Building a PQ from an array of elements:

Input: $A[1..n]$, $A[0]$ not used.
Rearrange $A[1..n]$ so that it satisfies heap order.

$O(n)$ algorithm:

for $i \leftarrow n/2$ downto 1 do
    percolateDown(i)

Heap Sort Algorithm Input: $A[1..n]$, $A[0]$ unused.

1. Create a Max heap with $A[1..n]$
2. for $i \leftarrow n$ downto 2 do
    $x \leftarrow$ DeleteMin()
    $A[i] \leftarrow x$
3. A is in ascending order

Implementation

```
buildHeap()
for (i=n; i>0; i--) {
    Exchange A[1] <-> A[i]
    size--
    percolateDown(1)
}
```

Minimum spanning trees (MST)

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

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

Prim's Algorithm

Idea: Grow a tree starting from some node (source node)

S = set of nodes connected by the tree (Initially, $S = \{src\}$)

while $S \neq V$ do
    Find a minimum weight edge $e=(u,v)$ connecting S to some node in $V-S$. $u \in S$, $v \in V-S$
    Extend tree by adding this edge e to tree
    $S \leftarrow S \cup \{v\}$

Implementation #1 - use a priority queue of edges

```
for u ∈ V do { u.seen ← false ; u.parent ← null }
src.seen ← true          wmst ← 0
Q ← new priority queue of edges
  for edge e : src.Adj do { Q.add(e) }
while Q is not empty do
        e ← Q.remove()
        if e.from.seen and e.to.seen continue.
        u ← endpoint of e ∈ S   (u.seen = true)
        v ← endpoint of e ∉ S
        v.parent ← u
        wmst ← wmst + w(e)
        v.seen ← true
        for edge f ∈ v.Adj do { if [not (f.from.seen and f.to.seen)
                                  Q.add(f) }
                                w ← f.otherEnd(v)
                                if !w.seen { Q.add(f) }
```

Implementation #2 - Indexed Priority Queues

Node $v \in V - S$ stores in v.d, the weight of a smallest edge that connects v to some node in S.

```
for u ∈ V do { u.seen ← false; u.parent ← null; u.d ← ∞ }
src.d ← 0      wmst ← 0
Build a priority queue Q of vertices using u.d as priority of u
(Q has all vertices of V at the beginning)
while Q is not empty do
        u ← Q.remove()
        u.seen ← true      wmst += u.d
        for edge e ∈ u.Adj; do
                v = e.otherEnd(u)
                if (!v.seen and w(e) < v.d) then
                        v.d ← w(e) ; v.parent ← u
                        percolateUp (index of v in priority queue Q)
                                    ↑
                                    How do we track this?
```