• A queue that maintains order of elements according to some priority

• Contrast to Queue which is FiFo
  • PriorityQueue can implement a stack or a queue

• PriorityQueue 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.
  • Aside: 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</th>
</tr>
</thead>
<tbody>
<tr>
<td>offer</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>peek</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>poll</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Add N items, then Remove N items</td>
<td>Add:O(n) Remove: O(n<em>n) Overall:O(n</em>n)</td>
<td>Add:O(n<em>n) Remove: O(n) Overall: O(n</em>n)</td>
</tr>
</tbody>
</table>
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>
<th>Definition</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Node</td>
<td>A part of a tree.</td>
<td></td>
<td>2,5,7,9,7,1</td>
</tr>
<tr>
<td>Parent</td>
<td>A node that has children</td>
<td></td>
<td>2,5,6</td>
</tr>
<tr>
<td>Child</td>
<td>A node that has parents. Child nodes have exactly one parent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Tree</td>
<td>A structure of nodes such that parent nodes have at most two children</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Root</td>
<td>The node in a tree that has no parent.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Leaf</td>
<td>Any node that has no children</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Height</td>
<td>The maximum distance from a root node to a leaf.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Subtree</td>
<td>The part of a tree whose root is a given node</td>
<td></td>
<td></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
- \(O(\log n)\)
• 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)
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

```java
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
   1. If at root, STOP
   2. Compare with parent
   3. If greater, swap the GoTo 3.1
   4. stop (less -- or equal -- so do not need to keep going up)
4. return true
Peek and Poll

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

@Override
def public V peek() {
    if (isEmpty())
        return null;
    return backArray[0].theV;
}
```
Remove Top

In English
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=dwn;
        }
        else {    break;
        }
    }
}
General Removal

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

Heap:

```
   1
  /   
5     6
  / 
 9   8
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    11  22
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```

```
   1
  /   
9     11
     /  
    17  21
```

```
   1
  /   
9     11
     /  
    19  27
```
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;
}
```
# Complexity Analysis

<table>
<thead>
<tr>
<th></th>
<th>Unordered</th>
<th>Ordered</th>
<th>Heap Based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>offer</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(lg n)</td>
</tr>
<tr>
<td><strong>peek</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td><strong>poll</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td>O(lg n)</td>
</tr>
<tr>
<td><strong>Add N items, then Remove N items</strong></td>
<td>Add: O(n) Remove: O(n<em>n) Overall: O(n</em>n)</td>
<td>Add: O(n<em>n) Remove: O(n) Overall: O(n</em>n)</td>
<td>Add: O(n * lg n) Remove: O(n * lg n) Overall: (n * lg n)</td>
</tr>
</tbody>
</table>
Offer N followed by Poll N is sorting!!!!

- PQ on unordered == Selection Sort
- PQ on ordered == Insertion Sort
- PQ on Heap == Heap Sort
Selection Sort

- **Selection-sort:**
  - in place algorithm given an array with N items:
    - step 1: find the min from 0..(N-1) in array and swap with item in position 0
    - step 2: find min from 1..(N-1) in array and swap with item in position 1.
    - etc
- priority queue implemented with an unsorted array / arrayList / ...
- **Time:**
  - $O(n^2)$
  - In terms of priority $Q$, can split this into two phases
    - insertion $== O(N)$
    - polling $== O(N^2)$
## Selection Sort — Example

<table>
<thead>
<tr>
<th>Phase 1 Inserting</th>
<th>Inserting</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>7</td>
<td>(7)</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
<td>[7,4]</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g</td>
<td>[7,4,8,2,5,3,9]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Phase 2</td>
<td>Polling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>[2]</td>
<td>[7,4,8,5,3,9]</td>
<td>search=4, shift=3</td>
</tr>
<tr>
<td>b</td>
<td>[2,3]</td>
<td>[7,4,8,5,9]</td>
<td>search=5, shift=1</td>
</tr>
<tr>
<td>c</td>
<td>[2,3,4]</td>
<td>[7,8,5,9]</td>
<td>search=2 shift=3</td>
</tr>
<tr>
<td>d</td>
<td>[2,3,4,5]</td>
<td>[7,8,9]</td>
<td>search=3, shift=1</td>
</tr>
<tr>
<td>e</td>
<td>[2,3,4,5,7]</td>
<td>[8,9]</td>
<td>search=1, shift=2</td>
</tr>
<tr>
<td>f</td>
<td>[2,3,4,5,7,8]</td>
<td>[9]</td>
<td>search=1, shift=1</td>
</tr>
<tr>
<td>g</td>
<td>[2,3,4,5,7,8,9]</td>
<td>[]</td>
<td>search=1</td>
</tr>
</tbody>
</table>
Insertion Sort

- Insertion-sort
  - in-place algorithm
    - public Comparable[] sort(Comparable[] arra)
  - Step 0: start with item in position 0. Now the items in positions 0..0 are sorted
  - Step 1: look at item in position 1. Compare it to item in 0. If p1 is smaller, then swap. the items in position 0..1 are sorted with respect to each other
  - Step 2: determine where item in p2 should go in sorted list 0..N. If needed, For instance, bigger than 0 but smaller than 1. Make a space: save p1 into tmp. Shifting p1 into p2. Then put tmp into p1. Now the item in 0..2 are sorted.
  - Step N:

- Priority queue implemented with a sorted array/ ArrayList / ...

