Lecture 18

Heaps and HeapSort

Last time:

We looked at TreeMap and TreeSet of the Java Collections Framework.

More specifically, our focus was, in what situations, they can be useful.

TreeMap can be useful to sort all of the elements that are in HashMap.

TreeSet is also very useful to sort or perform a non-exact match search.

Besides, TreeSet provides many useful methods that can be found on its API doc.

Since they are implemented based on self-balanced tree (Red-Black tree), their major operations are guaranteed to have O(log N) running time complexity.

Another important thing to remember is that all of the objects that are being added into TreeMap or TreeSet data structure must be sortable.

In other words, you need to implement Comparable or Comparator interface for the object class that you intend to put into TreeMap or TreeSet.

Today, we will look into our last data structure, Heap!

In this lecture, we will look into Max-Heap.

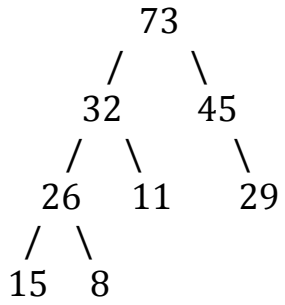
Step 1: Conceptual View

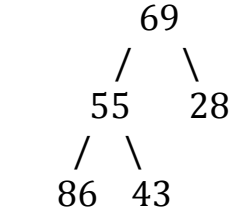
A max-heap is a binary tree with the following characteristics:

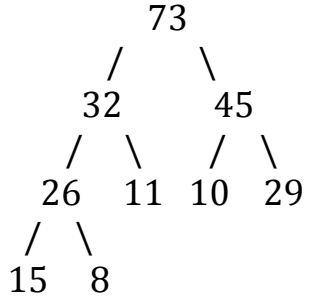
- It's complete (Almost). Every level of the tree has the maximum number of nodes except possibly the last level. It's filled in reading from left to right across each row.
- The largest data is in the root.
- For every node in the max-heap, its children contain smaller key value.

Examples

Are these heaps?



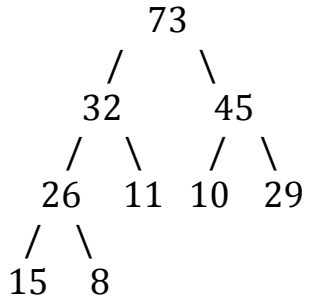




A few analyses

- A heap is **NOT** a sorted structure and can be considered as **PARTIALLY ordered**.
- What can we learn from the fact that a heap is a complete binary tree? It should always have the smallest possible height which is O(_____).
- It is a useful data structure when we need to keep getting the object with the highest priority.

Add (insert) a new data into a Max-Heap



Given the max-heap above, insert 51 into the heap.

Here is the algorithm (*Percolate up*)!

1. Insert a new data into the next available tree position so that the tree remains **complete**.
2. Swap the data with its parent(s), if necessary, until the tree is **restored to be a heap**. (Refer to the characteristics of the max-heap.)

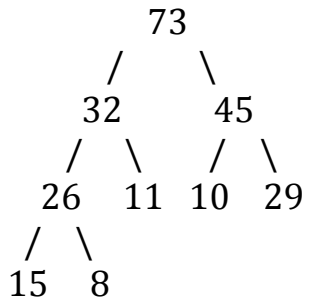
Knowing the algorithm for insertion, how do we build a max-heap?

Practice: Build a max-heap using given values.
26, 32, 45, 10, 29, 8, 11, 9, 73, 15

Remove the maximum data from a max-heap

Here is the algorithm (*Percolate down*)!

1. Remove root (the maximum value) and replace it with the last node of the lowest level to make sure the tree **stays complete**.
2. Swap the data with its larger child, if necessary, until the tree is **restored to be a heap**.

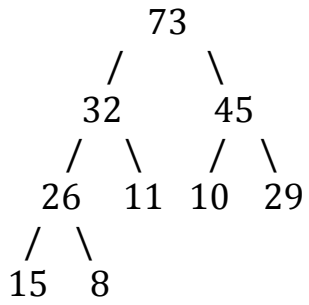


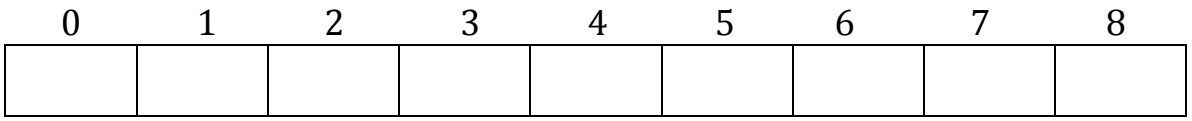
Given the max-heap above, remove the maximum data (root).

Step 2: Implementation View

So far, we looked at a max-heap in the form of a binary tree and its two major operations (insertion and remove max). But the heap in the form of a binary tree is only a conceptual representation.

How can it be implemented?





What do we see here?

- A heap is complete which means there cannot be any hole in the array. Every cell should be filled (from 0 to N-1).
- Maximum value is in the root. That means it is at the first index of the array.
- Once again, a heap is not a sorted structure which means values in the array are not sorted. (but *partially ordered*.)

