Lecture Notes

Priority Queues and Heaps

Welcome to session on Priority Queues and Heaps.

In the module of Stacks and Queues you learnt that the element entering first will be the element to come out first and that is called queues while, if the element going in last comes out first then it's called stack.
But this is always not the case, sometimes it may happen that the element going in first may have to come out first or the element going in first may have to come out first.
Consider the example of airline checkin queue- If you are in the checkin queue to board a flight but your flight is about to leave and many people are there infront of you for their checkin. So what will you do in that scenario.

You will certainly not wait for your turn because you might loose your flight, you will try to jump the queue and checkin early. This type of situation represents a priority queue.

Check-In Queue In Airport

The man who came in last has to leave early here, so his checkin will be done first.

Priority Queues

Consider a sports club who wants to sell its ticket for a match to the waitlisted customer it has. There are 4 types of tickets available club ticket, season ticket, online ticket and counter ticket and each of them are of different priority or are of different cost i.e club ticket being the highest priority cost the most while counter ticket cost the least and is of lowest priority. So what will be the order in which their tickets will be confirmed? Will it be first come first serve or of FIFO kind? Not necessarily, the club would certainly want to maximise its profit so it will sell the ticket which cost the most first.

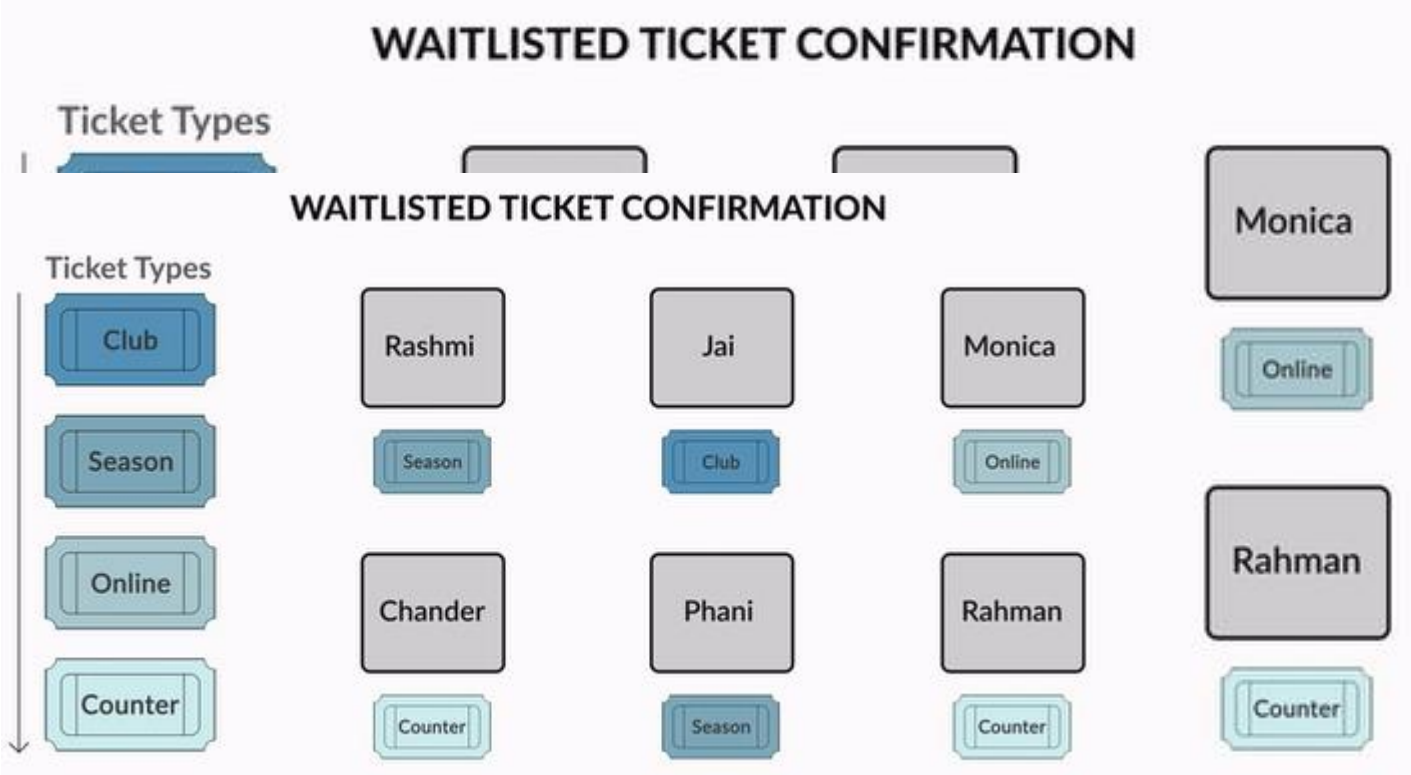
This structure of selling tickets on the basis of their priority and not in the order they came in is analogous to an abstract data structure called "**Priority Queue**". In a Priority queue, the element with the highest priority will go out first and the one with least priority will go in last.
In computers the highest priority element is shown with least integral value of priority In case of tickets club ticket will get a priority of 1 and counter ticket will get a priority of 4.
A Priority queue is therefore anything:

