Assignment 8: Heap

Mahad Ahmed

Fall 2022

Introduction

In this assignment we implemented a heap also called a priority queue. This assignment was quite long and some implementations were rather difficult. A priority queue is often implemented as a tree, a data structure we have worked with a lot in previous assignments. When it is a priority queue is implemented in using a tree is called a *heap*. In this assignment we are going to go thorugh each implementation and discuss it's pros and cons.

A heap

A heap as I said is a tree structure that has a root node which has the highest priority. The child nodes branches is called heaps. We have a min heap and a max heap. By min heap it means that the root the has the highest priority is also the smallest in value. For a max heap is the opposite. We implemented a min heap. A heap is a complete binary tree. A binary tree is a tree where each node can only have two children.

Balancing a tree

A problem with heap is keeping the tree balanced. Because if we add the same every time the tree will be unbalanced and look more like a linked list. To counter this we need to keep track of every node in each sub-tree and then add in the sub-tree with least number of nodes.

Add method

To insert an node into the tree we just look for the first empty spot. So if we just traverse through the tree until we find an empty spot somewhere. The insert will go from left to right when filling the tree. But when we add a node that is not in a right spot because we are implementing a priority queue then what shall we do?

Let say we have a binary tree with 2 as a root. It's children are nodes 4 and 8. The node with 4 has a child which is a node 9. So now we have a tree with 2 at the root, left child of 2 is 4 and right child of 2 is 8. Then the left child of 4 is 9. Now lets say we want to add 3. So firstly we need to find the first available spot for this node which is going to be the right child och node 4. But now the priority is not achieved as 3 is a smaller element than 4, 8, and 9. So what we have to do now is to pop i up into it's right place. Now 2 is still the root as it is the smallest element. It's children are now instead 3 as the left child of 2 and still 8 as the right child. Node 3 children are now 9 and 4. Now the priority is achieved.

Remove

To remove a node in the heap is pretty simple but what happens after is kinda difficult as you have many corner cases to consider. To keep it simple though we remove the root as this is the smallest element in a min heap. Then what people think to do is to just take the left child of the root and promote it because it is the smallest. But that will result in an open space where the left child was. This we lead to an incomplete tree. Because it is a heap it has to be a complete binary tree. What to do then?

What we do is after we remove the the root we take the latest node we added and put that one as root. Then we demote the root node to it's right place. Doing this will ensure we still have a heap.

Benchmarks

In this benchmark the instruction was to add 64 random elements between 0 and 100 to the heap and then return the depth.

Depth	**Add operations**
0	1
1	3
2	4
3	9
4	15
5	35

Table 1: Benchmark, heap

Array implementation

You could also implement a heap using an array. Although arrays are often looked at as a series of boxes linked together it can use it's indices to search through an array as a tree. A heap and an array go hand in hand. How will we implement it though? Let's say that we have and array with A in index 1, B at index 2 and so on until G which has index 7. We start from index 1 in this example and not index 0. We will now have these rules to help us: *2*i* where i is current index, this will be the current nodes left branch. *2*i+1* will be the current nodes right branch. *i/2* gives you the current nodes parent.

Now if we apply it to the given example above these rules will work.

Add and remove

To add an element works the same as in the tree structure. What I did was to put it in the first empty place in the array i.e in the back. Then after we will compare the element we inserted with the parent using the rules above. We do that until we find it's right position in the array.

Remove also works in the same way as for the tree. Remove the first element, move the last element to the first. Then let it "sink" or descend to it's right place by comparing it with the child node using the rules above.

Benchmarks

Elements n	**Add**	**remove**
100	50	550
200	49	780
400	52	1600
800	52	3200
1600	51	6300

Table 2: Benchmark, linked list number 1

Elements n	**Add**	**remove**
100	600	50
200	800	48
400	1540	49
800	3400	53
1600	6420	52

Table 3: Benchmark, linked list number 2

The question was which implementation would we choose, a linked structure or array? A priority queue can be implemented using an array and a linked data structure, but both has it's pros and cons. Let start with the linked structure implementation. The time complexity is *O(logn)* because you will not look through the whole structure rather you will go left or right depending on what you are searching for. The time complexity is the same for the array implementation. The operations on the linked structure is that one will remain constant while the other will be linear. This is not a big problem if the data is small but if you have a lot of data this implementation will render itself as inefficient. Where adding or removing data will be linear. For the array however can skip indices that is not useful for the operation we are doing. This is why I think that this array implementation is better that the linked version.