Advanced Datastructures: Project 1

Jan H. Knudsen (20092926)
Roland L. Pedersen (20092817)
Kris V. Ebbesen (20094539)

October 2, 2013

Contents

1 Introduction

Heaps have many applications: as simple priority queues, for graph algorithms, and even scheduling. As such, is is important to chose not only a fast heap, but the right heap for the job.

In this project, we implement and test two distinct types of heap, the binary heap and the fibonacci heap. The binary heap is very simple to understand, and has generally low constant factors. The fibonacci heap on the other hand is somewhat more complicated, and has quite large constant factors involved in the running time, but makes up for this by having insert and decreasekey operations being amortized constant time.

Finally, we see how this affects Dijkstras single-source shortest-path algorithm, on data sets that favour each of the heaps, and try to show a that choosing the correct heap indeed does make a significant difference in performance.

2 Heap Implementation Details

2.1 Binary Heap

The Binary Heap was implemented as a binary heap based on the description from [1]. Implementing the Binary Heap on top of an array, as opposed to modelling a tree, has several important advantages, and some disadvantages.

Note that finding the children of a node located at any given position of the array can be easily done by multiplying the position by 2 to find the left child, and additionally adding 1 to find the right child. Vice versa to find parent.

Space per node Since we do not need to store pointers to children and parent nodes, we can save quite a bit of space in the node structure.

Total Space Unfortunately, the array backing the heap needs to be big enough to support the entire heap. We ensure this by specifying it in the constructor, and allocating the array during the heap construction. This means that the heap will need to keep a large chunk of memory alive throughout its lifetime. We save some of this memory by storing only an array of pointers, and allocating space on the heap for the actual nodes as we go along.

Note that this problem can be somewhat avoided by growing or shrinking the array as the size changes (or simply use the vector class), but that might change the worst-case running time of the operations.

Finding the next open position Finding out where to initially place a new node in the heap is easy when using an array, as the index into the array can be easily computed from the size.

Keeping track of nodes When decreasing a key in an array-backed Binary Heap, we need to find the position of the node in the array. This unfortunately requires us to keep track of the index of a node as an additional field in the node structure.

2.2 Fibonacci Heap

The Fibonacci Heap was implemented following the description from [2]. The Fibonacci heap was implemented over several iterations, and some interesting specifics that were left out of the description from Cormen [2] were changed a lot during the development.

Children Lists The choice of data structure for the children of a node were non-specified, and as such, several options were tried. In the end, it was decided that they would use pointers to locate children, parents and siblings.

Roots List Just as the children lists, the choice of root list structure was left open. Like for the case of the children lists, it was decided to use pointers, and in fact the roots list is represented using the same pointers from the siblings, since roots have no siblings.

Maximum degree of a Node During the DeleteMin operation, we need to keep track of unique nodes of specific degrees, this could either be done using a map, or a sufficiently large array. In case of the array, we need to estimate what the largest degree possible in the roots list. This is done by the function $(\lceil \log_2(n) \rceil + 1) * 2$. This function is chosen carefully in that it will grow faster than that of the Fibonacci numbers, who grow by approximately log_ϕ, but only slightly so. Note that this function is much faster to compute than true logarithms, and could even be sped up more by using fast msb, or calling the native `msb` instruction on newer architectures.

2.2.1 Profiling and incremental development

Profiling of the Fibonacci Heap was done using the Very Sleepy profiler [3]. This profiling helped us to do the development in an iterative manner.

At first, a functioning Fibonacci Heap was implemented, using the unordered_set datastructure from the C++11 stdlib to store children and roots and using a large constant as maximum the degree.

Profiling showed that these sets were absurdly slow, especially when iterating and constructing new sets. They were then replaced by C++11 forward_lists, using an iterator stored at the node for child deletion, and doing a full rebuild of the root list at DeleteMin operations. This proved to be a great speedup, but profiling still showed that the stdlib data structures took up most of the time during tests.

Finally, the stdlib data structures were replaced by pointers, and the heap started showing good performance, with most of the time spent doing work on the the algorithm itself, or in operating system calls.

Along the way, an approximation for maximum degree was introduced, but it did not seem to make a massive difference in performance.

2.2.2 Future work for the Fibonacci Heap

If we were to make the Fibonacci Heap a fully functional we would most likely add the following functionality to it

IncreaseKey Increasing the key in a Fibonacci Heap can be done much like the DecreaseKey operation.

MergeHeaps Heap merging is fairly easy for Fibonacci Heaps if their root list are implemented correctly. Adding either a pointer to the last element of the roots list, or linking the first and last element using the sibling pointers would allow $O(1)$ heap merges.