1. into which you remove (removeMin) things from the front of the queue.

2. that removes the highest priority element first from the queue.

Priority Queue Operations

Considering the waitlisted ticket example –



In the above example club has the highest priority and counter the lowest. So, the order in which ticket will be confirmed will be starting with Jai then Rashmi, Phani, Monica, Chander and lastly Rahman.
There are two types of priority queue min and max but we will learn only about min-priority queue and java ADT class also implements min-priority queue.
Priority Queue ADT has a class in java for it through which you can directly implement some functions like:
poll() : It returns the minimum element from the priority queue as it will have the highest priority and then removes that element.
peek() : It returns the smallest element from the priority queue as it will have the highest priority .
add(T element) : It adds an element in the priority queue of type T.
isEmpty(), which returns true if the priority queue is empty and false otherwise.
size() which returns the size of the priority queue.

To implement priority queue you need to implement comparator function if the object type is generic.
If you change the comparator accordingly you can implement max-priority queue also.

Implementing a Priority Queue

You could either implement a priority queue using a custom class made using lists (array lists or linked lists) such as the one below, where you used a linked list to implement a Priority Queue.

In the following program, you have created the basic functions of priority queu.

```
import java.util.Comparator;
import java.util.List;

public class ListPQ<T> {

  protected final Comparator<T> comparator;

  public ListPQ(Comparator<T> comparator) {
    this.comparator = comparator;
  }

  public void add(T element) {
    if(!this.isEmpty()) {
      for(int i = 0; i < this.list.size(); i++) {
        T e = this.list.get(i);
        if(this.comparator.compare(e, element) >= 0) {
          this.list.add(i, element);
          return;
        }
      }
    }
    this.list.add(element);
  }

  public T removeMinimum() {
      return this.list.remove(0);
  }

  public T minimum() {
```

```
    return this.list.get(0);
  }
   public boolean isEmpty() {
       return this.list.isEmpty();
  }

   public int size() {
       return this.list.size();
  }

}
```

You could also use the internal Priority queue library provided by JAVA, which you could call using -

```
import java.util.PriorityQueue;
```

The features poll, peek, isEmpty and size can all be called using this library.

So the Priority Queue code implemented using this library would have looked like the following:

```
import java.util.PriorityQueue;
import java.util.List;

public class PQImplement<T> {

public static void main(String[] arg) {
      PriorityQueue <Integer> PQ = new PriorityQueue <Integer> ();

         for(int i=2; i<=20; i=i+2) {
             PQ.add(new Integer (i));
         }
      int x= PQ.peek();
      int y= PQ.poll();
      int z= PQ.size();

    }
}
```

There might be elements, such as custom objects, that don't have a predefined order. In such cases, you can utilise a comparator interface to create a custom order of the elements. Here is the code for comparator function.

```
Comparator

compare(a,b){
    if a<b
        return -1

    else if a>b
        return 1

    else
        return 0
        }
```

