Priority Queues ADT

(i) insert /add - add new elements

(ii) DeleteMin / ExtractMin / remove

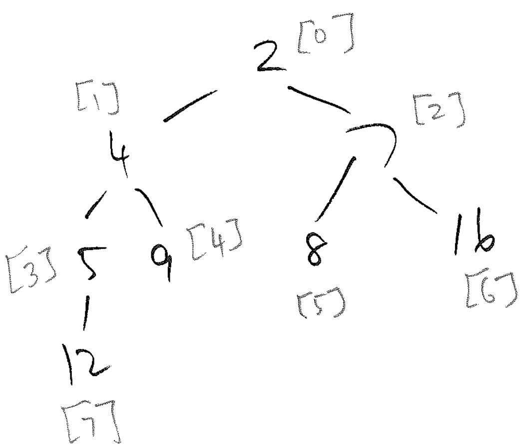
- remove element with max priority

Binary heap — implementation of PQ.

Complete binary tree - structure property

Value of node ≤ value of children — order property

Parent has higher priority



Size elements
↓
arr[0 ... Size-1]

remove():

```
min ← pq[0]                    ← move( )
pq[0] ← pq[size--]
percolateDown(0)
return min
```

percolateDown(i):   // Heap order may be violated at i, but only with its children

```
x ← pq[i]
c ← 2*i+1
while c ≤ size-1 do
    if c < size-1 and pq[c] > pq[c+1] then
        c++
    // c is smaller child of i
    if x ≤ pq[c] then break
    pq[i] ← pq[c]          // move( )
    i ← c
    c ← 2*i+1
                           // move(...)
pq[i] ← x
```

peek(): return pq[0]

Implementation of operations:

Heap occupies pq[0 .. n-1]

Height of tree = $\log_2 n$ RT of each op = $O(\log n)$.

helper function: parent(i) = return (i-1)/2

---

add(x):

```
if size ~~n~~ = pq.length then
    (a) resize to bigger array    ← Java
 or (b) throw exception           ← We do this.

pq[size] ← x
percolateUp(size)
size++
```

---

percolateUp(i): // Heap order may be violated at pq[i] with respect to parent of i.

```
x ← pq[i]
while i > 0 and x < pq[parent(i)] do
    pq[i] ← pq[parent(i)]      ← move(pq, i, pq[parent(
    i ← parent(i)
pq[i] ← x                      ← move(pq, i, x)
```

Minimum spanning Trees (MST): Prim's algorithm.

Indexed priority queues:

PQ + index of each element is stored in the element.

PQ

move(pq, i, x):
  pq[i] ← x

Indexed PQ

move(pq, i, x):
  pq[i] ← x
  x.putIndex(i)

percolateUp(index of v in q):

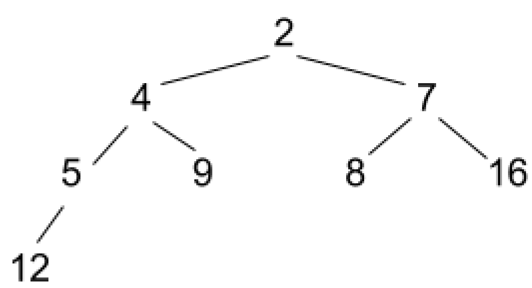
  percolateUp(v.getIndex())

<u>Priority Queues</u>: Each element has a priority. Elements are removed in the order of their priorities. Traditionally, smaller the value of priority, higher its priority. Duplicates are allowed, but not null values.

Operation	**Traditional name**	**Java method**
Insert a new element	Insert (x)	add (x), offer (x)
Remove element with maximum priority	DeleteMin ( ), ExtractMin ( )	remove ( ), poll ( )
Element with maximum priority	Min ( )	peek ( )

Java's PriorityQueue is iterable, but the iteration order is not in order of priority. There is a contains( ) operation, but it is inefficient. There is no operation to remove arbitrary elements.

<u>Implementation of priority queues</u>: Binary heaps (usually stored in arrays).



Array implementation:

2	4	7	5	9	8	16	12
0	1	2	3	4	5	6	7

Root is stored at index 0.
For a node at index i:
  Parent's index: $(i - 1) / 2$.
  Index of Children: $\{2i+1,\ 2i+2\}$.

Heaps are binary trees in which each node stores one element, satisfying the following properties:

1. Order property: priority of node $\leq$ priority of its children, i.e., it has higher priority than its children,
2. Structure property: complete binary trees (all but last levels are full; last level is packed from left).