Fast log_2 The approximation for the maximum height is done by bit-shifting to find log_2.this could most likely be improved greatly by using either the fast msb from the lectures or the built-in msb instruction of modern processors.

3 Theoretical Worst-case Running Times

3.1 Binary Heaps

3.1.1 MakeHeap

Constructing an empty Binary Heap requires only a constant amount of work. If we look at the heap as a graph of nodes, we simply need to specify that the size is 0 and that there is no root. As such, this operation will be $O(1)$.

Note that an array-backed, Binary Heap will need to initialize an array of size $O(n)$, which might not be a constant-time operation, depending on the underlying hardware and operating system.

3.1.2 FindMin

Finding the minimum in an Binary Heap is a simple matter of locating the root of the heap. This will either be stored as a pointer in the heap, or, in the case of an array-backed heap, at the start of the array. As such, the operation is $O(1)$.

3.1.3 Insert

Inserting an element in a Binary Heap is done by inserting the element at the first free position at the bottom of the heap and bubbling it up through the tree by swapping it with its parent until it is at the correct position. Since we perform only a constant amount of operations at each level of the heap, and the heap is logarithmic in height, this operation take $O(\log(n))$ time.

3.1.4 DeleteMin

Deleting the minimum element of a Binary Heap is done by swapping the root and the last element of the heap, removing the last element (the previous root), and swapping the new root with its largest child until it is at the correct position in the heap. Much like the Insert operation, we perform only a constant amount of work for each level of the heap, and this makes the operation take $O(\log(n))$ time.

3.1.5 DecreaseKey

Decreasing a key in a Binary Heap can be done by decreasing the key value of the specified node, then bubbling it up in the tree by repeatedly swapping it with its parents as long as the parent has higher key. Since this is the same complexity as an insert, it takes $O(\log(n))$ time.

3.2 Fibonacci Heap

3.2.1 MakeHeap

Initializing a Fibonacci heap is a matter of setting a few variables to their default values, such as setting the size to zero, the root list to an empty list and no value for the minimum node. This can be done in $O(1)$ time.

3.2.2 FindMin

A Fibonacci Heap keeps a pointer to the minimum element at all times. This makes finding the minimum element an $O(1)$ operation.

3.2.3 Insert

Inserting an element into a Fibonacci Heap is done is done by creating a new node, and appending it to the root list. If the root list is a data structure that supports inserts in constant time, this can be done in $O(1)$ time.

3.2.4 DeleteMin

Deleting the minimum of a Fibonacci Heap is done by adding all of the minimum nodes children to the root list, removing the minimum, and then continuously linking root nodes until only root node of each degree exists. After this, we look through the root list, and set the minimum to be the root with the lowest key.

This leads to the following time requirements.

Finding the minimum As shown earlier, $O(1)$.

Adding all the children to root list Since the number of children to a node is limited by its degree, and the maximum degree of a node is logarithmic with respect to n, we can do this in $O(\log(n))$ time as long as we can insert elements into our root list in constant time.

Linking roots Since linking two root nodes together takes time, and the linking of two roots removes one of them from the root list, we have that we can link all of the roots in $O(\#roots)$ time. Since we can have only one root for each element in the heap, this can take $O(n)$ time.

Finding the new minimum After we have linked the roots together, we have at most one root for each degree, and since the maximum degree of a root if logarithmic, we have that we can search through the new list of roots in $O(\log(n))$ time.

This gives us a total worst case running time of the DeleteMin operation of $O(n)$ time.

3.2.5 DecreaseKey

Decreasing the key of a node in a Fibonacci Heap is done by decreasing the key of the node and removing it from the child list of the parent, then adding it to the root list. We then continue to do this for the parent as long as we keep encountering marked parents, and mark the final, non-marked, parent we encounter.

In the worst case this requires us to traverse all the way up to the root of the tree our node belonged to. Note that the height of a tree is limited by the degree of its root, which is at most logarithmic. This means that the operation takes at most $O(\log(n))$ time.