- Time:
  - $O(n^2)$
  - In terms of PQ
    - Add: $O(n^2)$
    - Remove: $O(n)$
  - Generally faster than selection sort
# Example

## Phase 1 — Inserting

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>7</td>
<td>(7)</td>
</tr>
<tr>
<td>(b)</td>
<td>4</td>
<td>(4,7)</td>
</tr>
<tr>
<td>(c)</td>
<td>8</td>
<td>(4,7,8)</td>
</tr>
<tr>
<td>(d)</td>
<td>2</td>
<td>(2,4,7,8)</td>
</tr>
<tr>
<td>(e)</td>
<td>5</td>
<td>(2,4,5,7,8)</td>
</tr>
<tr>
<td>(f)</td>
<td>3</td>
<td>(2,3,4,5,7,8)</td>
</tr>
<tr>
<td>(g)</td>
<td>9</td>
<td>(2,3,4,5,7,8,9)</td>
</tr>
</tbody>
</table>

## Phase 2 — polling

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a)</td>
<td>(2)</td>
<td>(3,4,5,7,8,9)</td>
</tr>
<tr>
<td>(b)</td>
<td>(2,3)</td>
<td>(4,5,7,8,9)</td>
</tr>
<tr>
<td>..</td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>(g)</td>
<td>(2,3,4,5,7,8,9)</td>
<td>()</td>
</tr>
</tbody>
</table>
Heap Sort

- Heap-sort:
  - Insertion — no more than $\log_2(n)$ steps per insertion
  - Deletion — no more than $\log_2(n)$ steps per deletion

- priority queue is most commonly implemented with a heap

- Time:
  - Add: $O(n \times \log_2(n))$ — doable in $O(n)$.
  - Remove: $O(n \times \log_2(n))$

- Note: with a lot of work can do this without an additional array.
Example

Phase 1 — Inserting
(a) 7  (7)
(b) 4  (4,7)
(c) 8  (4,7,8)
(d) 2  (2,4,8,7)
(e) 5  (2,4,8,7,5)
(f) 3  (2,4,3,7,5,8)
(g) 9  (2,4,3,7,5,8,9)

Phase 2 — polling
(a) (2)  (3,4,7,5,8,9)
(b) (2,3)  (4,5,7,9,8)
..  ..  ..
(g) (2,3,4,5,7,8,9)  ()
<table>
<thead>
<tr>
<th>size</th>
<th>selection</th>
<th>Insertion</th>
<th>Insertion (improved)</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16</td>
<td>15</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>2000</td>
<td>8</td>
<td>12</td>
<td>26</td>
<td>3</td>
</tr>
<tr>
<td>4000</td>
<td>24</td>
<td>23</td>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>8000</td>
<td>96</td>
<td>95</td>
<td>81</td>
<td>10</td>
</tr>
<tr>
<td>16000</td>
<td>370</td>
<td>378</td>
<td>315</td>
<td>17</td>
</tr>
<tr>
<td>32000</td>
<td>1585</td>
<td>1359</td>
<td>1218</td>
<td>36</td>
</tr>
<tr>
<td>64000</td>
<td>5771</td>
<td>5590</td>
<td>4605</td>
<td>77</td>
</tr>
<tr>
<td>128000</td>
<td>23087</td>
<td>21547</td>
<td>19849</td>
<td>161</td>
</tr>
<tr>
<td>256000</td>
<td></td>
<td></td>
<td></td>
<td>345</td>
</tr>
<tr>
<td>512000</td>
<td></td>
<td></td>
<td></td>
<td>1128</td>
</tr>
<tr>
<td>1024000</td>
<td></td>
<td></td>
<td></td>
<td>1973</td>
</tr>
<tr>
<td>2048000</td>
<td></td>
<td></td>
<td></td>
<td>3225</td>
</tr>
<tr>
<td>4096000</td>
<td></td>
<td></td>
<td></td>
<td>7577</td>
</tr>
<tr>
<td>8192000</td>
<td></td>
<td></td>
<td></td>
<td>18586</td>
</tr>
</tbody>
</table>

10000 == 1 second

anything below 1000 is very noisy