly-linked list of trees obeying the min-heap property. Each node stores pointers to its parent (if any) and some arbitrary child. Additionally, every node stores its degree (the number of children it has) and whether it is a "marked" node. Finally, each Fibonacci heap stores a pointer to the tree with the minimum value.

To insert a node into a Fibonacci heap, a singleton tree is created and merged into the rest of the trees. The **merge** operation works by simply splicing together the doubly-linked lists of the two trees, then updating the min pointer to be the smaller of the minima of the two heaps. Peeking at the smallest element can therefore be accomplished by just looking at the min element. All of these operations complete in $O(1)$ time.

The tricky operations are **dequeueMin** and **decreaseKey**. **dequeueMin** works by removing the root of the tree containing the smallest element, then merging its children with the topmost roots. Then, the roots are scanned and merged so that there is only one tree of each degree in the root list. This works by maintaining a dynamic array of trees, each initially null, pointing to the roots of trees of each dimension. The list is then scanned and this array is populated. Whenever a conflict is discovered, the appropriate trees are merged together until no more conflicts exist. The resulting trees are then put into the root list. A clever analysis using the

potential method can be used to show that the amortized cost of this operation is $O(lgn)$.

The other hard operation is **decreaseKey**, which works as follows. First, we update the key of the node to be the new value. If this leaves the node smaller than its parent, we're done. Otherwise, we cut the node from its parent, add it as a root, and then mark its parent. If the parent was already marked, we cut that node as well, recursively mark its parent, and continue this process. This can be shown to run in $O(1)$ amortized time using yet another clever potential function. Finally, given this function, we can implement **delete** by decreasing a key to $-\infty$, then calling **dequeueMin** to extract it.

# Part II
# Heap Operations

## 1 Inserting a node

The following procedure inserts node x into Fibonacci heap H, assuming of course that the node has already been allocated and that key[x] has already been filled in.

FIB-HEAP-INSERT(H, x)
1 degree[x] 0
2 p[x] NIL
3 child[x] NIL
4 left[x] x
5 right[x] x
6 mark[x] FALSE
7 concatenate the root list containing x with root list H
8 if min[H] = NIL or key[x] < key[min[H]]
9 then min[H] x
10 n[H] n[H] + 1

To determine the amortized cost of FIB-HEAP-INSERT, let H be the input Fibonacci heap and H' be the resulting Fibonacci heap. Then, t(H') = t(H) + 1 and m(H') = m(H), and the increase in potential is

((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1 . Since the actual cost is O(1), the amortized cost is O(1) + 1 = O(1).

## 2 Finding the minimum node

The minimum node of a Fibonacci heap H is given by the pointer min[H], so we can find the minimum node in O(1) actual time. Because the potential of H does not change, the amortized cost of this operation is equal to its O(1) actual cost.

## 3 Uniting two Fibonacci heaps

The following procedure unites Fibonacci heaps H1 and H2, destroying H1 and H2 in the process.

FIB-HEAP-UNION(H1,H2)
1 H MAKE-FIB-HEAP()
2 min[H] min[H1]
3 concatenate the root list of H2 with the root list of H
4 if (min[H1] = NIL) or (min[H2] NIL and min[H2] < min[H1])
5 then min[H] min[H2]
6 n[H] n[H1] + n[H2]
7 free the objects H1 and H2
8 return H Lines 1-3 concatenate the root lists of H1 and H2 into a new root list H. Lines 2, 4, and 5 set the minimum node of H, and line 6 sets n[H] to the total number of nodes. The Fibonacci heap objects H1 and H2 are freed in line 7, and line 8 returns the resulting Fibonacci heap H. As in the FIB-HEAP-INSERT procedure, no consolidation of trees occurs.

The change in potential is
(H) - ((H1) + (H2))
= (t(H) + 2m(H)) - ((t(H1) + 2 m(H1)) + (t(H2) + 2m(H2)))
= 0, because t(H) = t(H1) + t(H2) and m(H) = m(H1) + m(H2). The amortized cost of FIB-HEAP-UNION is therefore equal to its O(1) actual cost.

## 4 Extracting the minimum node

The process of extracting the minimum node is the most complicated of the operations presented in this section. It is also where the delayed work of consolidating trees in the root list finally occurs. The following pseudocode extracts the minimum node. The code assumes for convenience that when a node is removed from a linked list, pointers remaining in the list are updated, but pointers in the extracted node are left unchanged. It also uses the auxiliary procedure MELD, which will be presented shortly.

FIB-HEAP-EXTRACT-MIN(H)
1 z min[H]
2 if z NIL
3 then for each child x of z
4 do add x to the root list of H
5 p[x] NIL
6 remove z from the root list of H
7 if z = right[z]
8 then min[H] NIL
9 else min[H] right[z]
10 MELD(H)

11 n[H] n[H] - 1
12 return z

Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct degree value.

1. Find two roots x and y in the root list with the same degree, where key[x] key[y].

2. Link y to x: remove y from the root list, and make y a child of x. This operation is performed by the FIB-HEAP-LINK procedure. The field degree[x] is incremented, and the mark on y, if any, is cleared.

The procedure MELD uses an auxiliary array A[0 . . D(n[H])]; if A[i] = y, then y is currently a root with degree[y] = i.

MELD(H)
1 for i 0 to D(n[H])
2 do A[i] NIL
3 for each node w in the root list of H
4 do x w
5 d degree[x]
6 while A[d] NIL
7 do y A[d]
8 if key[x] > key[y]
9 then exchange x y
10 FIB-HEAP-LINK(H,y,x)
11 A[d] NIL
12 d d + 1
13 A[d] x
14 min[H] NIL
15 for i 0 to D(n[H])
16 do if A[i] NIL
17 then add A[i] to the root list of H
18 if min[H] = NIL or key[A[i]] < key[min[H]]
19 then min[H] A[i]
FIB-HEAP-LINK(H, y, x)
1 remove y from the root list of H
2 make y a child of x, incrementing degree[x]
3 mark[y] FALSE

The actual cost of extracting the minimum node can be accounted for as follows. An O(D(n)) contribution comes from there being at most D(n) children of the minimum node that are processed in FIB-HEAP-EXTRACT-MIN and from the work in lines 1-2 and 14-19 of Meld. It remains to analyze the contribution from the for loop of lines 3-13. The size of the root list upon calling Meld is at most D(n) + t(H) - 1, since it consists of the original t(H) root-list nodes, minus the extracted root node, plus the children of the extracted node, which number at most D(n). Every time through the while loop of lines 6-12, one of the roots is linked

to another, and thus the total amount of work performed in the for loop is at most proportional to D(n) + t(H). Thus, the total actual work is O(D(n) + t(H)).

The potential before extracting the minimum node is t(H) + 2m(H), and the potential afterward is at most (D(n) + 1) + 2m(H), since at most D(n) + 1 roots remain and no nodes become marked during the operation. The amortized cost is thus at most

O(D(n) + t(H)) + ((D(n) + 1) + 2m(H)) - (t(H) + 2m(H))

= O(D(n)) + O(t(H)) - t(H)

= O(D(n)), since we can scale up the units of potential to dominate the constant hidden in O(t(H)). Intuitively, the cost of performing each link is paid for by the reduction in potential due to the link reducing the number of roots by one.