4 Single Operation Experiments

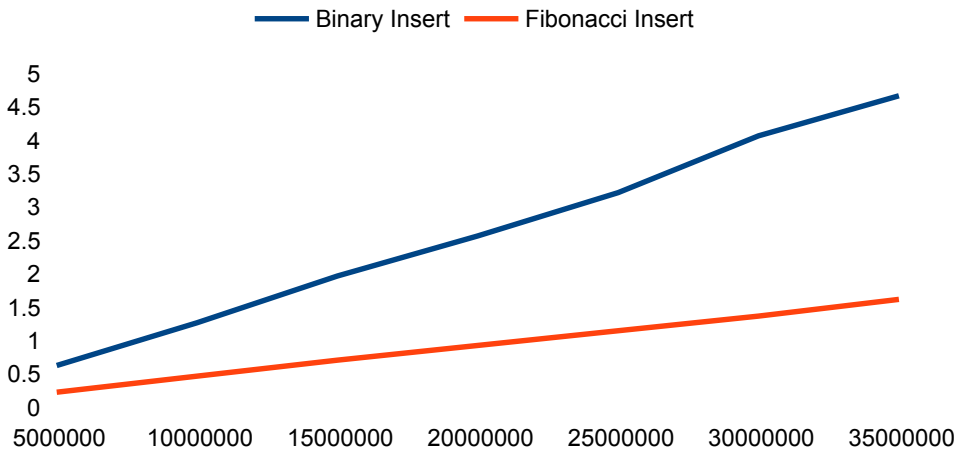
We have attempted to experimentally determine the running times, or rather, the difference in running time of our implementations of `Insert`, `DeleteMin` and `DecreaseKey` in the two types of heaps. We have done so by running each operation 5, 10, 15, 20, 25, 30 and 35 million times on each type of heap. We have also counted the number of comparisons done by each heap for the operations.

4.1 Running Times

We have measured the running time using the difference of two calls to `clock()`, one right before and one right after the experiment, converting into seconds by division with CLOCKS_PER_SEC.

4.1.1 Insert

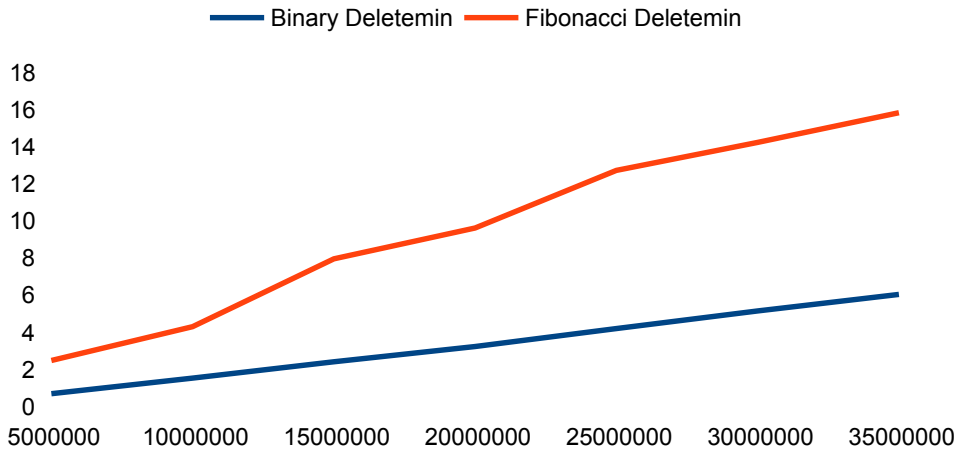
Testing the running time of the insert operation was done by inserting elements with integer keys between 0 and $n-1$ in reverse order, so that each element inserted had the lowest key, to simulate a worst-case situation where each newly inserted element is the new minimum.



We can see that the fibonacci heap is considerably faster for this operation.

4.1.2 DeleteMin

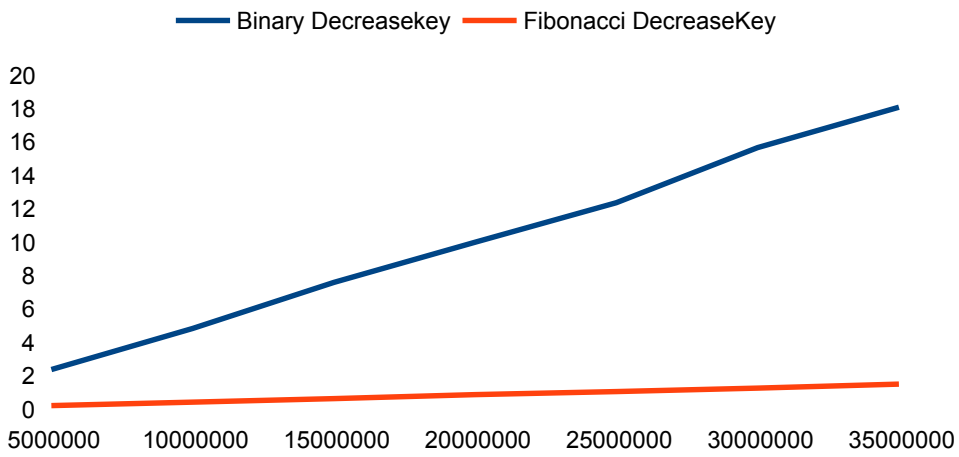
Testing the running time of the deletemin operation consisted of first inserting the relevant amount of elements exactly like in the test of inserts, then resetting the comparison count, and then running DeleteMin on the heap an equal amount of times only running the clock while the deletemin operations were running.



