Heaps

Aditya Gupta

October 2022

1 Introduction

In this assignment, we analyse a priority queue which is essentially a queue where the priority for which item leaves the queue first is determined by some separate factor, such as the value of the item in our case. We implement a min-heap or min-priority queue where the priority is such that the lowest elements leave the queue first. We implement this construct by making use of 3 different kinds of data structures that is a Linked List, a binary tree and finally using an array.

2 Heaps using Linked Lists

In this implementation of the priority queue, we make two different implementations. In the first case, we get a time complexity of O(1) for add and O(n) for remove and vice versa in the second case.

2.1 Case 1:

In this case, we make the add() function such that all the elements in the list are added in a sorted manner, to begin with. This means that the remove function takes constant time as we always remove the head element as it's supposed to be the element with the highest priority and thus it would be the first element to be removed. But this means that we have to search for the right position to place the element in the queue and thus it takes linear time to add elements to the priority queue.

```
public void addPriorityLinear(int key){
    if(head == null){
        head = new Node(key);
    }
    else if (head.data > key) {
        Node temp = new Node(key);
        temp.next = head;
        head = temp;
```

```
    }
    else{
        Node current = head;
        while(current.next!=null && current.next.data<key){
            current = current.next;
        }
        Node nxt = current.next;
        current.next = new Node(key);
        current.next.next = nxt;
    }
}
```

2.2 Case 2:

In this case, we make the add() function such that all the elements are added to the end of the list and this means that we can add items in constant time (as we can keep track of the tail element). But this also means that removing elements from the list is going to take linear time as we have to search for the elements with the lowest value and thus the highest priority.

```
public void addDataConstant(int data){
    if(head == null){
        head = new Node(data);
        tail = head;
        return;
    }
    else{
        if(tail != null){
            tail.next = new Node(data);
            return;
        }
    }
}
```

3 Benchmarks:

Here are the results from benchmarking the above implementations of the linked-list-based heap data structure.

3.1 Case 1:

We clearly see that for this case we have linear time (O(n) time complexity) growth when adding elements while a constant time(O(1) time complexity) is taken to remove elements as expected from how we implemented the data structure for the first case.

Case	n	Time(Add)	Time(Remove)
1	250	65	58
2	500	143	59
3	1000	290	63
4	2000	562	59
5	4000	1184	48
6	8000	2038	51
7	16000	4117	56

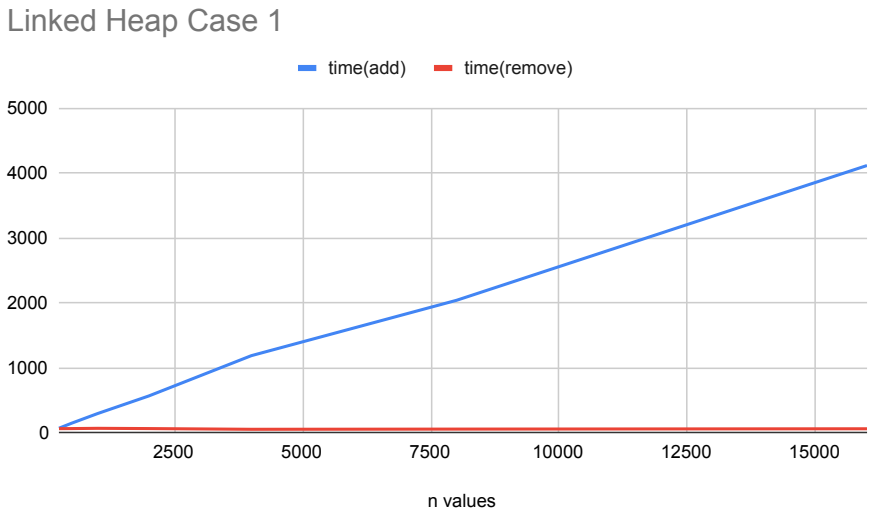

Figure 1: Graph for Case 1 of Linked Heaps

3.2 Case 2:

We clearly see that for this case the time complexity of the add function is O(1) or constant time while for the remove function it is given by O(n) or linear time. This is because we always add to the tail node or the last node in the list which we keep track of thus we can add nodes to it in constant time. While when we want to remove elements we have to search for the element with the highest priority in the list.

