Report

ADS2 AE2

Umar Yusuf
2613065y
18 March 2022

Documentation

To run the code for any of the queues, create an object of the type of the queue you want to create from the *Queue* package. E.g;

```
MinPriorityQueueHeap queue = new MinPriorityQueueHeap();
                                                       Or
MinPriorityQueueBST queue = new MinPriorityQueueBST();
                                                       Or
MinPriorityQueueEfficientBST queue = new MinPriorityQueueEfficientBST();
```

You can then insert, get the smallest value and extract-min as follows;

```
queue.insert(5);
int min = queue.min();
int extractedMin = queue.extractMin();
```

For the MinPriorityQueueHeap, you have the option to set an initial capacity by passing in an int at initialisation as follows;

```
MinPriorityQueueHeap queue = new MinPriorityQueueHeap(25);
```

For the ropes problem, you can run it by calling the Ropes.optimalConnect() method and passing in an array of integers representing the ropes as follows;

```
Ropes.optimalConnect(new int[] {4, 8, 3, 1, 6, 9, 12, 7, 2});
```

I have provided examples of the usage of each as methods in the *Main* class. These can be run to see example output using the Main.main method.

Part 1

A) For this I decided to use a min-heap as the base. I made a few changes though. Mainly in how the heap is initialised and built. In the regular minHeap you would have to pass in an array which is then used to build the heap, I had to change this because I was insert nodes one after the other with the insert function not as a complete array. I did this by creating an empty int array at initialisation of the queue and an integer *size* that would keep track of how many nodes were in the array so I could have a sort of partition between the part of the array that has actual nodes and the part that was full of zeros (default values of elements in java int array). This presented another problem, what if a user inserts more elements than the heap array can contain. I fixed this by checking if the number of nodes (size) is equal to the capacity of the heap array at each insert and doubling the capacity of the heap array if so.

This then allows the insert operation to just set the element at index size to the new element and increment size to add a new node. This would run in O(1), however to ensure that the min-heap property is maintained the heap must be rebuilt after each insert. I do this using the buildHeap method which is a modified version of the BULD-MAX-HEAP from lecture 8, which heapifies which uses the smallest element at heapify to build a Min-Heap instead, and uses size as n instead of array.length so that the default zeros are ignored. This buildHeap function runs in O(n) complexity so the insert function actually has a time complexity of O(n).

Moving onto the min operation, in a min heap the smallest node is always at the top of the heap. This makes it possible to have a min operation that just returns the element at index 0 of the heap array and has a time complexity of O(1).

The last operation, extract-min first checks if size is equal to zero and throws an IndexOutOfBounds exception. This is done because the extraction logic includes decrementing size and can lead to an array underflow if the size is zero. It then sets a variable min to the element at the start of the heap array to get the smallest node in the heap as explained in the min operation. Initially I wanted to use a loop to move each of the elements left one spot in the array, this would have had a complexity of O(n) but I figured out a more efficient way of just switching the first node with the last node and then decrementing size so the first node, that is now in the position of the last node, is not counted as part of the heap going on this allows it to extract from the heap in O(1). It then re-heapifies with the heapify method that runs in O(log n) to ensure that the min-heap property is upheld. This means that the extract-min operation also has a time complexity of O(log n).

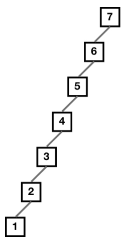
B) This ADT contains a private Node class within it that is used to create nodes that can be added to the BST. Each node contains pointers to its parent node, and the child nodes on its left and right as well as an integer called key that represents its value. I did it like this so that I can just edit the private Node class in the more efficient version of this ADT *(see 1d)* instead of creating a different Node file for each of them which would have required the nodes for each to have different class names. I also have a size variable that tracks the number of nodes in the tree by incrementing at every insert and decrementing at every extract. This allows me to throw an IndexOutOfBound exception if the user tries to get or extract the min node of an empty tree.

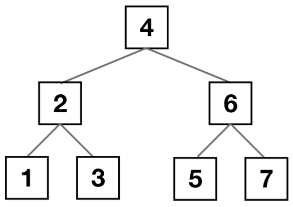
Starting with the insert operation, to add a new node (lets call this new node x) to a tree you would have to set it as either the left or right of a node that is already in the tree (this would be the new nodes parent element). This is a relatively easy operation and has a time complexity of O(1). But which element should parent the new node, and would the new node be to the left or the right of its parent? This is where the binary search tree comes into play, the binary-search-tree property tells us that the left child of any node x must less than or equal to x, and the right child of x must be greater than or equal to x. This principle is used all across my implementation of the ADT. When inserting a node, I first check if the root of the tree is null, if so I set the root to the new node. Otherwise, I check if the element belongs to the left or the right of the root element by comparing the key of the root to the value of the new node. I move down the tree recursively, repeating this process until I find a node (lets call this x) whose child position (left or right) which the new node fits into (determined by the binary-search-tree property) is null. I then create the new node and add it to the BST as either the left or right child of x as determined by the binary-search-tree property. Because we are moving down the tree and not checking all of the elements the average time complexity is O(h) where h is the height of the BST.

Moving on to the Min operation, due to the binary-search-tree property in a valid BST the smallest node of a BST would always be the leftmost leaf (node where left and right are null) of the tree. This makes finding the smallest node as easy as starting at the root and going down to the left until a node is found that doesn't have a left child and returns the key of that node. The average time complexity for this would be O(h) as again we are going down the tree and not checking all elements.

