**1. Understanding Problem:**

You are required to complete the code of our Priority Queue class using the heap data structure. Here is the list of functions that you are supposed to complete: add -> Should accept new data. remove -> Should remove and return smallest value, if available or print "Underflow" otherwise and return -1. peek -> Should return smallest value, if available or print "Underflow" otherwise and return -1. size -> Should return the number of elements available. We have 3 different ways to write these functions and each way offers different complexities of each function. But in this article we will discuss the most efficient approach.

Please watch the question video carefully. The theoretical details of required functionality are explained in detail there. You need to implement the functions to achieve what is explained in the theoretical discussion in the question video. Input and Output has been managed for you.

For more clarity of the question, watch the question video

This is a modal window.

YouTube unknown error (150)

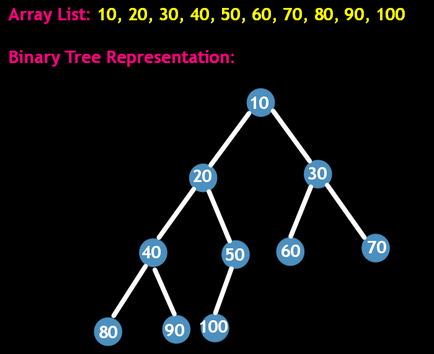
**2. Approach:**

The most efficient way to code our different functions is the one where add(data) and remove() takes O(log n) and peek() takes constant time. So here we will learn about this way. To achieve this we use a Heap data structure. Heap is a Binary Tree based data structure with two additional properties. And those properties are: Heap Order Priority (HOP): ➢ What is it? This means that the priority of a parent is always greater than its child. And to be specific, the priority of either child is not pre decided. ➢ Why is it? This property basically helps us to achieve the most efficient time complexity of the peek() function which is constant (O(n)). Complete Binary Tree (CBT): ➢ What is it? Supposing that the height of the tree is h, then according to this property, at least h-1 levels of the Heap should be completely filled. And the last level should be filled from left to right. ➢ Why is it? Use of this property again helps us to achieve the most efficient time complexity of add(data) and remove() functions when used together which is O(log n) of each. For more clarity; watch part of the video lecture

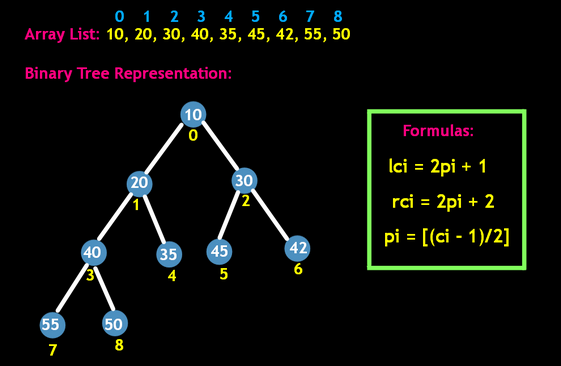
For more clarity of the question, watch the question video

Play Video

So, talking of implementation, what is important to know? It is important that you know, the data structure used for storage by the heap is Array List. Now you may say that, above, we said Heap is a Binary Tree based data structure. Yes! We said that and this may be a little confusing. But please pay attention. We said that it's a Binary Tree based data structure and with this we implied on its visualization part. We can visualize it as a binary tree as well.

****

But using the view of Binary Tree makes the implementation look easy. Let's say the Array List view is the usual view and Binary Tree has an unusual view. Keeping the properties of Heap in mind and trying to make the functions efficient, let's get started. Consider the below given array list and its corresponding binary tree representation.

****

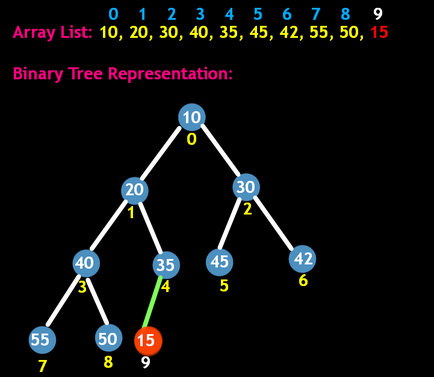
If you keenly observe the indices of array list in binary tree representation specifically each parent-child relation than we can generalize this using formulas: lci (left child index) = 2 \* pi (parent index) + 1 rci (right child index) = 2 \* pi (parent index) +2 And using these formulas, we can not only access the child's index from the parent but also the parent's index from either child unlike Binary Trees. pi (parent index) = [ci (child index) -1/2] So, we can say that by just visualizing binary tree representation of heap, we got some extra powers! Not to forget, these powers are applicable until the properties of Heap are followed. For more clarity; watch part of the video lecture

