2 Abstract Data Type: Priority Queue

Readings: CLRS 6

Definition 1. *A* Min priority queue *is an abstract data type supporting two operations:*

- **insert(x):** *Inserts element x into the data structure.*
- **extractMin:** *Returns the smallest element and removes it from the data structure.*

Similarly, we can define a max priority queue*, which supports an extractMax operation instead of the extractMin operation.*

Note: All the results for min priority queues also hold for max priority queues.

Example: Sorting by using priority queues (we will call this "priority queue sorting"):

```
create an empty min priority queue PQ
for i:=1 to n
  PQ.insert(A[i])
for i:=1 to n
  A[i]:=PQ.extractMin
```

Trivial implementations of priority queue:

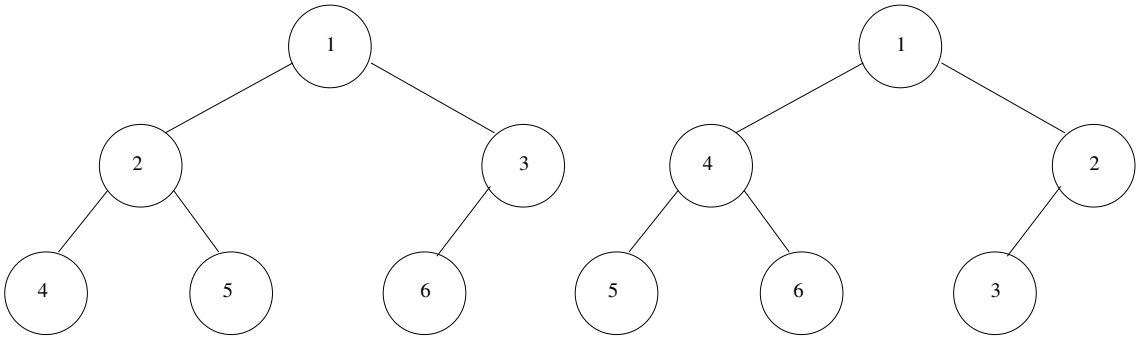
- **Unsorted array.** insert $O(1)$, extractMax $O(n)$
  priority queue sort becomes selection sort
- **Sorted array.** insert $O(n)$, extractMax $O(1)$
  priority queue sort becomes insertion sort

2.1 Heaps

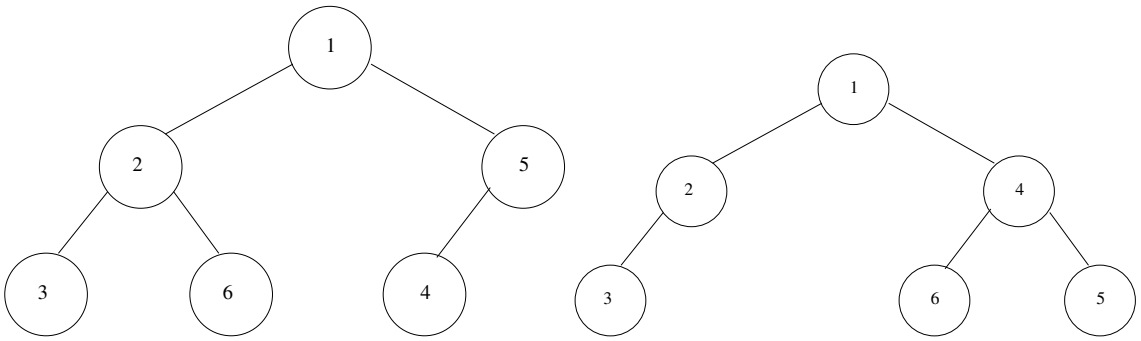
A *heap* is a data structure that implements priority queue using a binary tree to store the elements in the data structure. A valid heap must satisfy the following three properties:

1. **Heap property:** The children of every node x must have a value larger than or equal to x.
   (Intuitively: heavier elements sink to the bottom.)
2. All levels of the binary tree except for the last level are completely filled (i.e., level h has 2^h elements).
3. The last level is always filled from left to right. In particular, any node is either a *leaf of the tree* (has no children), has *both children*, or has *only a left child*.

Examples: The following structures are valid heaps representing the set of numbers $\{1, 2, 3, 4, 5, 6\}$:



The following structures violate one of the required heap properties and therefore are not valid heaps:



Inserting element into heap.

- Add the element to the left-most empty position at the bottom level (or start a new level if the bottom level is full)
- This may lead to violation of the heap property
- "Sift the element up" to restore the heap property (see SiftUp below)

```
SiftUp(node):
  if (node does not have a parent) done!!
  if (node.parent.value>node.value)
  | swap(node.value,node.parent.value);
  | SiftUp(node.parent);
```

Extracting the minimum element from the heap.

