Priority Queue and heap

Jonathan Nilsson Cullgert

Fall 2023

Introduction

A priority queue is a normal queue except that the data that is removed is not the data that was added to the queue but the data with the highest priority. The priority could be anything but in this assignment the priority is an integer which all the data has. In this implementation all the data only carries a priority and potentially a size. In a real world implementation the data would of course actually have some sort of that would be added, removed or modified in the priority queue. In this assignment the priority queue is implemented in three different data structures, a linked structure, in a heap and as an array.

Linked structure

The first implementation of the priority queue is to do it in a linked structure. This implementation will have nodes that hold the priority of the node as well as a pointer to the next node in the queue. The linked structure has two different ways to construct the **add()** and **remove()** methods. One of the methods has to have a time complexity of $O(1)$ and one $O(n)$, so either remove is fast or add is fast.

The first implementation of the linked structure is with a fast **add()** method. This method will just add every new node to the front of the queue and not sort it. This way the adding will constant. Since the list is not sorted the **remove()** method will have to look for the node with the highest priority through the entire list and then remove the node with the highest priority. I implemented it by searching through the list twice, once to find the highest priority and then removing the node with that priority.

```
while (test.next != null){
    test = test.next;
    if (test.item < min){
        min = test.item;
    }
```

```
}
if (cur.item == min){
    head = cur.next;
    if(cur.next == null){
        head = null;
    }
    return min;
}

while (cur.item != min ){
    prv = cur;
    cur = cur.next;
}
prv.next = cur.next;
```

This **remove()** method has a time complexity of $O(n)$ since it has to search through the entire list without nestled while-loops. The time for this implementation can be seen in 1 as Linked add(1) and the time for a different amount of operations and the result is clearly quite unstable. But the time complexity it should have is $O(n)$ since the remove method has that time complexity.

The next version is with the fast **remove()** method, in this version the **add()** sorts the list similar to insertion sort which means that it has a time complexity of $O(n)$. Since the list will be sorted the node with the highest priority will be first in the list. Therefore the **remove()** method will just remove the first item in the list with and return its data. The time complexity for this method will be $O(1)$. The remove method will also move the root to the next element in the list which still will be sorted.

```
:
while (item >= cur.item && cur.next != null){
    prev = cur;
    cur = cur.next;
}

if(cur.next == null && item > cur.item){
    Node node = new Node(item, null);
    cur.next = node;
    return;
}
if(cur.next == null && item < cur.item){
    Node node = new Node(item, cur);
    prev.next = node;
    return;
}
```

```
Node node = new Node(item, cur);
prev.next = node;
return;}
```

As we can see in 1 and 2 the linked structure with a fast **add()** is slightly slower than the other implementation of linked structure. The reason for this is because the slow add uses insertion sort which is faster than going through two while-loops. They have the same time complexity but one is preferable to the other.

Tree

The next way to implement the priority queue is to do it in a binary tree structure. Since in previous assignments the binary tree has time complexities of $O(log(n))$ it should be faster then the linked way. If one naively tries to implement the priority queue the same as in previous assignments, high priorities to the left and low priorities to the right one might assume that it will work perfectly. The problem is that if **add()** and **remove()** is used on this tree, nodes will be removed more from the left and added more to the right. This will eventually skew the tree so that it will form a linked structure which we know will have a time complexity of $O(n)$. It will not be more efficient than the other way.

Heap

The way to make this priority queue work in a tree is to change where in the tree high priority nodes will be stored. If the node with the highest priority is stored in the root, we will be able to access it in $O(1)$ time, just find the root. The problem then becomes a question of which node do we replace the root with. If we let the children of the root be sub trees constructed as heaps we know that they will be the next highest priority in the entire tree. We then only have to compare the two children to each other to see which has the highest and promote that one to the root. We can implement this recursively since the subtrees will be heaps and their subtress will be heaps and so on. There might be some problems if a root only has one child our if it is a leaf but these cases will just be handled separately.

With this implementation the removing and adding items to the heap will have a time complexity of $O(log(n))$ which is much better than the linked version. We can also see this in 1 it is much faster than the two linked versions.

The next method to add to the heap is a method to take the root, do something with it, lower its priority and then push it down the tree to the location that it should be in. This method will take in an integer which it will increment the priority with. This method will also return the depth which the root is pushed down the tree. For example if the root is pushed down so it switches with its child and then stays there, the depth it has been pushed is 1.

The reason that we want this push method is because it is very useful, and is often used in system such as operating systems. Taking the root, working with it and then giving it a new priority is most of the time better than removing the root, working with it, giving it a new priority and then adding it back in to the tree. As seen in 3 just pushing items in the tree is much faster than removing and adding, especially if it is done a lot of times in a row.

Another benchmark we want to do on the **push()** method is to see how far down the tree a node is pushed down on average. The benchmark that is made is in a tree with 1023 nodes, every node is given a priority between 0-10000, and everytime a node is pushed its priority is incremented by 10-100. If the push operation is done 1000 times, the average depth a node is pushed is 1 level. The reason the node is incremented by such a small value is because during real world usage the priority of a node is not decreased so much. If the priority is incremented by 10000, on average a node is pushed down 6 nodes out of 10 available levels.

Array implementation

The final implementation of the priority queue is inside an array. This implementation is like taking a tree and placing it inside an array. Every parent has its two children by taking the index of parent $(parent * 2) + 1$ for the left child and $(parent * 2) + 2$ for the right child. The root will be at index 0.

The first thing we want to implement is to make sure that the array is always filled as much as possible. If we do not do this we will have some parts of the array as null when they could be attributed with a priority, which will be a waste of memory. We can do this by creating a bubble method which will start at the end index of the last element of the array plus one. We then know that if the index we are at has its parent at $(n-1)/2$ if it is odd and $(n-2)/2$ if it is even. We then compare their priority and swap them if necessary, we also apply this recursively so that we stop when it is lower priority than their parent or if they have reached the root.

When we remove an element we take the element in the root and store it so that we can return it later. We then store the element in the last position at the root, and let it sink down the array since we probably do not have

a heap structure any longer. In the sink method we compare the current element with its two children at $n * 2 + 1$ and $n * 2 + 2$, we swap with the child with the highest priority. We then call this method recursively so the element sinks down to the position where it is supposed to be.

With these two methods the array will always be as full as it can be and it will be working as a heap, and all the "subtrees" will be heaps as well.

Benchmark

The final part is to compare the different ways of implementing the priority queue, the linked structure, a heap and an array. As we can see in 1 the linked way is incredibly inefficient and should be avoided. The array is also fast than the heap at a higher number of operations but they act about the same with a low number of operations. The array implementation is faster than the heap but unlike the heap, it has to be initialized so that it is can handle the size of the elements it will hold. Another problem with it is that unless it is completely full it will using memory which is not used and is therefore not memory efficient. Overall a one has to consider if speed or allocated memory is what will be prioritized.

n	Linked add(1)	Linked add(n)	Heap	Array
100	900	200	200	100
200	700	600	300	200
400	500	300	300	200
800	1400	800	300	200
1600	5300	4300	800	300

Table 1: Time to add and remove n elements in us.

Implementation	Ratio
Linked add(1)	17.7
Linked add(n)	14.3
Heap	2.7
Array	1

Table 2: Time ratio between the different implementations of the priority queue.

n	push	remove and add
100	200	230
200	180	300
400	300	370
800	300	700
1600	570	1900

Table 3: Table of which time it takes to either push the root or remove and add in us