For more clarity of the question, watch the question video

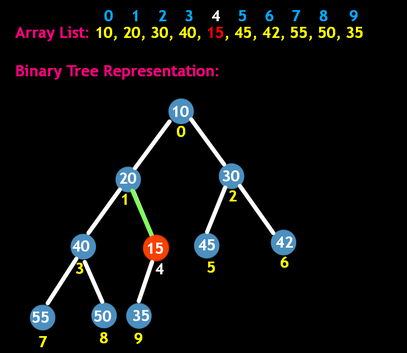
Play Video

ADD(DATA):

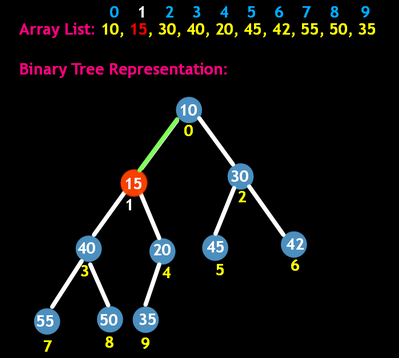
Suppose we have to add 15 to the above tree. We should add this element to the array list such that properties of Heap are not violated. ➢ We start with placing the element at the end of the array list.

****

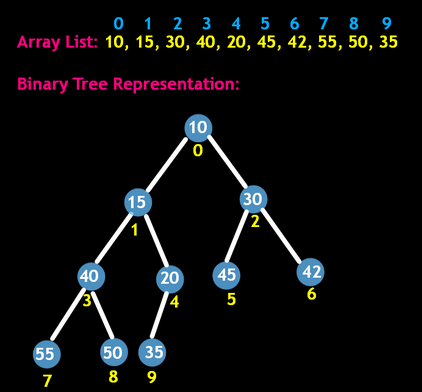
At this point, in the Binary Tree representation of this tree, it can be seen that the CBT property is still valid but HOP property has been violated as 15 has more priority than its parent 35. ➢ So to honor the HOP property, it is necessary that we Up-heapify a Binary Tree representation and simultaneously make changes in Array List. ➢ To up-heapify the binary tree, we start with swapping the parent's data with the child's data which is 35 and 15 in this case respectively. ➢ On doing so, 15 becomes a parent of 35.

****

➢ Now again we will check whether 15 is still better than its new parent 20 or not. ➢ In this case also, 15 has again more priority than 20. So we swap 15 and 20.

****

➢ And yes, we again check this. But this time 15's new parent is 10 which has more priority than 15. So we stop at this point.

****

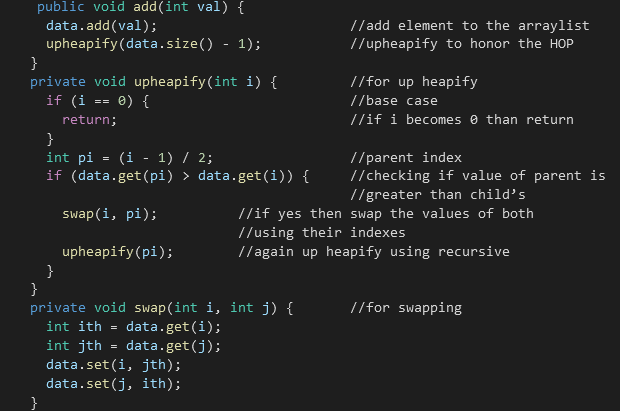
So, by following the above steps, we can say that Heap's property HOP has been restored.