Then lastly the extract-min operation, this is similar to the min operation in that it goes down from the root of the BST leftwards until it finds the smallest (leftmost) node. It then stores this node in a variable, min. Now at first I wanted to extract by just setting the left pointer of the min node's parent to null. However this presented a problem when the min node had a right child because it would have cut off the entire left subtree of the parent not just the one node. I overcame this by transplanting the min node with its right child and then returning the key of the min node. The average time complexity of this operation is O(h) as well, since all the extra logic added, including the transplant method to that of the min operation has a complexity of O(1).

C) I have so far demonstrated the average time complexity of each of the BST operations. But to understand the worst case we must understand that a valid BST containing a given set of nodes can take multiple forms that have different heights. For example a BST with nodes 1,2,3,4,5,6,7 can be expressed validly as both;



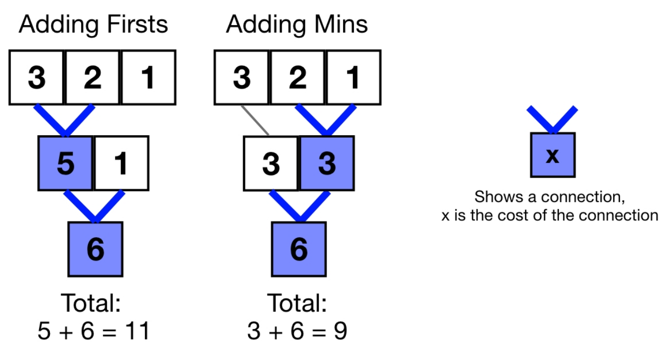


The first is severely unbalanced and all the nodes are in a straight line so the height(h) is equal to the number of nodes (n). Since all operations run O(h), this would mean that they also run in O(n). The second however is balanced and each time you traverse down you half the number of nodes left to traverse. So the height is O($\log_2$ n). A self balancing tree ensures that the tree remains balanced (almost) so the worst case running time is O($\log_2$ n).

D) My solution to the problem at hand hand requires us to look back at the first BST queue implementation. In that implementation the min and extract-min only ran in O(h) because that is what is required to find the smallest node in the tree. But what if the smallest node was calculated at insert and stored in a variable? This would allow me to just access the variable at min and extract-min allowing them to run in O(1). This however had its own challenges. After the smallest node is extracted, the variable containing the smallest node would then be have a node that doesn't exist in the tree. We could then calculate the successor of the smallest node and set the smallest node to it, but then extract-min would no longer be O(1). The solution to this was to add an extra pointer in each node that points to its successor and calculate that pointer at the time the node is created instead of at extract. This solves our problem but also presents another problem. Given a node x, at every insert, there is the possibility that a node is added that is bigger than x but smaller than the current successor stored in x. This new node should then be the successor of x. The successor of x would always either be its parent or the minimum its right subtree because the node to the left is smaller than it. Because, our new node is inserted after x it would not be x's parent so it must be in its right subtree. So the new nodes predecessor would be one of its ancestors (nodes higher than it in the BST). This allowed me to solve the problem by recalculating the successor of each of the new node's ancestors at each insert. Just like that, min and extract-min can run in O(1).

Part 2

A) I noticed that the total cost of connections is the sum of all the connection costs. So there are two ways of reducing the total cost. Either reduce the number of connections or reduce the cost of each connection. It would not be possible to reduce the number of connections because we are only told the how to calculate the cost of joining two ropes, not three or four and so on. Therefore we must reduce the cost of each connection. Given that the cost of connecting two ropes is equal to their sum, by always connecting the two smallest ropes we can minimise connection costs as shown below:



To do this I create a min-priority queue (I used the heap queue class from 1a) and insert the rope length that are given. This is important because it allows me to easily and efficiently extract the shortest length since the extract min operation of the heap min queue is the most used and it allows me to get the smallest ropes efficiently because it only makes two operations which are the array indexing and return. If I were to use just a normal array, I would have to loop over the whole array every time I want to get the smallest rope length and even after that there would be no simple way of extracting it.

I start off by creating a variable total cost that starts as zero to track the cost. I then use the extract-min operation to get the shortest length and then use extract-min again to get the second shortest and store them both in variables. Remembering that the cost of connecting two ropes is equal to their sum, I add the cost of connecting them to the total cost and insert the length of the new rope made from the connection (which is also the sum of the two ropes that were connected). I repeat this process until there is just one length left in the queue. At that point all the ropes have been connected and the length left in the queue is the length of the final rope. The total cost is now the optimal connection cost.

```
B) The initial queue is: PriorityQueue:9 [ 1 2 4 3 6 9 12 8 7 ]
Connected 1 and 2 for a cost of 3
The new queue is: PriorityQueue:8 [ 3 3 4 6 8 9 12 7 ]
Connected 3 and 3 for a cost of 6
The new queue is: PriorityQueue:7 [ 4 6 6 7 8 12 9 ]
Connected 4 and 6 for a cost of 10
The new queue is: PriorityQueue:6 [ 6 7 10 9 8 12 ]
Connected 6 and 7 for a cost of 13
The new queue is: PriorityQueue:5 [ 8 9 10 12 13 ]
Connected 8 and 9 for a cost of 17
The new queue is: PriorityQueue:4 [ 10 12 13 17 ]
Connected 10 and 12 for a cost of 22
The new queue is: PriorityQueue:3 [ 13 17 22 ]
Connected 13 and 17 for a cost of 30
The new queue is: PriorityQueue:2 [ 22 30 ]
Connected 22 and 30 for a cost of 52
The new queue is: PriorityQueue:1 [ 52 ]
The final rope length is: 52
The total cost is: 153
```