A heap or priority queue

Isak Wilkens TIDAB

Fall 2022

Introduction

This is a report on the eighth assignment "A heap or priority queue" on the course "ID2021" TIDAB, KTH. The assignment is to implement different types of priority queues which in some cases is also called a heap. A priority queue does not follow the principle of first in first out that we worked with in the previous assignment. Instead each item in the queue is given a priority which determines the order for which it can leave the queue. Items with a high priority is often at the start of the queue. The priority queues implemented will all be of the type whereas the items with the lowest values have the highest priority. We will in this assignment look at four different implementations of a priority queues and discuss these:

1. Single linked list where the add method has a time complexity of O(1) and the remove function has a time complexity of O(N).
2. Single linked list where the add method has a time complexity of O(N) and the remove function has a time complexity of O(1).
3. A priority queue implemented with the a binary tree structure
4. An implementation of a priority queue using an array.

Single linked list 1

The first priority queue was implemented with the code structure from the previous assignments with linked lists. The change this time was that the nodes were given an extra property to hold on to called priority. This value determined when they should leave the queue. For this first implementation the add method simply adds a node to the start of the list. To remove an item we however need to go through the whole list to see which item has the lowest priority. Therefore giving the add function a time complexity of O(1) and the remove function a time complexity of O(N) every time. The code for this new remove function is given below, it is implemented using

pointers to the smallest element and keeping track of its previous element. This way we can remove the smallest element from the list and return it.

```
public Node remove(){
     Node next = head;
     Node smallest = head;
     Node previous = null;

     while(next.tail != null){
         if (next.tail.priority < smallest.priority){
             smallest = next.tail;
             previous = next;
         }
         next = next.tail;
     }

     if(smallest != head){
         previous.tail = smallest.tail;
     }
     else{
         head = head.tail;
     }

     return smallest;
 }
```

The results for bench marking add and remove operations on lists of differing sizes is given below. Each value is the mean of ten thousand runs for each list size.

Repetitions(n)	**10**	**100**	**1000**	**10000**	**100000**
Run-time(ns)	$3,5 * 10^2$	$7,1 * 10^2$	$4,8 * 10^3$	$4,3 * 10^4$	$4,2 * 10^5$

Table 1: Mesurements of the run-time as a function of linked list size for the priority queue. Run-time in nano seconds.

Single linked list 2

This time the priority queue will be implemented with a remove function of O(1) and an add function of O(N). This time the values in the list will have to be sorted since we do not we can not move back in the list so we will have to have the values sorted to find the correct position. To find the correct position for our new value we simply iterate through the linked list and keep

moving while the next element is smaller than the one we are trying to add. The implementation for this add function is given below:

```
public Node add(int item, int priority){
    LinkedPriority.Node next = head;
    Node new_node = new Node(item, priority, null);
    if (head == null){
        head = new_node;
    }
    else if (head.priority > priority){
        new_node.tail = head;
        head = new_node;
    }
    else{
        while(next.tail != null && next.tail.priority < priority){
            next = next.tail;
        }
        new_node.tail = next.tail;
        next.tail = new_node;
    }
    return head;
}
```

Table 2 shows the bench marking of this new and updated version of our linked list implementation of a priority queue.

Repetitions(n)	**10**	**100**	**1000**	**10000**	**100000**
Run-time(ns)	$1,6 * 10^2$	$3,1 * 10^2$	$3,2 * 10^3$	$3,6 * 10^4$	$2,2 * 10^5$

Table 2: Mesurements of the run-time as a function of linked list size for the priority queue. Run-time in nano seconds.