If HOP is still applicable then it means the element with highest priority will be present at index 0 of Array List, which can be accessed in the constant time. Making the time complexity for peek() function O(1). And talking about time complexity of add(data), it takes constant time to add the element at the last in an Array List and log n time (because of tree's height) at max to up-heapify the added data. Making the overall time complexity O(log n). So to conclude, we can say that because of Heap's CBT property, we could use the array list's implementation and because of which we could run the indices formulas. By which we could access the parent's index and therefore swap the data and up heapify the tree. And that's why we could do this in order of O(log n). If no array list were used and we had to use only binary trees to implement Heap, then it would have been impossible to do this addition in this time complexity. Because it would have used some sort of traversal to reach the spot where swapping was to be done. And all the binary tree traversal, be it postorder, preorder, inorder or level order, takes a O(n) time to do so. For more clarity; watch part of the video lecture

For more clarity of the question, watch the question video

Play Video

Let's try to code the add(data) function:

****

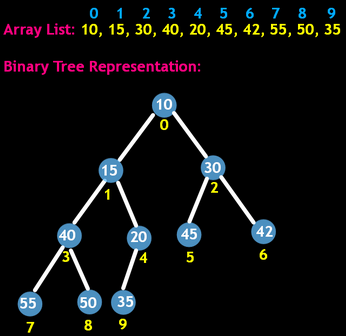
For more clarity; watch part of the video lecture

For more clarity of the question, watch the question video

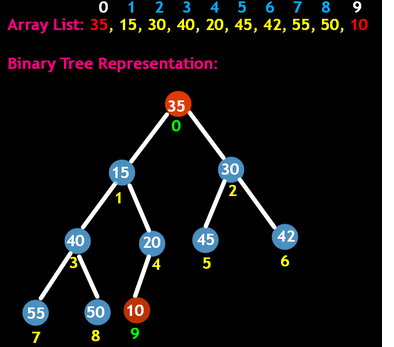
Play Video

REMOVE():

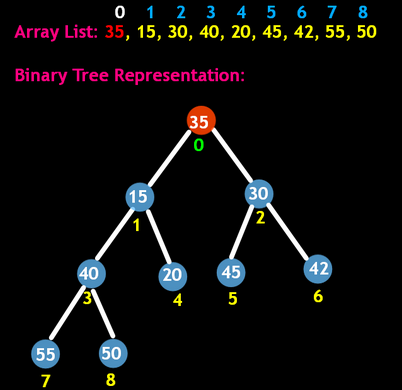
Suppose we have to remove an element from the given tree. To do so we follow these steps:

****

➢ First of all we swap the first element of the array list with the last index. In this case, we swap 10 with 35.

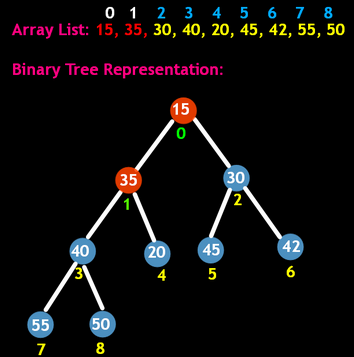
****

➢ And then remove the element from the last index, which is 10.

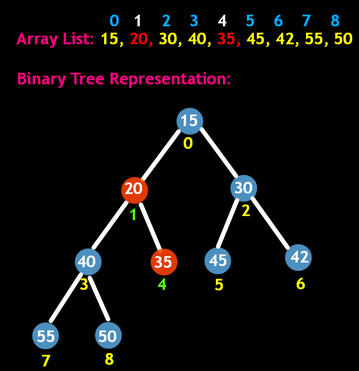
****

➢ Why didn't we remove this element directly from the first index?

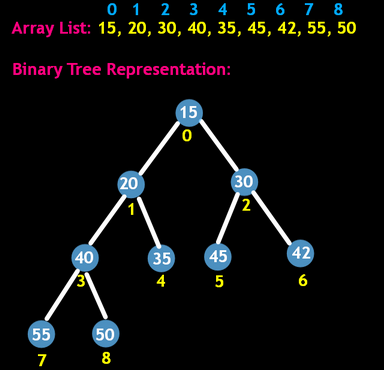
Because doing so will take O(n) time. As after removing the element we will need to shift each element to the left. ➢ Now we make use of down-heapify to honor the HOP property. ➢ So to use down-heapify, if the parent's priority is less than either child, we swap the parent's data with it's child's with maximum priority. ➢ In this case we swap 35 with 15 because 15 has highest priority among 35, 15 and 30. ➢ So 15 now becomes the new parent of 35 and 30.