Case	n	Time(Add)	Time(Remove)
1	250	59	50
2	500	53	89
3	1000	51	163
4	2000	53	345
5	4000	53	712
6	8000	47	1507
7	16000	52	3001

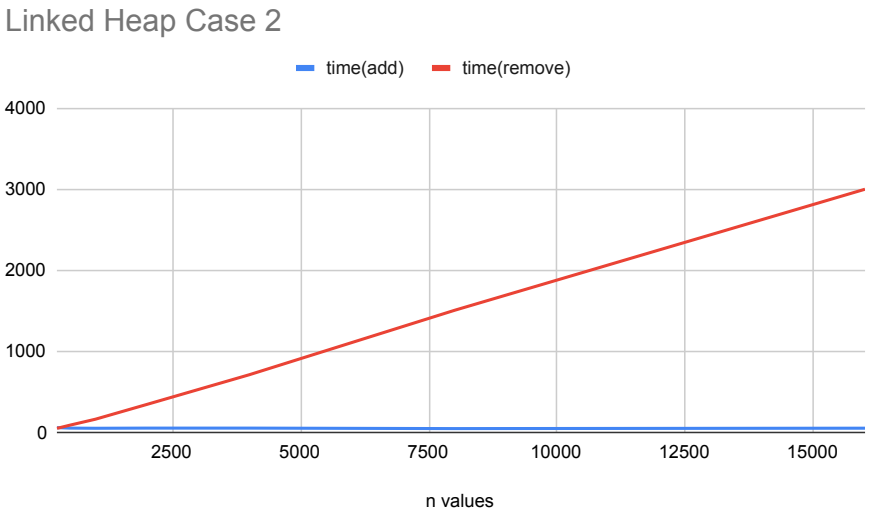

Figure 2: Graph for Case 2 of Linked Heaps

4 Heaps using a Binary Tree

We use a binary tree similar to the one we used in our previous assignment but we just have to adjust it so that it works like a heap. In a binary tree, the lowest element is the leftmost element but in a heap, the root element has to be the lowest element (or the element with the highest priority in a general scenario).

4.1 Adding an element to a Heap:

We first check if the root of the heap tree is null, if it is then we set the root node of the heap's internal tree to the value passed by the user and we increment the size of the Heap. Otherwise, We add the data using the helper function addDataToNode() taking the root node as the current node.

```
public static Integer add(Integer key){
    if(root == null){
        root = new TreeNode(key);
        size++;
        return 0;
    }
    else{
        return addDataToNode(root,key);
    }
}

public static Integer addDataToNode(TreeNode current, Integer data) {
    current.size++;
    int depth = 1;
    if(data < current.data){
        Integer temp = data;
        data = current.data;
        current.data = temp;
    }
    if (current.left == null) {
        current.left = new Heaps.TreeNode(data);
        Heaps.size++;
    } else if (current.right == null) {
        current.right = new Heaps.TreeNode(data);
        Heaps.size++;
    } else{
        if (current.left.size <= current.right.size){
            depth += addDataToNode(current.left,data);
        }
        else {
            depth += addDataToNode(current.right,data);
        }
    }
    return depth;
}
```

For every node inside the heap tree, the node has a data value and size. The size of a node indicates the number of nodes below it. Whenever the helper function is called we increment the size of the current node. This is followed by checking if the data passed by the user is smaller than the data of the current node, if it is then we swap the data passed by the user with the data of the current node and move further into the heap.

This way the element with the highest priority is always at the top of the heap. After that, we check for the nullity of its child nodes, and if either of them is null then these nodes are set to the data passed by the user and we increment

the size of the current node. If both the left and the right nodes are not null then we compare their sizes. This means that to keep the tree balanced we add new nodes on the branch with the smaller size.

4.2 Removing an element from a Heap:

Since this is a priority queue we remove the element with the highest priority first. This means that the root of the heap tree is the element that gets removed every time we call this function. But this means we need to readjust some nodes so that it tree continues to function like a priority queue.

```
public void removeRoot(TreeNode node){
    node.size--;
    if(node.left == null){
        swapNode(node,node.right);
        if(node.right.isLeaf()){
            node.right = null;
            return;
        }
        removeRoot(node.right);
    }
    else if(node.right == null){
        swapNode(node,node.left);
        if(node.left.isLeaf()){
            node.left = null;
            return;
        }
        removeRoot(node.left);
    }
    else if (node.left.data <= node.right.data)
    {
        swapNode(node, node.left);
        if (node.left.isLeaf())
        {
            node.left = null;
            return;
        }
        removeRoot(node.left);
    }
    else if (node.right.data < node.left.data)
    {
        swapNode(node, node.right);
        if (node.right.isLeaf())
        {
            node.right = null;
            return;
```

```
        }
        removeRoot(node.right);
    }
}
```

We first decrement the size of the tree. After this, we need to promote one of the child nodes as the new root node based on their values or priority. But before that, we check if either of the child nodes is null so that we can directly set the other child node as the root. If the left node is null then we set the right node as the root. To do this we swap the values of the current node and the node to its right. We follow this up by checking if the right node is by chance a leaf by calling the isLeaf() function which just checks if both left and the right child nodes are null for a node.

