Priority Queues

Ayda Ghalkhanbaz

Fall 2023

Introduction

In this assignment we will look into the implementation details of an abstract data structure namely priority queues. We will also construct this data structure using different approaches, which in later sections will be discussed about their pros and cons.

Priority Queue

Priority queues, as mentioned earlier, are abstract data structures that operate similarly to queues. However, priority queues come with an added feature, where every item in the queue has a priority. In other words, items with higher priority are served before other elements, and in a case where two or more items have the same priority, their order of insertion will choose which item should be served first. In this task, we will implement a priority queue with two approaches; the first approach includes an *add()* method with a time complexity of $\mathcal{O}(1)$ and a *remove()* method with a time complexity of $\mathcal{O}(n)$, while the other approach is quite the opposite.

Method

In the primary approach, an *add()* method with time complexity of $\mathcal{O}(1)$ and a *remove()* method with time complexity of $\mathcal{O}(n)$ were implemented. The *add()* method was straightforward, since we only add the new node at the beginning of the list and this takes a constant time. On the other hand, the *remove()* method traverses the entire list in order to find a node with the smallest value (highest priority), unlinking the node from the list and adjusting the pointers of the previous and next nodes. This operation obviously takes a time proportional to the size of the list.

The second approach operates diametrically opposite, i.e. adding a new node has a time complexity of $\mathcal{O}(n)$ and removing a node takes a constant time. When adding a new item to the list, the *add()* method traverses the list to find the appropriate place for the item. This operation results in a list

with sorted items which makes it easier for the *remove()* method to work. As the list is already sorted, the *remove()* method locates the item to be removed instantly just by looking at its head node. Thus, returning the first node in the list takes a constant time, therefore it has a time complexity of $\mathcal{O}(1)$.

Result

The two approaches were benchmarked by adding and removing random items while varying the size of the lists. The results are shown in the table1.

size	add O(1)	remove O(n)	add O(n)	remove O(1)
100	25	200	1000	23
200	25	300	2000	19
400	24	500	3000	17
800	23	1000	6000	17
1600	23	2000	1100	5
3200	11	4000	2100	6
6400	6	8500	4400	7

Table 1: Benchmark results of adding and removing random items using different approaches in *ns*

Discussion

The outcomes of the benchmarking, shown in table1, confirm the theoretical time complexities, meaning that the implementations were quite accurate and we've reached the expected results. The selection between the two implementations depends on the preferences. Considering a scenario where we need to add frequently, it is then beneficial to have an *add()* method that can operate in a constant time. Conversely, if frequent removals are needed the preference shifts. Thus, the choice of the implementation depends on the specific requirements.

Heap

Heap is a specialised tree-based data structure that is used to implement priority queues. In a min heap, smaller items have higher priority and are positioned close to the root of the tree, where the root holds the smallest value among all data.

The advantage of using a heap instead of priority queues (implemented in the previous task) lies in the fact that removal and addition operations

have a time complexity of $\mathcal{O}(log(n))$ in their worst-case scenario (due to traversing the entire tree). In contrast, removal/addition operations in the previous task can have a time complexity proportional to the size of the list.

In this task a heap is implemented using two approaches: binary tree and arrays. Subsequently, we will benchmark them and discuss their respective pros and cons.

A Tree Implementation

A heap tree, as mentioned before, keeps the smaller values (highest priority) close to the root, and as we go deeper in the tree, the values increase. In this task, both the *add()* and *remove()* methods utilise recursion, to keep the tree structure correct.

In order to implement the *add()* method, first an *enqueue()* method was implemented. This method sets the item to be added as the root of the tree if the tree is empty, otherwise it calls the *add()* method. The code snippet below demonstrates *add()* method.

```
public void add(int item){
        size++;
        //if the item is smaller than current then swap
        if(item < this.value){
            int temp = this.value;
            this.value = item;
            item = temp;
        }
        //no left children
        if(left == null){
            left = new Node(item);
        }// no right children
        else if(right == null){
            right = new Node(item);
        }// if left side has a smaller size
        else if(left.size <= right.size){
            left.add(item);
        }
        else if(right.size < left.size){
            right.add(item);
        }
    }
```

As we can see in the code snippet above, the *add()* method looks for an appropriate place for the new item, i.e. if it is small enough then it should

be positioned close to the root, otherwise, it constantly seeks for a free and suitable spot.