****

➢ Now we again check, if the new position of 35 honors the HOP property or not. ➢ No it does not honor as its one of the new child, which is 20 has higher priority than 35. So we now swap 20 and 35.

****

➢ As 35 has no more children, we halt the process of down heapify

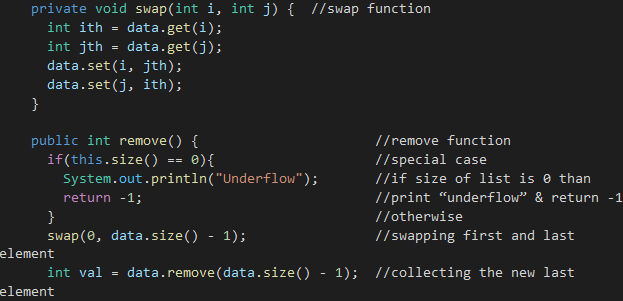
****

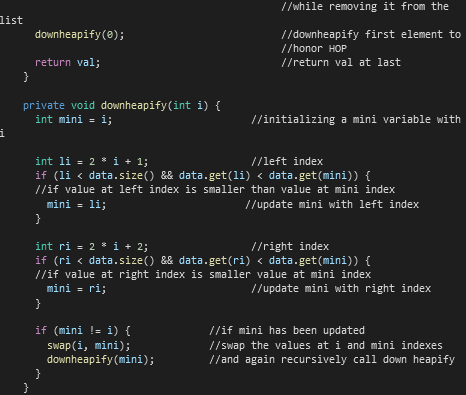
For more clarity; watch part of the video lecture

For more clarity of the question, watch the question video

Play Video

Let's try to code remove function:

****

****

For more clarity; watch part of the video lecture

For more clarity of the question, watch the question video

Play Video

Let's try to code peek function and understand it simultaneously:

****

To get at the peek at any time we just call peek(). It is a function of O(1) that is constant time. Inside this function, first of all we handle a special case; if the size of the list is 0. If the size of the list is 0 then we print underflow and return -1. Otherwise we return the value at the zeroth index (root of the tree basically). For more clarity; watch part of the video lecture

For more clarity of the question, watch the question video

Play Video

SIZE():

Let's try to code size function:

****

To get at the size of the priority queue at any time we just call size(). It is a function of O(1) that is constant time. Inside this function, we simply return the size of the arraylist in use.

**3. CODE:**

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

public static class PriorityQueue {

ArrayList< Integer> data;

public PriorityQueue() {

data = new ArrayList<>();

}

public void add(int val) {

data.add(val);

upheapify(data.size() - 1);

}

private void upheapify(int i) {

if (i == 0) {

return;

}

int pi = (i - 1) / 2;

if (data.get(pi) > data.get(i)) {

swap(i, pi);

upheapify(pi);

}

}

private void swap(int i, int j) {

int ith = data.get(i);

int jth = data.get(j);

data.set(i, jth);

data.set(j, ith);

}

public int remove() {

if (this.size() == 0) {

System.out.println("Underflow");

return -1;

}

swap(0, data.size() - 1);

int val = data.remove(data.size() - 1);

downheapify(0);

return val;

}

private void downheapify(int i) {

int mini = i;

int li = 2 \* i + 1;

if (li < data.size() && data.get(li) < data.get(mini)) {

mini = li;

}

int ri = 2 \* i + 2;

if (ri < data.size() && data.get(ri) < data.get(mini)) {

mini = ri;

}

if (mini != i) {

swap(i, mini);

downheapify(mini);

}

}

public int peek() {

if (this.size() == 0) {

System.out.println("Underflow");

return -1;

}

return data.get(0);

}

public int size() {

return data.size();

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

PriorityQueue qu = new PriorityQueue();

String str = br.readLine();

while (str.equals("quit") == false) {

if (str.startsWith("add")) {

int val = Integer.parseInt(str.split(" ")[1]);

qu.add(val);

} else if (str.startsWith("remove")) {

int val = qu.remove();

if (val != -1) {

System.out.println(val);

}

} else if (str.startsWith("peek")) {

int val = qu.peek();

if (val != -1) {

System.out.println(val);

}

} else if (str.startsWith("size")) {

System.out.println(qu.size());

}

str = br.readLine();

}

}

}

We hope that this article was helpful. If somehow you are finding it d