If the right node is a leaf then we set it to null and return, otherwise, we recursively call the remove function to adjust the heap. We do the same process in case the right node was null instead of the left one and proceed in the same way as before. If both the left and the right nodes are not null then we compare their values and the one with the smaller value is chosen as the root node. After that, we adjust the other nodes in the tree in the same way as mentioned before.

4.3 Increment Root:

We call the incrementData() function to increment the value of the root. After this, we push the root down into the heap and readjust all the nodes inside of the heap accordingly if the root value after incrementing is larger than at least one of its child nodes. We check for the nullity of the child nodes and if either of them is null we only need to worry about the non-null node.

We check if the data of the non-null node is less than or equivalent to the node passed as an argument, swap the nodes and call the pushDown() function again but recursively. If neither of the child nodes of the argument node is null then we compare their data values and the one with the smaller value is chosen as the node that will replace the root. After this, we follow the same procedure as before where we swap our nodes and then recursively call the pushdown function.

```
public Integer pushDown(TreeNode node){
    if(node.isLeaf()) return 0;
    if(node.right == null){
        if(node.left.data <= node.data){
            swapNode(node,node.left);
            return 1+pushDown(node.left);
        }
    }
    else if (node.left == null) {
        if(node.right.data < node.data){
```

```
            swapNode(node,node.right);
            return 1+pushDown(node.right);
        }
    }
    else{
        if(node.left.data <= node.right.data){
            if(node.left.data < node.data){
                swapNode(node.left,node);
                return 1+pushDown(node.left);
            }
        }
        else{
            if(node.right.data < node.data){
                swapNode(node,node.right);
                return 1+pushDown(node.right);
            }
        }
    }
    return 0;
}
```

4.4 Depth Benchmark:

Here we benchmark the average depth that our add and increment operations have to go down into the heap. For the benchmark, we add 64 random numbers in the range of 1-100 into the heap tree. After this, we test the depth of the add and the increment operations. For this benchmark, I got a value of 6 as the average depth that the add operation has to traverse, while for the case of the push function I got an average depth of 1.78 as the average depth that the push function needs to go down the heap.

These results were obtained after doing 1000 iterations of the above procedure where we generated a random tree for each iteration. The function for the addDataToNode() and pushDown() was changed slightly so that we can retrieve the depth values for this benchmark. The code for this is available in the previous section where we discussed the implementation of the add and increment functionality.

5 Heaps using an Array

An array lends itself very well to making the heap as it makes it a lot easier to implement. As explained in the assignment document, the data in the array is placed in such a way that the left child nodes are given by the formula of $2n+1$ and all the right children are given by $2n+2$ for a parent node at index n. This holds true for all trees where the root node is placed at index 0. I implement a

slightly modified version of the same thing where I have it such that the root is placed at the index = 1, which means that the position of the left child node is given by $2n$ and for the right node the position is $2n + 1$, thus we can always find the parent node by the formula $position/2$.

5.1 Adding elements into the Array Heap:

The size variable is used to keep track of the number of elements that exist in the heap and as a pointer variable. If the value for the size variable exceeds the maximum capacity of the internal array, we just simply return as we cannot add any more elements into the heap.

Otherwise, we pre-increment the size value and use it as the index at which we add the new key into the heap array. After this, in case the value of the newly added node is lower than the value of its parent node, then we swap their values so that our array qualifies as a heap.

```
public void add(Integer key){
    if(size >= capacity){
        return;
    }
    heap[++size] = key;
    int current = size;
    while (heap[current] < heap[parent(current)]){
        swap(current, parent(current));
        current = parent(current);
    }
}
```