The *remove()* method follows a similar logic to that of the *add()* method. First, a *dequeue()* method was constructed where it calls the *remove()* method if the tree has more than 1 element and returns the root. The *remove()* is responsible for finding a new root that is the smallest value in the tree, but as it elevates a new root, its entire subtree should be elevated as well. The *remove()* method recursively calls itself till all subtrees have an appropriate root.

A further method named *push()* has been implemented. This method, increments the value of the root and it pushes down the tree to a suitable spot. Similar to previous methods, first a *push()* method was implemented that takes in an integer as parameter, updates the root's value with it and then calls another method named *pushRecursive()*. The implementation of this method was challenging, requiring many considerations of various edge cases. This method utilises multiple if-else statements. The *pushRecursive()* method then returns how deep the pushed element went down in the tree (see page 8 for the implementation of the *pushRecursive()* method).

An Array Implementation

The array implementation of the heap tree was easier since we deal with indexes and not nodes. Traversal, adding and deletion is simpler because any position can be reached using its index. The implementation includes a constructor and several methods; *add()*, *remove()*, *push()* and other helper methods.

The *add()* method takes an argument of type *int* and adds it to the end of the array, and finds a right position for it using a helper method *bubble()*. The *add()* method is also responsible for doubling the size of the array when it becomes full. The *bubble()* method identifies the parent position of the newly added item. If the new item is smaller than its parent then it swaps its place with the parent. The method recursively calls itself until the new item finds its suitable spot.

The code snippet below illustrates the implementation details of the *bubble()* method.

```
public void bubble(int pos){
    if(pos == 0)
        return;
    // finding the parent position
    int parentPos = (pos-1)/2;
    if(prioArr[pos] < prioArr[parentPos])
        swap(prioArr,pos,parentPos);
```

```
        bubble(parentPos);
    }
```

The *remove()* method returns the item positioned at index 0 (the smallest item) and utilises the helper method *sink()* to find the best candidate to be placed at index 0 as the smallest item (highest priority). The *sink()* method identifies left and right children's indexes and employs a series of if-statements to determine the smallest item and locates it as the root. This process is recursively called until every item is in its correct position.

The *push()* method also employs the *sink()* method. It increments the item at position 0 and with the assistance of the *sink()* method, finds a suitable position for it. The *sink()* method implementation details are demonstrated in the code snippet below.

```
    public int sink(int pos, int depth){
        int childL = (2*pos) + 1;
        int childR = (2*pos) + 2;
        if(childR < itemNo){
            if(prioArr[childL] < prioArr[childR]){
                if(prioArr[pos] > prioArr[childL]){
                    swap(prioArr, pos, childL);
                    depth++;
                    return sink(childL, depth);
                }
            }
            else{
                if(prioArr[pos] > prioArr[childR]){
                    swap(prioArr,pos,childR);
                    depth++;
                     return sink(childR, depth);
                }
            }
        }
        return depth;
    }
```

Result

The benchmarking of this task comprised two parts. Firstly, the time for each of the implementations when pushing the root with a random value was measured (figure 1). Secondly, we've measured the time that it takes

for each implementation to remove the root, increment it with a random value and then add it to the heap again (figure 2). The process was conducted on various sizes of heaps filled with random values.