- Finding the minimum is easy—it is stored in the root of the tree
- Remove the minimum
- Fill the gap with the right-most element at the bottom level
- Restore the heap property by "sinking" the element down (see Heapify below)

```
Heapify(node):
  // pre: node.left, node.right are roots
  //      of valid heaps or null
  // post: node is a root of a valid heap
  if not(node.left) and not(node.right) done!!
  else
  | if node.right and node.right.value<node.left.value
  | | k:=node.right
  | else
  | | k:=node.left
  |
  | if node.value<=k.value done!!
  | else
  | | swap(node.value,k.value);
  | | Heapify(k);
```

How fast are these operations? Running time of both Heapify and SiftUp is in the worst case proportional to the height of the tree representing the heap.

Definition 2 (Height of a tree)**.** *Height of a tree is the number of edges on the longest path from the root of the tree to a leaf of the tree.*

Theorem 1. *A heap of height h has at least 2^h and at most $2^{h+1} - 1$ elements.*

Proof. At level i of the heap (except for the bottom-most level) we have exactly 2^i elements. At the bottom level there are x elements, where x is at least 1 and at most 2^h. Therefore the total number of elements n in the heap is:

$$n = \left(\sum_{i=0}^{h-1} 2^i \right) + x = 2^h - 1 + x$$

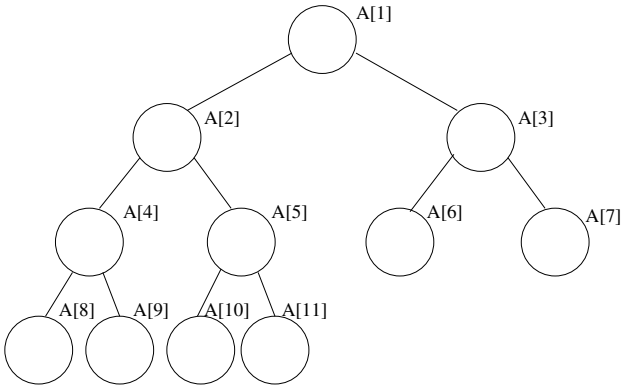
and therefore $2^h \leq n \leq 2^{h+1} - 1$. □

Corollary 1. *A heap has height $\Theta(\log n)$, where n is the number of elements in the heap.*

Thus the running time of both Heapify and SiftUp is $\Theta(\log n)$. If we use this implementation of the priority queue in conjunction with the priority queue sort, we obtain $\Theta(n \log n)$ sorting algorithm.

Implementing heaps. There are two things that we can do to simplify the implementation of heaps.

1. **Implement heaps in an array.** Store root in $A[1]$ and continue with elements level-by-level from top to bottom, in each level left-to-right:



   Now we can implement necessary operations within the heap as follows:

node:	position in the array
node.value:	A[node]
node.parent:	$\lfloor$node/2$\rfloor$
node.left:	$2 *$ node
node.right:	$2 *$ node $+ 1$

2. **Remove the recursion.** The SiftUp and Heapify functions are recursive, but there is always only a single recursive call that is made at the end of the function. This is called *tail recursion* and it can be always replaced with a while loop.

```
SiftUp(node):
  while (node>1 and A[node]<A[node/2]) {
  | swap(A[node],A[node/2]);
  | node:=node/2

Heapify(node,size):
  // pre: node.left (2*node), node.right (2*node+1) are roots
  //      of valid heaps or out of bounds
  // post: node is a root of a valid heap

  while (2*node<=size and A[node]>A[2*node])
     or (2*node+1<=size and A[node]>A[2*node+1])
  | if 2*node+1>size or A[2*node]<A[2*node+1]
  | | k:=2*node
  | else
  | | k:=2*node+1
  | swap(A[node],A[k]);
  | node:=k
```

Building heaps bottom up. If we are given an array $A[1 \dots n]$ of elements and we want to build a heap in place (i.e., without auxiliary storage), we can use the following procedure:

```
BuildHeap(n):
  // build heap from values stored in A[1..n]
  for i:=n/2 downto 1
    // inv: elements i+1,...,n are roots of valid heaps
    Heapify(i);
    // inv: elements i,i+1,...,n are roots of valid heaps
```

Theorem 2. *BuildHeap constructs a valid heap.*

Proof. We prove the claim by showing that the following invariant holds: At the beginning of iteration i, elements $i+1, i+2, \dots, n$ are roots of valid heaps.

We show that the invariant holds by induction on value of i.

- **Base case:** If $i = \lfloor n/2 \rfloor$, elements $i+1, i+2, \dots n$ do not have any children. Since a singleton is always a valid heap, they are all roots of valid heaps.
- **Induction step:** Because of the invariant, children $2*i$ and $2*i+1$ of node i are already roots of valid heaps. Therefore the precondition of calling Heapify(i) is satisfied, and according to the postcondition of Heapify, node i will be therefore a root of a valid heap after the iteration is finished.

Therefore when i gets to 0 (after the last iteration is finished), $A[1]$ is a root of a valid heap, which is what we wanted to prove. □

Even though this routine does not look faster than inserting elements into the priority queue one-by-one (which takes $O(n \log n)$ time), we can show that the overall running time of the BuildHeap is $O(n)$.

Theorem 3. *BuildHeap runs in $O(n)$ time.*

Proof. Let the *depth of a node in a tree* be the number of edges on the path from the root to that node. Thus depth of the root node is 0, and the largest depth of an element in the heap is h, where h is the height of the heap.