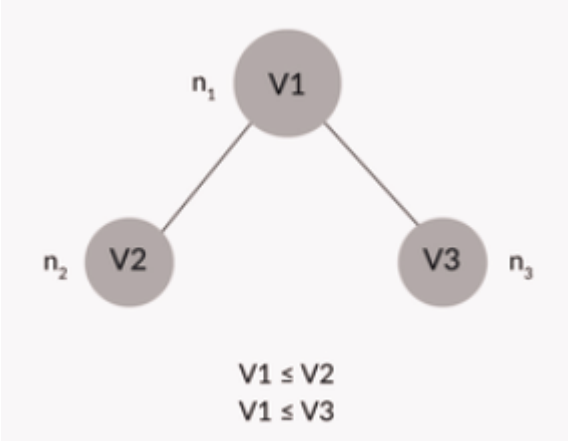
Heaps

	Insert	FindMin	RemoveMin
Unordered ArrayList	O(1)	O(n)	O(n)
Unordered LinkedList	O(1)	O(n)	O(n)
Ordered ArrayList	O(n)	O(1)	O(1)
Ordered LinkedList	O(n)	O(1)	O(1)

Lists are not the most efficient way to implement Priority queue, as you can see in the above table. Overall Big O for all operations is O(n). So, we have an alternative implementation for that known as Binary Heaps. Heaps can implement all of the operations of Priority Queue in maximum O(log n) time.

Binary Heap has two basic properties:

- They are complete binary trees
- Order property: They maintain a hierarchical (level-wise) order among the nodes of their trees.

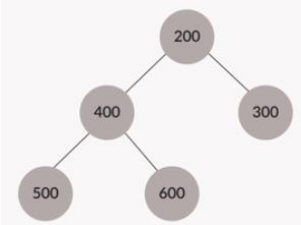


The order among the nodes at the same level does not matter and duplicate nodes can exist in a heap. Also, there are two types of heaps:

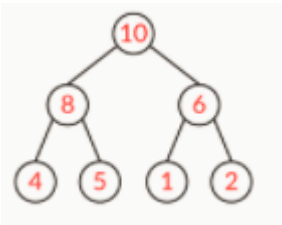
- Min-heap
- Max-heap

In min-heap the minimum element is always at the root node. While in max-heap maximum element is always at the root node.

Min-Heap:



Max-Heap:



Addition or removal from a heap is done always from the last node. The minimum value, which is at the root node, cannot be removed directly. First, it is swapped with the last node and then it is deleted from the last node.

Any modifier operation(add or removeMin) performed on heap will lead to the disturbance of heap property. So, we have to perform Heapify operation to restore the heap property:

There are two heapify operations:

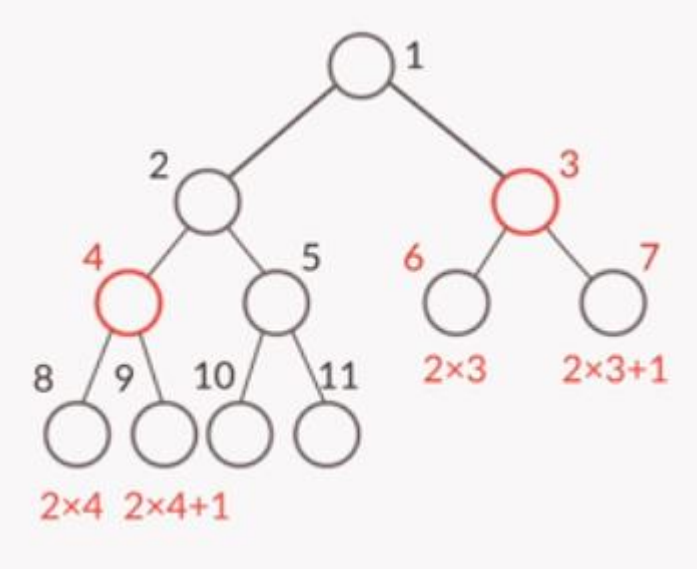
- HeapifyUp is used during insertions
- HeapifyDown is used during deletions

Both of these operations are recursive as the property violations can bubble up to the root or bubble down to the leaf node.