Figure 1. Pushing the root in a tree heap vs an array heap

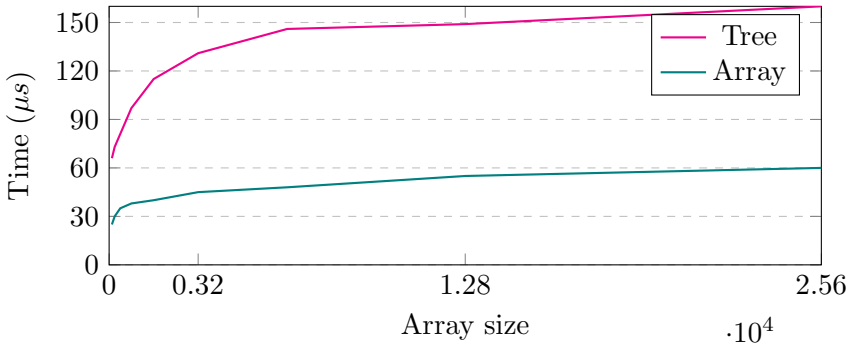
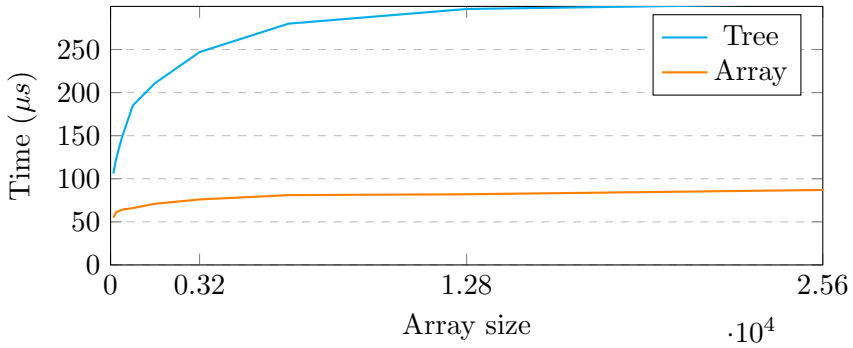


Figure 2. Remove and add the incremented root to a tree heap vs an array heap



Discussion

As evident from the results shown in the figures above, we can observe that both implementations have a time complexity of $\mathcal{O}(log(n))$ which confirms the theoretical time complexity of a heap. However this logarithmic behaviour was more or less predictable, given that a heap inherits tree features, whether implemented as a binary tree with nodes or as a tree within an array. However, we can see that the operations were done in much less time in array implementation.

In fact, multiple factors result in the speed differences between these two approaches. While the tree-based implementation may have certain features making it preferable in specific cases, its drawbacks, such as poor cache locality and requiring more memory to store pointers leads to a slower performance compared to the array implementation.

Arrays, as discussed in previous assignments, are more efficient due to their simpler structure, optimised memory usage, proximity to the CPU

cache and better cache locality, as elements are stored sequentially in the memory.

In conclusion, while the array implementation outperforms the tree-based approach, both implementations are valuable and each may excel in particular cases, making them useful in different contexts.

The *pushRecursive()* method used in the tree implementation of the heap:

```
public int pushRecursive(int pushedNode, Node current, int depth){
    if(current.left == null){
        if(current.right == null || pushedNode < current.right.value)
            return depth;
        // if the pushed is greater than right value, swap!
        else if(pushedNode > current.right.value){
            swap(current, current.right);
            depth++;
        }
        if(current.right.size > 1)
            depth = pushRecursive(pushedNode, current.right, depth);
    }
    else if(current.right == null){
        if(current.left == null || pushedNode < current.left.value){
            return depth;
        }
        else if(pushedNode > current.left.value){
            swap(current, current.left);
            depth++;
        }
        if(current.left.size > 1)
            depth = pushRecursive(pushedNode, current.left, depth);
    }
    // if none of branches is empty, we go to a branch with
    // smaller value (since root should be smallest)
    else if(current.right.value > current.left.value){
        if(pushedNode > current.left.value){
            swap(current, current.left);
            depth++;
        }
        else if(pushedNode < current.left.value)
            return depth;
        if(current.left.size > 1)
            depth = pushRecursive(pushedNode, current.left, depth);
    }
    else{
        if(pushedNode > current.right.value){
            swap(current, current.right);
            depth++;
        }
```

```
        else if(pushedNode < current.right.value)
            return depth;
        if(current.right.size > 1)
            depth = pushRecursive(pushedNode, current.right, depth);
    }
    return depth;
}
```