Tree implementation of a priotiry queuel

To improve upon our priority queue we have seen from the previous algorithms using a binary tree structure operations can be done in O(log(n)) run time complexity. The binary tree from the previous assignment is being used now to implement a priority queue or heap. The root of the tree will always have the item with the highest priority in the queue. Each branch of the tree will then work as separate heap structures. As done before with the linked lists we add information to the nodes to give them a priority value, but here they also gain a size. The size of a node tells the program how many nodes that are under the current node. For example if the size of a

node is 4 it has three nodes branched down under it. We will be able to take advantage of this feature when adding elements to the queue. The code for the add operation is given below with comments for the different parts that will be further explained.

```
// 1
private void add(Integer priority, Integer value){
    if (this.priority == priority){
        this.value = value;
        return;
    }
    //2
    if (priority < this.priority){
        int tempValue = this.value;
        int tempPrio = this.priority;
        this.value = value;
        this.priority = priority;
        value = tempValue;
        priority = tempPrio;
    }

    //3
    this.size++;

    // 4
    if (this.left == null){
        this.left = new Node(priority, value, 1);
    }
    else if (this.right == null){
        this.right = new Node(priority, value, 1);
    }
    else if (this.left.size < this.right.size){
        this.left.add(priority, value);
    }
    else{
        this.right.add(priority, value);
    }
}
```

Code section (1) in the add function checks if the node that is being added has the same priority, in this case we just put the new value into the root. (2) If an item in the tree has a higher priority than the one that we just added it needs to go further down and we swap their data. (3) When a node moves down the tree, its size needs to be increased. (4) If the branches

are leaves we add a new node, if not we recursively call this function to traverse further down the tree.

We now need to be able to remove values, the code for this is given below:

```
public Node remove(){
    //1
    if (this.left == null){
            this.priority = this.right.priority;
            this.value = this.right.value;
            this.right = null;
            this.size--;
            return this;
        }
    if(this.right ==null){
        this.priority = this.left.priority;
        this.value = this.left.value;
        this.left = null;
        this.size--;
        return this;
    }

    //2
    if (this.right.priority < this.left.priority){
        this.priority = this.right.priority;
        this.value = this.right.value;
        this.size--;
        if ( this.right.size == 1){
            this.right = null;
        }
        else{
            this.right = this.right.remove();
        }
        return this;
    }
    else{
        this.priority = this.left.priority;
        this.value = this.left.value;
        this.size--;
        if (this.left.size == 1){
            this.left = null;
        }
        else{
            this.left = this.left.remove();
```

```
            }
            return this;
        }
    }
```

In the first part of the remove function (1), we check if a branch is null we can then simply move up the data from the other branch and then set the branch to null. The value of the node is returned. (2) is the case when the node has two branches we compare the sizes to see if we should remove the left or the right one. In the case of the following node only having size 1 we can use the same method as abode and set the pointer to null afterwards. If the size is greater than one we will instead recursively call the remove function again to traverse further down the tree.

Both of these functions are then implemented from outside the nested Node class the code for this is given below, here we handle the special cases of the root being null.

```
public void add(Integer priority, Integer value){
        if (root == null){
            root = new Node(priority, value, 1);
        }
        else{
            root.add(priority, value);
        }
    }

public void remove() throws NullPointerException{
    if(root == null){
        throw new NullPointerException("Heap is empty");
    }
    else if (root.right == null && root.left == null){
        root = null;
    }
    else{
        root.remove();
    }
}
```

To add further functionality to the priority queue a push method is added. This method is for when we want to take the item waiting at the root of the tree, and put it back into the priority queue with a new priority. To see how far down the tree it ends up a variable for depth is added that keeps track of how far down the tree the item is pushed. This is dependent on the increased value of priority we give to the re-queued item. The code for the push operation is given below:

```
public Integer push(Integer incr){
        this.root.priority += incr;
        Node curr = this.root;
        return push(this.root.priority, curr, (Integer)0);
    }

private Integer push(Integer rootValue, Node curr, Integer depth){
    // 1
    if (curr.left == null){
        if (rootValue < curr.right.priority){
            return depth;
        }
        //Om noden till höger har högre prio ska den upp så vi swappar
        //Och ökar djupet
        else if(curr.right.priority < rootValue){
            depth++;
            swap(curr, curr.right);
        }
        // 2
        if (curr.right.size != 1){
            depth = push(rootValue, curr.right, depth);
        }
    }
    if (curr.right == null){
        if (rootValue < curr.left.priority){
            return depth;
        }
        else if(curr.left.priority < rootValue){
            depth++;
            swap(curr, curr.left);
        }
        if(curr.left.size != 1){
            depth = push(rootValue, curr.left, depth);
        }
    }

    // 3
    if (curr.right != null && curr.left.priority > curr.right.priority){
        if(curr.right.priority > rootValue){
            return depth;
        }
        else if(curr.right.priority < rootValue){
            depth++;
            swap(curr, curr.right);
```

```
            }
            if (curr.right.size != 1){
                depth = push(rootValue, curr.right, depth);
            }
        }
        else{
            if (curr.left.priority > rootValue){
                return depth;
            }
            else if(curr.left.priority < rootValue){
                depth++;
                swap(curr, curr.left);
            }
            if (curr.left.size != 1){
                depth = push(rootValue, curr.left, depth);
            }
        }
        return depth;
    }
```

