Binary Heaps, Fibonacci Heaps and Dijkstras shortest path

Kristoffer Just Andersen, 20051234

Troels Leth Jensen, 20051234

Morten Krogh-Jespersen, 20022362

Project 1, Advanced Data Structures 2013, Computer Science
October 2013
Advisor: Gerth Stølting Brodal

AARHUS
UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Contents

Chapter 1

Introduction

needs content

Chapter 2

Binary heaps

needs content

2.1 Binary heap with array

needs content

2.2 Implementing decrease key

needs content

2.3 Time-complexity for binary heap with array

2.4 Binary heap with pointers

needs content

2.5 Implementing decrease key

needs content

2.6 Time-complexity for binary heap with pointers

2.7 Testing correctness of Binary Heaps

needs content

Chapter 3

Fibonacci heaps

In this chapter we focus on Fibonacci heaps, which is a data structure that has a forest of rooted trees as opposed to a binary heap that only has one tree [1]. The data structure was invented by Michael L. Fredman and Robert Endre Tarjan and was published in the Journal of ACM in 1987. It has it name because the size of any subtree in a Fibonacci heap will be lower bounded by F_{k+2} where k is the degree of the root in that subtree and F is the Fibonacci function. Below is the time-complexities of each of the heap operations listed:

Operation	Binary heap	Fibonacci heap v1 (amortized)	Fibonacci heap v2 (amortized)
MakeHeap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
FindMin	$\Theta(1)$	$\Theta(1)$	$\mathrm{O}\left(l(\lg(\frac{n}{l})+1)\right)$
Insert	$\Theta(\lg n)$	$\Theta(1)$	$\Theta(1)$
DeleteMin	$\Theta(\lg n)$	$\mathrm{O}(\lg n)$	$\Theta(1)$
DecreaseKey	$\Theta(\lg n)$	$\mathrm{O}(1)$	$\mathrm{O}(1)$
Delete	$\Theta(\lg n)$	$\mathrm{O}(\lg n)$	$\Theta(1)$
Meld	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

3.1 Fibonacci heap version 1

The first Fibonacci heap variant we present is the original version proposed in FT87. A potential function is used to analyze the perfomance, thus the above stated time-complexities are amortized.

Our implementation pretty much follows from the article, with few exceptions. The article do not specify exactly how a node is found from an item in constant time, so we decided to place a pointer on the item. Melding is not totally destructable since we join the heap in of the two existing heaps and return an arbitrary one.

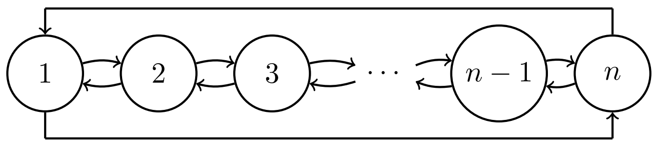
The article mentions that Delete takes $\mathrm{O}(1)$ if the node to remove is not the min-node and without cascading deletes. The children of the node to delete must be moved up onto the root which can only be done in constant time if every children has a pointer to a parent pointer. In this way, we only have to change one pointer to update all parent pointers for the children of the beforementioned node. Since the running time of the algorithm is amortized $O(\lg n)$ we chose a simpler version where we just update all parent pointers.

3.2 Worst case time-complexity for Fib heap v1

There are three operations where changes to the potential occurs, and thus, the stated times are amortized for DeleteMin, DecreaseKey and Delete. Below we illustrate the worst-case for each of these operations by showing a configuration and how that configuration can be obtained from a sequence of operations :

3.2.1 Worst case time-complexity for DeleteMin

The worst configuration for DeleteMin is when all the nodes in the heap is at the root in a linked list:



This configuration can be achievede by just calling insert n times. For simplicity, let us assume that n is odd, and a call to DeleteMin happens. In the above example 1 will be removed and we are left with $n-1$ root nodes to link. This results in $n-1$ key comparisons, but all the trees of rank 1 will be joined too and this will continue until no trees of duplicate size is found.