Here is the pseudo code for HeapifyDown operation:

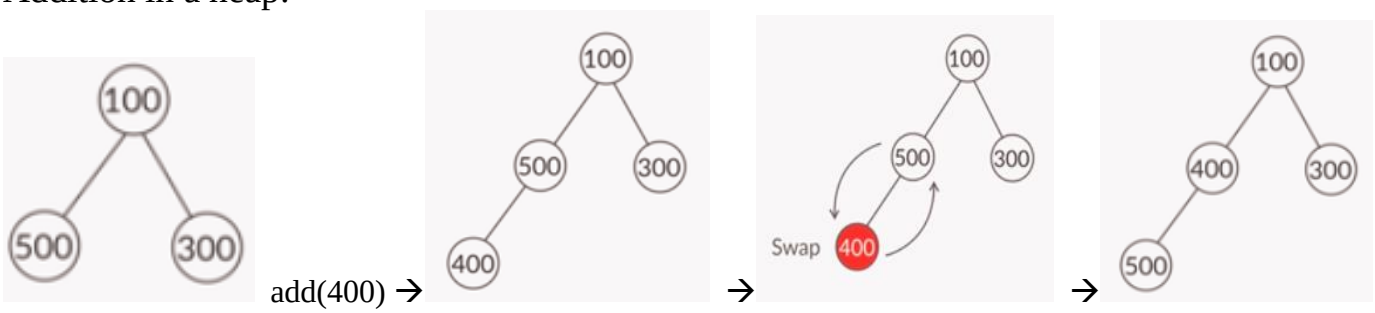
```
HeapifyDown(A,Index,n)
   min =Index
   l = 2*i
   r = 2*i +1

   If l<n and A[l]<A[min]
         min=l

   If r<n and A[r]<A[min]
         min=r

   If min != Index
         swap(A[min], A[Index])
         HeapifyDown(A,min,n)
```

Heap in Java is implemented as an array. In the array, the root node is the initial element of the array and any child node's index is 2*i or 2*i+1, if the parent's index is i. In this implementation the array starts with index 1.