5.2 Removing elements from the Array Heap:

As this is a min priority queue, we always remove the smallest element first which in this case is the root. The root's position is stored as a constant called FRONT. We store the value of the root in a temporary variable and set the new root to whatever last value was added into the heap. After this, we call upon the minHeapify() function at the FRONT position to adjust all the values according to the heap's hierarchy. After which we just return the old root value.

```
public Integer remove(){
    int popped = heap[FRONT];
    heap[FRONT] = heap[size--];
    minHeapify(FRONT);
    return popped;
}
private void minHeapify(int position){
    if(!isLeaf(position)){
        int swapPosition = position;
```

```
        if(rightChild(position)<size){
            swapPosition =
                    (heap[leftChild(position)] < heap[rightChild(position)])?
                    leftChild(position):rightChild(position);
        }
        else{
            swapPosition = heap[leftChild(position)];
        }
        if (heap[position] > heap[leftChild(position)] ||
            heap[position] > heap[rightChild(position)]){
            swap(position,swapPosition);
            minHeapify(swapPosition);
        }
    }
}
```

The minHeapify method first checks whether the position that has been given to it as an argument is a leaf or not. The minHeapify() subroutine's purpose is to relocate a larger value to a deeper position inside of the heap where it fits in accordance with the heap's hierarchy. But if the position we have is already a leaf then it cannot be "heapified" further.

Otherwise, we create a swapPosition variable which is used to store the value of the position of the child node that is smaller than the current position. First, we check which immediate child value is the smallest and then we store that position as the value of the swapPosition variable. After this, we swap the values at the current position with the value at the position swapPosition and call this function recursively for that position so that all the values in the tree are adjusted correctly.

5.3 Increment Operation in Heap Array:

This operation is not very different from the one we implemented for the Tree heap. The implementation would just involve incrementing the value at the FRONT followed by calling the minHeapify() function to readjust the location of all the values.

6 Benchmarks: Heap Arrays vs Linked Heaps

Here we benchmark the time taken by the add and remove operations for our heap array and the linked list heap implementation. We do this by adding a random value to the heap and removing the priority element from it right after that. This is done over 1000 iterations and we take the average time taken to this task for the two implementations.

Since we did two different implementations of the linked list heap for this assignment I have included the benchmark for both of them in this. All the time values are given in milliseconds. Here AH means time for the Array Heap, LH1 means time taken for the first case of the linked list implementation and LH2 is time for the 2nd case of the linked list implementation.

Case	n	Time AH	Time LH1	Time LH2
1	250	9.83	4.61	4.35
2	500	10.09	9.34	9.41
3	1000	12.05	16.54	18.92
4	2000	18.50	35.02	34.77
5	4000	30.80	68.43	72.24
6	8000	43.65	140.05	145.73
7	16000	92.93	297.46	306.71
8	32000	120.85	608.80	634.09

The time complexity of the add operations and remove operations in the case of the array heap and the tree heap are given by $O(log(n))$. In the case of the Tree, this is because whenever we add or remove something we go down either the left or right path each time thus reducing the number of paths that can be taken by half each time as we recurse further down the heap.

In the case of the array heap, we recurse at the position of one of the child nodes based on their value and thus each time we choose one of the 2 child nodes every time we recurse and thus we have a logarithmic time complexity for this kind of heap as well. While in the case of the Linked Lists they are either $O(1)$ and $O(n)$ or vice versa based on the implementation.

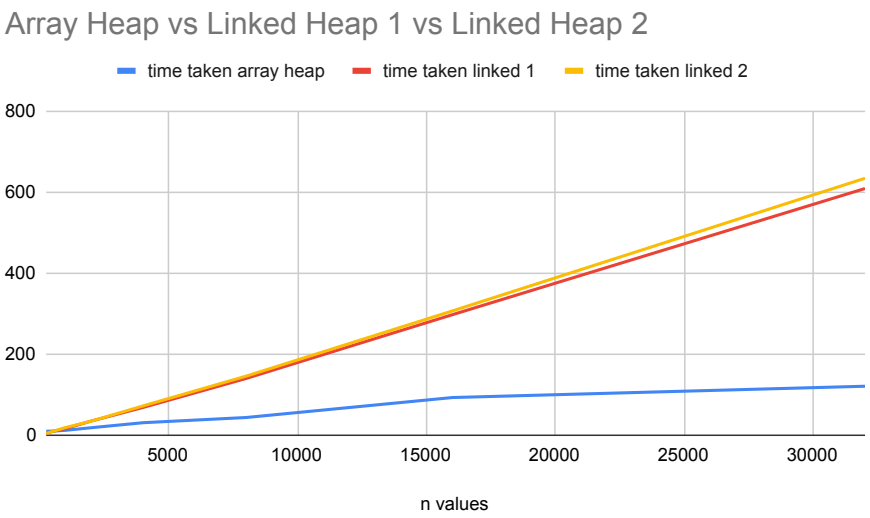

Figure 3: Graph for the above benchmark