Operations will restore structure property first, and then ensure order property, using percolate Up/Down.

```
Heap occupies  pq [ 0···size−1].
Height of tree is log(size).
Running time of each operation: O(log(size)).

parent ( i ): return ( i −1 ) / 2

add ( x ):
   if size = pq.length then    // can resize pq here
      throw Exception "Priority queue is full"
   pq [ size ] ← x          把新加的放在最后然后向上调整
   percolateUp ( size )
   size++

percolateUp ( i ):
   x ← pq [ i ]
   while i > 0 and x < pq [ parent(i) ] do
      pq [ i ] ← pq [ parent(i) ]   把parent放下来
      i ← parent(i)                 更新i
   pq [ i ] ← x
```

```
remove ( ):
   min ← pq [ 0 ]
   pq [ 0 ] ← pq [ −−size ]     存好最顶上的值，然后把最后个值
   percolateDown ( 0 )          放在最顶上，然后向下调整
   return min

percolateDown ( i ):
   x ← pq [ i ]
   c = 2 ∗ i + 1    左孩子
   while c ≤size - 1 do
      if c < size - 1 and pq [ c ] > pq [ c + 1] then
         c++                c得到左右孩子中较小的那个
      if x ≤pq [ c ] then break     比左右孩子都小就调整完成
      pq [ i ] ← pq [ c ]   把孩子中较小的放到parent位置
      i ← c       更新i位置（当前调整的位置）
      c ← 2 ∗ i + 1     更新c位置（调整位置的孩子位置）
   pq [ i ] ← x

peek ( ):   return pq [ 0 ]
```

Minimum spanning trees (MST)

Input: Undirected, connected graph $G = (V, E)$, weights on edges $w : E \to \mathbb{Z}$.

Output: Spanning tree $T \subseteq E$, such that $w(T) = \sum_{e \in T} w(e)$ is a minimum among all spanning trees of G.

Prim's algorithm for finding MST:

从指定的顶点开始寻找最小权值的邻接点。图G=<V,E>，初始时S={V0}，把与V0相邻接，且边的权值最小的顶点加入到S。不断地把S中的顶点与V-S中顶点的最小权值边加入，直到所有顶点都已加入到S中。

Grow a tree starting at some node src as source.
S = Set of nodes connected by the tree. Initially, S = {src}.
while S $\neq$ V do
  Find a edge, $e = (u, v)$ of minimum weight, connecting some $u \in$ S with some $v \in$ V − S.
  Extend tree by adding edge e to tree. S ← S ∪ {v}.

```
Prim1( G, src ):  // Implementation #1 using a priority queue of edges
   for u ∈ V do  { u.seen ← false;   u.parent ← null }
   src.seen ← true;   wmst ← 0;   Create a priority queue q of edges
   for( Edge e: src ) { q.add(e) }
   while q is not empty do
      e ← q.remove( ).  Let e = (u, v), with u.seen = true.  找到weight最小的那条
      if v.seen then continue  // skip this edge   访问过了就跳过
      v.seen ← true;   v.parent ← u;   wmst ← wmst + w(e)   没访问过就访问(u,v)
      for( Edge e2: v ):
         if ! e2.otherEnd( v ).seen then q.add( e2 )   把新加入的顶点连着的没有访问过的顶点的边加入队列
   return // MST is implicitly stored by parent pointers
```

```
Prim2( G, src ):  // Implementation #2 using indexed priority queue of vertices
   // Node v ∈ V−S stores in v.d, the weight of a smallest edge that connects v to some u ∈ S
   for u ∈ V do  { u.seen ← false;   u.parent ← null;   u.d ← ∞ }
   src.d ← 0;   wmst ← 0
   q ← new priority queue of vertices with u.d as priority of u
   while q is not empty do
         u ← q.remove( );    u.seen ← true;   wmst ← wmst + u.d
         for( Edge e: u ) do
            v ← e.otherEnd( u )
            if ! v.seen and e.weight < v.d then
                  v.d ← e.weight;   v.parent ← u;
                  percolateUp( index of v in q )  // How do we find the index of v in q?
   return
```

```
class PrimVertex implements Comparator<PrimVertex>, Index {
   int d, index;
   public int compare( PrimVertex u, PrimVertex v ) {
         if ( u.d < v.d ) return -1; else if ( u.d == v.d ) return 0; else return 1;  }
   public void putIndex( int i ) { index = i; }
   public int getIndex( ) { return index; }
}
```