Note that for a node at a particular depth k, Heapify operations takes at most $h-k$ moves. This is because at every step, the current node in Heapify moves down one level, and there are only $h-k$ levels to go.

For simplicity, let us assume that the last level of the heap is completely full (the proof extends directly also to the heaps that do not have the last level full). The total number of elements in such heap is $n = 2^{h+1} - 1$, where h is the height of the heap. Since at depth k there are 2^k nodes, the total number of moves in BuildHeap can be summarized as follows:

$$\begin{aligned}\sum_{k=0}^{h-1} 2^k(h-k) &= [m := h-k] \\ &= \sum_{m=1}^{h} m \cdot 2^{h-m} = 2^h \sum_{m=0}^{h} m(1/2)^m \\ &\leq 2^h \sum_{m=0}^{\infty} m(1/2)^m,\end{aligned}$$

and since $\sum_{i=0}^{\infty} i \cdot c^i = c/(1-c)^2$ (see TCSCS; the formula can be applied here for $c = 1/2$), the total number of moves in BuildHeap is $2^h \cdot 2 = 2^{h+1} = n+1$, and therefore the running time of BuildHeap is $\Theta(n)$. □

HeapSort and MaxHeaps. Heap sort is the priority queue sort with heaps. However, we do several modifications to achieve better implementation in place (i.e., the algorithm does not use auxiliary space):

- We will not use separate space for storing the priority queue and the result. Instead, in step i of the algorithm, the first $n-i$ elements of array A will represent the heap, and the last i elements will represent the partial result of sorting.
- So far we talked about MinHeaps (heaps that support operation ExtractMin). In this implementation we will use MaxHeaps (heaps that support ExtractMax instead of ExtractMin). MaxHeaps are implemented in the same way as MinHeaps, but we need to change the direction of the comparisons.

```
HeapSort:
  for i:=n/2 downto 1
  | MaxHeapify(i,n)
  for i:=n downto 1
  | Swap(A[1],A[i]);
  | MaxHeapify(1,i-1);

MaxHeapify(node,size):
  while (2*node<=size and A[node]<A[2*node])
     or (2*node+1<=size and A[node]<A[2*node+1])
  | if 2*node+1>size or A[2*node]>A[2*node+1]
  | | k:=2*node
  | else
  | | k:=2*node+1
  | swap(A[node],A[k]); node:=k
```

2.2 Summary

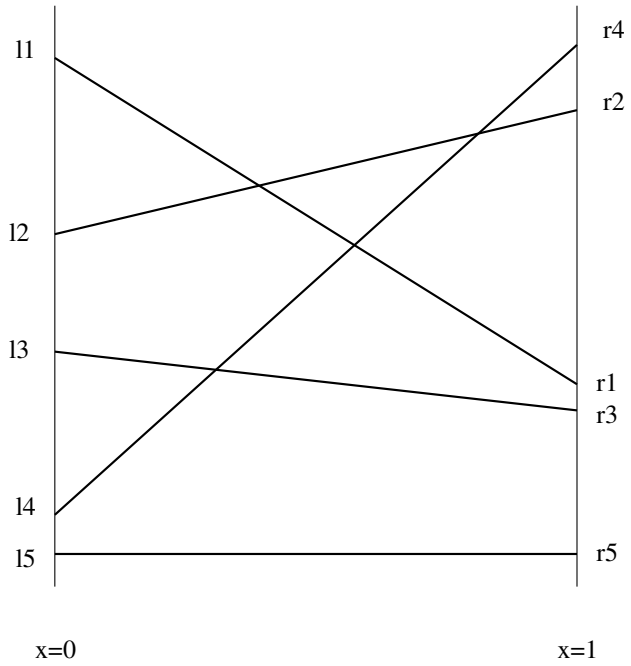
In this section we introduced abstract data type priority queues with several implementations:

Implementation	Insert	ExtractMin	PQ sort	Note
Unsorted array	$\Theta(1)$	$\Theta(n)$	Selection sort $\Theta(n^2)$	
Sorted array	$\Theta(n)$	$\Theta(1)$	Insertion sort $\Theta(n^2)$	
Counters	$\Theta(1)$	$\Theta(1)$	Counting sort $\Theta(n)$	small domain (e.g. $[1, 1000]$)
Heaps	$\Theta(\log n)$	$\Theta(\log n)$	Heap sort $\Theta(n \log n)$	

2.3 Other applications of priority queues

(Solutions of some of these exercises were presented in class.)

Intersections of line segments: We are given n line segments, where the i-th line segment has end coordinates $(0, \ell_i)$, $(1, r_i)$.



List all intersections of these line segments.

- Trivial solution: $O(n^2)$
- Solution using priority queues: $O(n \log n + k \log n)$, where k is the number of intersections.

Real-time event simulation: Consider n balls on a pool table moving, each in a different direction and at a different speed. Given the speeds and directions of all balls, compute the position of all balls at time t.

- The running time can be improved by storing all the important time points (e.g. two balls collide, ball collides with the boundary) in the priority queue.