Major operations we want to implement in our max-heap:

- Insert
- Remove Max

Let's have our interface first.

```
public interface HeapInterface {
      /**
       * Insert a new value into a heap
       * @param value value to be inserted
       * @return boolean to check whether it was added or not
       */
      boolean insert(double value);

      /**
       * remove the highest priority value (maximum for max heap)
       * @return removed value
       */
      double removeMax();
}
```

Here is a skeleton of the max-heap class.

```
public class Heap {
      private Node[] heapArray;
      private int currentSize;

      public Heap(int initialCapacity) {
            heapArray = new Node[initialCapacity];
            currentSize = 0;
      }

      public boolean insert(double value) {
           // TODO implement this method
      }

      public double removeMax() {
           // TODO implement this method
      }
}
```

Just like BST, we need a node class as a private static nested class.

```
// private static nested Node class
private static class Node {
      private double value;  // data
      public Node(double value) {
            this.value = value;
      }
}
```

Inserting a new data into Max heap.

Basic steps for insertion:

1. Insert a new data into the next available tree position so that the tree remains complete.
2. Exchange data with its parent(s) (if necessary) until the tree is restored to be a heap (*Percolating up*!)

```
public boolean insert(double value) {
      if(currentSize == ____________________) return false;
      Node newNode = new Node(value);
      heapArray[____________________________] = newNode;
      percolateUp(_______________________________);
      return true;
}

private void percolateUp(int index) {
      // save the bottom node, newly added one
      Node bottom = heapArray[index];

      // find the parent index
      int parent = (____________________________)/2;

      // parent's value is smaller than the new value
      while(index > 0 && heapArray[___________________].value <
bottom.value) {
            heapArray[index] = heapArray[_____________________];
            index = _________________________; // move up
            parent = (_________________)/2; // new parent
      }

      heapArray[index] = ____________________; // proper location
}
```

Removing the max value from the heap.

Three basic steps for removal:

1. Remove the root
2. Move the last node into the root
3. Percolate or trickle down the last node where it is below the bigger nodes and above the smaller nodes

```
public double removeMax() {
      Node root = heapArray[_____________];
      heapArray[0] = heapArray[_____________]; // root<-last one
      percolateDown(_____________); // percolate down
      return root.value;
}

private void percolateDown(int index) {
      Node top = heapArray[_________________________];
      int largerChild; // larger child's index
      // loop till the index that has a child
      while(index < ______________________________) {
            int leftChild = ______________________________;
            int rightChild = leftChild + 1;
            // find which one is larger child
            if(rightChild < ______________________________ &&
                        heapArray[leftChild].value <
heapArray[rightChild].value) {
                  largerChild = rightChild;
            } else {
                  largerChild = leftChild;
            }
            // if top is bigger, then stop
            if(top.value >= heapArray[____________________].value)
                  break;
            // move the values up
            heapArray[index] = heapArray[________________________];
            index = ______________________; // go down
      }
```

```
    // put top value into proper location to restore the heap
    heapArray[index] = _____________________;
}
```

Efficiency of insert(double value) and removeMax()
Where do we spend most of time for these operations?

One important thing you need to notice is that while percolating up or down, we did not really swapped as we mentioned in Step1 : Conceptual View.

One swap requires _____ copies. But, in our implementation, we have reduced the number of copies.

For example, we have four nodes and need to swap all of them.
A ←→ B ←→ C ←→ D

How many copies do we need to perform?

How about our implementation? Let's draw its operations!

How many copies do we need to perform?

This will reduce the running time, especially in case heap is big.
However, *the main point here is that the number of comparisons and necessary copies is bounded by the _____ of the heap.*

Of course, there are other moves but they are constant for the operations. Thus, we can ignore.

In conclusion, we can say that the two operations perform in O(log N) time.

Now that we know those major operations of a heap have the running time complexity of O(log N), we might have this question:

"What if we use this for sorting?"

This sorting algorithm is called obvisously ***HeapSort***.

Basic idea is:

- First, insert unordered items into a heap using insert() method to build a heap.
- And, call removeMax() for max heap or removeMin() for min heap repeatedly so that we get the items in sorted order.

```
for(int i=0; i<size; i++)
    theHeap.insert(anArray[i]);
for(int i=0; i<size; i++)
    anArray[i] = theHeap.remove();
```

insert() and removeMax() methods run in O(logN) time.
Each must be applied N times which makes the entire sort run in O(_____) time.

Priority Queue

Priority Queues may be used for any application that requires processing items based on some priority, such as task scheduling in computers. The major operations in priority queues are insert(object) and delete().

The Java Collections Framework has PriorityQueue class that is Heap implementation. (By default, min heap)

Its major operations and their running time complexities are:

- add(object) or offer(object) : inserting a new element O(log N)
- poll() : removing the head _____
- peek() : finding the head _____
- remove(object) : remove an element _____

The PriorityQueue class of the Java Collections Framework does not guarantee to traverse the elements in any particular order since it is based on heap.

Comparison with BSTs

Even though two of them are binary trees, they are conceptually different and their implementations are fundamentally different.

Which tree is for easier searching?

Which tree is for retrieving the maximum value quickly?

Which tree is guaranteed to be complete or almost-complete?

Which tree is guaranteed to be balanced?

Time complexity

	Binary Search Tree	Max Heap (Min Heap)
Insertion	_____ (average: O(logN))	
Deletion	_____ (average: O(logN))	
Search	_____ (average: O(logN))	
Peek Max	-	