Priority Queue

- A queue that maintains order of elements according to some priority
- Contrast to Queue which is FiFo

- **PriorityQueues are about the order in which things are removed, NOT the way in which they are stored.**
  - the items may or may not be sorted, or otherwise arranged.
  - This statement applies to stack and queues also, it is just convenient in those cases to arrange data to make retrieval easy
# Complexity Analysis

<table>
<thead>
<tr>
<th></th>
<th>Unordered</th>
<th>Ordered (using SAL)</th>
<th>Heap Based</th>
</tr>
</thead>
<tbody>
<tr>
<td>offer</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(lg n)</td>
</tr>
<tr>
<td>peek</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>poll</td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(lg n)</td>
</tr>
</tbody>
</table>

Unordered PQ == Selection Sort
Ordered PQ = Insertion Sort
Binary Heap

- A heap is a “binary tree” storing keys at its nodes and satisfying:
  - heap-order: for every internal node \( v \) other than root, \( key(v) \geq key(parent(v)) \)
  - Heap is filled from top down and within a level from left to right.
    - at depth \( h \), the leaf nodes are in the leftmost positions
    - last node of a heap is the rightmost node of max depth
# Binary Tree — terms

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>A part of a tree.</td>
</tr>
<tr>
<td>Parent</td>
<td>A node that has children</td>
</tr>
<tr>
<td>Child</td>
<td>A node that has parents. Child nodes have exactly one parent</td>
</tr>
<tr>
<td>Binary Tree</td>
<td>A structure of nodes such that parent nodes have at most two children</td>
</tr>
<tr>
<td>Root</td>
<td>The node in a tree that has no parent.</td>
</tr>
<tr>
<td>Leaf</td>
<td>Any node that has no children</td>
</tr>
<tr>
<td>Height</td>
<td>The maximum distance from a the root node to a leaf.</td>
</tr>
<tr>
<td>Subtree</td>
<td>The part of a tree whose root is a given node</td>
</tr>
</tbody>
</table>
Height of a Heap

• A binary heap storing $n$ keys has a height of $O(\log_2 n)$

• This is NOT true for general binary trees
Insertion into a Heap

• Insert as new last node
• Need to restore heap order
Upheap

• Restore heap order
  ▫ swap upwards
  ▫ stop when finding a smaller parent
  ▫ or reach root

• $O(\log n)$
Poll

- Removing the root of the heap
  - Replace root with last node
  - Remove last node
  - Restore heap order
Downheap

• Restore heap order
  ▫ swap downwards
  ▫ swap with smaller child
  ▫ stop when finding larger children
  ▫ or reach a leaf
• $O(\log n)$
Heaps are built on Arrays

- Parent from child
  - suppose child is at location childLoc in array
    - parentLoc = (childLoc-1)/2
- Child from Parent
  - suppose parent is at parentLoc in array
    - leftChild = parentLoc*2+1
    - rightChild = parentLoc*2+2

Locations of Parents and children are in strict mathematical relationship

- Parent from child
  - child at loc 4 (value 7)
    - parent is at (4-1)/2 = 1 (value 5)
- Child from Parent
  - parent at loc 2 (value 6)
    - leftChild =2*2+1 = 5 (value 1)
    - rightChild = 2*2+2 = 6 (value — not used)
Priority Queue using Heaps

startup

```java
public class PriorityQHeap<K extends Comparable<K>, V extends AbstractPriorityQueue<K, V>> {
    private static final int CAPACITY = 1032;
    private Pair<K, V>[] backArray;
    private int size;

    public PriorityQHeap() {
        this(CAPACITY);
    }

    public PriorityQHeap(int capacity) {
        size=0;
        backArray = new Pair[capacity];
    }

    @Override
    public int size() {
        return size;
    }

    @Override
    public boolean isEmpty() {
        return size==0;
    }
}
```
Heap Insertion
Priority Queue offer method

public boolean offer(K key, V value)
1. Ensure there is room — if not return false
2. Add new items to end of heap (low and left viewed graphically)
   first unoccupied viewed array-wise
3. Repeat until at root
   1. Compare with parent
   2. If greater, swap and continue
   3. If less stop
4. return true
Peek and Poll

```java
@Override
public V poll() {
    if (isEmpty())
        return null;
    Entry<K,V> tmp = backArray[0];
    removeTop();
    return tmp.theV;
}

@Override
public V peek() {
    if (isEmpty())
        return null;
    return backArray[0].theV;
}
```
private void removeTop()
{
    backArray[0] = backArray[size-1];
    backArray[size-1]=null;
    size--;
    int upp=0;
    while (true)
    {
        int dwn;
        int dwn1 = upp*2+1;
        if (dwn1>size) break;
        int dwn2 = upp*2+2;
        if (dwn2>size) {  dwn=dwn1;
        } else {
            int cmp = backArray[dwn1].compareTo(backArray[dwn2]);
            if (cmp<=0)  dwn=dwn1;
            else dwn=dwn2;
        }
        if (0 > backArray[dwn].compareTo(backArray[upp]))
        {
            Pair<K,V> tmp = backArray[dwn];
            backArray[dwn] = backArray[upp];
            backArray[upp] = tmp;
            upp=down;
        }
        else { break; }
    }
}
General Removal

- swap with last node
- delete last node
- may need to upheap or downheap

Heap:

```
  1
 /   \
5     6
  \
  9
   /
 11
  /  \
9    11
   /  \
17   15
    /  \
   17  19
    /  \
   21 21
    /  \
  27 22
```
Heap Insertion
Priority Queue offer method

```java
public boolean offer(K key, V value) {
    if (size >= (backArray.length - 1))
        return false;
    // put new item in at end data items
    int loc = size++;
    backArray[loc] = new Pair<K, V>(key, value);
    // up heap
    int upp = (loc - 1) / 2; // the location of the parent
    while (loc != 0) {
        if (0 > backArray[loc].compareTo(backArray[upp])) {
            // swap and climb
            Pair<K, V> tmp = backArray[upp];
            backArray[upp] = backArray[loc];
            backArray[loc] = tmp;
            loc = upp;
            upp = (loc - 1) / 2;
        } else {
            break;
        }
    }
    return true;
}
```