We can see that, for this operation, the binary heap is the faster one.

4.1.3 DecreaseKey

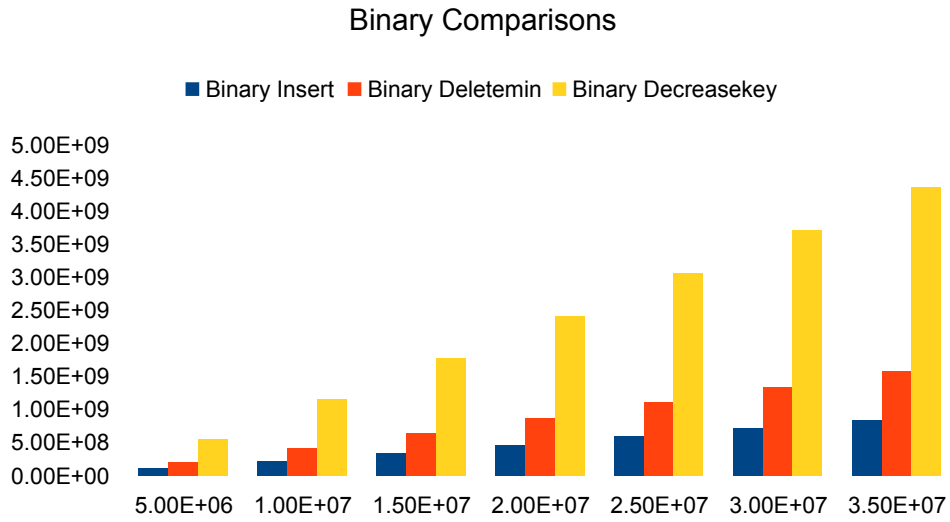
The setup for this test was the same as for DeleteMin. The test itself consisted of going through each inserted element, decreasing its key to a new minimum (one below the previous minimum). This was repeated 5 times.

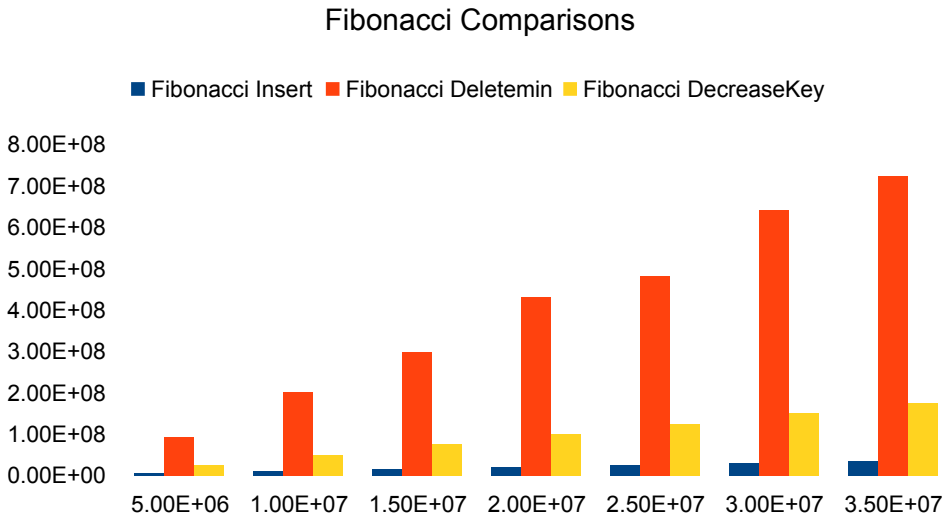


For this operation, we again see the fibonacci heap being much faster.

4.2 Comparisons

Counting the comparisons was done simply by keeping a counter variable in the heap and upcounting it whenever a comparison was done.





We can see that the binary heap spends most comparisons in its decreasekey operation, whereas the fibonacci heap spends them in the deletemin operation because it defers fixing up the structure after insert and decreasekey operations until it needs to do so after deleting a node in the deletemin operation.

Also note the axis: even though the fibonacci heap does a lot of comparisons when doing deletemin, it is still less than what the binary heap does with the same amount of elements.