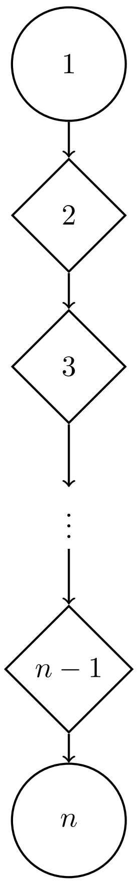
$$\#\text{ of operations} = O(n-1) + O\left(\frac{n-1}{2}\right) + O\left(\frac{n-1}{2^2}\right) + \cdots + O\left(\frac{n-1}{2^{\lg(n-1)-1}}\right)$$

which is $\Theta(n)$.

3.2.2 Worst case time-complexity for Delete and DecreaseKey

If Delete is invoked with the min-node as argument then Delete calls DeleteMin, therefore, the worst case for Delete is $\Theta(n)$, but we will show that without the min-node as argument, we still end up with $\Theta(n)$.

If we delete a child to an arbitrary node x we mark x if is not marked and if it is marked, we cut x from its parent, move the subtree formed by x to the root and try to mark the previous parent of x. This could result in cascading deletes. Therefore, the worst situation would be the following configuration:

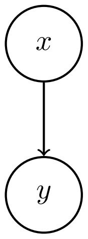


where a diamond is modelling a marked node. If either DecreaseKey or Delete is called with an item corresponding to node n a cascading delete will begin and will not stop until it reaches 2 in this example. The amount of operations is therefore the entire chain:

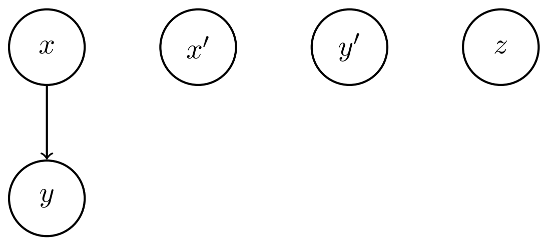
$$\text{length of chain} = n - 1$$

which is $\Theta(n)$.

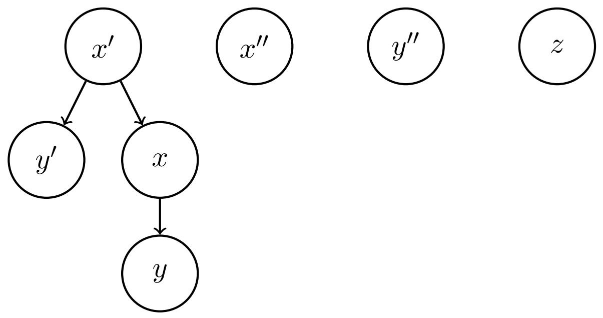
Such a configuration can be obtained by calling Insert with two very high numbers x and y and one smaller and call DeleteMin:



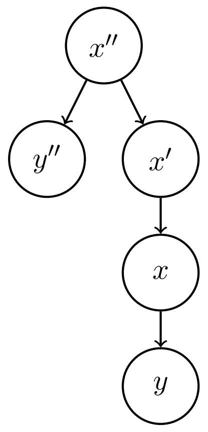
If we insert three new items x', y' z with keys slightly smaller where z is the smallest key we will have the following forest:



If we call DeleteMin and call insert with three new items, we can get the following configuration:



where z again is the smallest key. We can Delete y' and call DeleteMin, which now gives:



By using this approach we can easily build up a single chain of height n.

3.3 Fibonacci heap version 2

needs content

3.4 Worst case time-complexity for Fib heap v2

needs content

3.5 Testing correctness of Fibonacci Heaps

needs content

Chapter 4

Test-results

needs content

Chapter 5

Dijkstra

needs content

Chapter 6

Binary heap vs Fibonacci heap

needs content

Chapter 7

Test-results

needs content

Chapter 8

Conlusion

needs contents

Bibliography

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