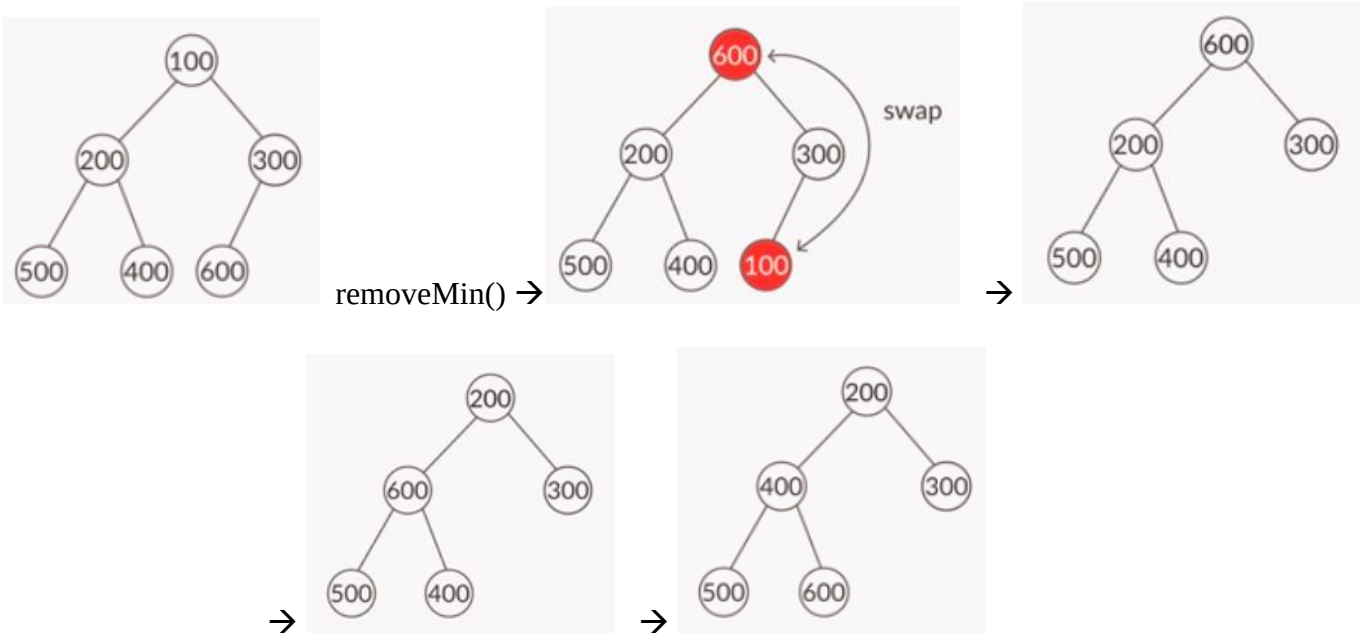


Addition and Removal in a Heap

Addition in a heap:



Remove Minimum in a heap:



Both addition and deletion in a heap takes O(log n) time.

HeapSort

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure(Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest(depending upon Max-Heap or Min-Heap), so we put the first element of the heap in our array. Then we again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. We keep on doing the same repeatedly untill we have the complete sorted list in our array. So, Priority Queues can be implemented using a heap to better its performance. Heap sort is not a Stable sort, and requires a constant space for sorting a list.

Application of Priority Queues and Heaps

There are many uses of priority queues and heaps. Some of them are:

- Event-driven simulation: customers in a line
- Collision detection: "next time of contact" for colliding bodies
- Data compression: Huffman coding
- Graph searching: Dijkstra's algorithm, Prim's algorithm
- AI Path Planning: A* search
- Statistics: maintain largest M values in a sequence
- Operating systems: load balancing, interrupt handling
- Discrete optimization: bin packing, scheduling
- Spam filtering: Bayesian spam filter

Summary

Here's a quick summary of the topics you learnt about in this session:

1. Priority queues: This is an abstract data type (ADT) that resembles a regular queue or a stack data structure, but this data structure has the functionality to add a 'priority' to each of its elements to make organising data more efficient. In a priority queue, a high-priority element is served before a low-priority element. If a priority queue has two elements with the same priority, the order of the elements in the queue determines which one gets served first.

   In its most basic form, a priority queue ADT supports the following operations:

   - poll(): Removes an instance of the minimum element from a priority queue and returns the element.
   - peek(): Returns the minimum element of a priority queue but does not make any changes to the state of the priority queue.
   - add(T element): Adds an element to a priority queue.
   - isEmpty(): Checks whether a priority queue is empty or not and returns 'True' if the priority queue is empty and 'False' otherwise.
   - size(): Returns the number of elements in a priority queue.

   You learnt that a normal queue data structure could not implement a priority queue efficiently because searching for the highest-priority element will take O(n) time. A list, whether sorted or not, will also require O(n) time for either insertion or removal.
   Instead, you found a data structure that is guaranteed to give good performance in this special application.

   This session also took you through the heap data structure, which inserts and removes elements in O(log n) time. A heap has two basic properties. First, it is a complete binary tree; and second, it is partially ordered. This means that there is always a relationship between a child node and its parent's node. There are two variants of the heap data structure, depending on the parent-child relationship.

   - Max-heap: It is a heap data structure in which all the child nodes are less than or equal to their parents. As the root stores a value that's greater than or equal to its children's values, the root holds the maximum of all values in the tree.

   - Min-heap: It is a heap data structure in which all the child nodes are greater than or equal to their parents. As the root stores a value that's less than or equal to its children's values, the root holds the minimum of all values in the tree.

   You can sort a sequence of elements efficiently by using a heap-based priority queue. This algorithm is called **heap sort**.