4.3 Conclusions from the running times

From our data, we see that our implemented data structures perform as expected from theoretical analysis, with the fibonacci heap heavily outperforming the binary heap on inserts and decreasekey, while being somewhat slower on deletes, due to being more complex in general. Note that one of the reasons why the fibonacci heap performs exceedingly well on decreasekey is that the tree structures are never constructed by a deletemin, instead the entire heap is one huge list of single nodes.

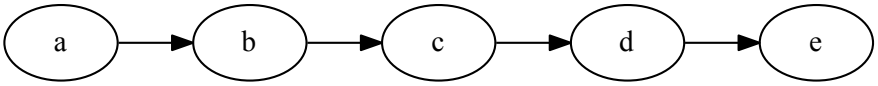
5 Dijkstra Implementation Details

Dijkstra's Algorithm is implemented in a very straightforward way. It is given an array of nodes, and computes the single-source shortest path from the first of these elements. We chose to represent neighbours in a list structure, instead of a distance matrix, to better accomodate sparse data sets.

6 Graph Families

When choosing test sets for Dijkstras Single-Source Shortest Paths that hopefully will exhibit different running times based on the performance of the underlying heaps DecreaseKey running time it is important to maximize the difference in the amount of edges. In order to do this, we wish to chose one test set that is very sparse, and one that is very dense.

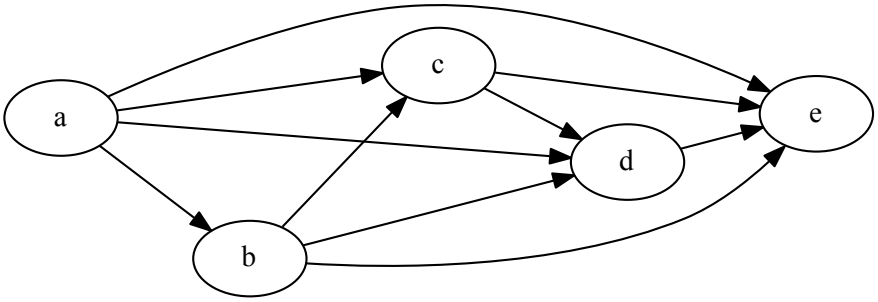
6.1 Sparse Graph



As our sparse graph, we chose a line of nodes, where each node has both in and out degree of 1, except for the source, which has in-degree 0, and the end, which has out-degree 0. Such a graph will have $O(n)$ edges, and will in fact contain the minimal amount of edges required to keep the graph fully connected.

Alternatives to this graph could be a cycle of nodes, any directed tree, or a star graph, with edges going out from the center nodes.

6.2 Dense Graphs



As our dense graph, we have chosen a modified version of our sparse line of nodes, where each node has had an outgoing edge to every node in its transitive closure that it did not already have an outgoing edge to added. Such a graph will contain $O(n^2)$ edges. Note that we could add even more edges to this graph, but since Dijkstra's algorithm will not call DecreaseKey for already visited nodes, we would not increase the amount of calls to DecreaseKey.

An alternative to this graph could have been complete digraph, to fully maximize the amount of edges.

7 Dijkstra Experiments

7.1 Experimental Data

In testing our Dijkstra implementations, we chose to generate the graphs described in the previous sections, as they are both fitting examples of different running time cases, and fairly simple to construct. For the sparse graph, we run the data from 10000 to 200000 nodes in increments of 10000, while we run it with 1000 to 20000 nodes in increments of 1000 when working on the dense graphs.

7.2 Conclusion from the results

From the results we can see, that the fibonacci heap is slower than the binary heap on the spars graph, which stems from the slower deletemin in the fibonacci heap, while decreasekey has no effect.

When running on the sparse graph, we can see the fibonacci heap growing at a slower pace than the binary heap, since it uses constant time per decreasekey instead of logarithmic. Even though the running time difference does not appear to be very big on the dense graph, it is actually quite impressing when we look at how heavily binary outperforms fibonacci on the sparse graph. On larger input the running time differences for the sparse graph would have likely favoured fibonacci even more, but unfortunately we did not have the required memory to run much larger data sets.

7.3 A note on great big data sets

Actually, it would have been possible to run with much larger data sets, if we had chose to not actually store the neighbour information, but use a weighting function instead. Since the distances between two nodes is fairly predictable, this could have greatly reduced the memory consumption.

References

[1] Robert Sedgewick *Algorithms in C, Parts 1-4.* Addison-Wesley, 1998, Third edition.

[2] Cormen et al. *Introduction To Algorithms.* MIT Press, 2009, Third edition.

[3] http://www.codersnotes.com/sleepy