The first part of the code for the push operation (1) checks if the given priority to the root element is still less then what is under it, it then stays there. (2) if the node to the right has a higher priority we swap the data in the nodes and increase the depth. (3) If there is more then one node under where we are now, meaning that the size is not 1 we recursively call the push operation to traverse further down the tree. The next part of the code (4) one of the branches are not null and we need to continue down the tree. In the same way as we did in the previous operations we compare the priority of the branches and see if we need to traverse down further.

To bench mark this firstly a 64 random elements are added with random priorities from 0 to 100 and values to the heap, then a push operation is done with random increments from 0 to 20 . This is repeated a thousand times. This gave an average of 2,01 in depth where the largest that was found was 4 and the smallest 0. When instead done with add the values were twice as high. This results follow what should come from the written code where the push operation should be O(log(n)) and add O(n).

Priority queue with an array

To create a priority queue with an array we need to keep track of which item in the array is a parent node and the branches below that will be called children. By keeping track of where the parent node is and the two children we can make us of an array as a heap. However if it is to work as a heap or priority queue we need to re-balance the array when adding

and removing so that the element at the start of the queue has the highest priority.

```
//1
public boolean isNull(int curr){
    if(curr > (size /2)){
        return true;
    }
    else{
        return false;
    }
}
//2
 public void balance(int curr){
    if(isNull(curr) == false){
        int temp = curr;
        if(rightChild(curr) <= size){
            if (heap[leftChild(curr)] < heap[rightChild(curr)]){
                temp = leftChild(curr);
            }
            else if (heap[leftChild(curr)] > heap[rightChild(curr)]){
                temp = rightChild(curr);
            }
            else{
                temp = heap[leftChild(curr)];
            }
        }
        if (heap[curr] > heap[leftChild(curr)] || heap[curr] < heap[rightChild(c
            swap(curr, temp);
            balance(temp);
        }
    }
}
//3
    public int remove(){
    int toRemove = heap[first];
    heap[first] = heap[size--];
    balance(first);
    return toRemove;
}
//4
    public void add(int newItem){
    if (size > maximal_size){
        throw new IndexOutOfBoundsException();
```

```
        }

        heap[++size] = newItem;
        int curr = size;

        while (heap[curr] < heap[parent(curr)]){
            swap(curr, parent(curr));
            curr = parent(curr);
        }
    }
```

The first method (1) checks whether we are on a leaf or not in the queue. (2) Is a balancing method for when we have removed an item. We then need to take the last item in the queue which will be the largest and let it sink through the array sorting all the other items to their proper place with regards to this large item. (3) Shows the remove method that takes the first item in the queue and then calls to the balancing method. (4) Is the add method where we add a node to the end of the queue and then let it "bubble" up to the start sorting the remaining elements with regards to this one.

When bench marking the priority queue as an array it was on mean faster than the linked list which confirms it works as intended similarly to a balanced tree structure with O(log(n)) for both add and remove.

Conclusion

The linked list implementations had their pros and cons. One can be used on an unsorted array, the one with remove and O(n). And the faster one with add as O(n), we then however need to sort the list first. But as learning in the sorting assignment this is often worth it. The tree implementation was very hard to implement and took a long time. The array felt far superior in runtime and implementation.