

## Priority Queues

Reading: *CLRS* Chapter 6/*CLR* Chapter 7

---

### Priority Queue Contract

Given a set of elements with a priority ordering (which we will denote by  $>$ ), A **mutable priority queue** is a collection of such elements supporting the following key operations:

**PQ-Empty()**

Returns an empty priority queue.

**PQ-Insert( $P, elt$ )**

Modifies  $P$  by inserting  $elt$  into priority queue  $P$ .

**PQ-Delete-Max( $P$ )**

Deletes from  $P$  and returns the largest element (by the priority ordering) of priority queue  $P$ .

**Array-To-PQ( $A$ )**

Constructs and returns a priority queue containing the  $n$  elements of array  $A$ . May mutate  $A$ .

---

### Priority Queue Implementations

What are the running times of the priority queue operations for an  $n$ -element priority queue  $P$  using the following representations?

	PQ-Insert( $P, elt$ )	PQ-Delete-Max( $P$ )	Array-To-PQ( $A$ )
unsorted list			
list sorted high to low			
binary search tree			
heap (this lecture)			

---

### Heap Sort

Given heap operations with the above running times, it's easy to construct a guaranteed  $O(n \lg(n))$  sorting algorithm:

**HeapSort( $A$ )**

$H \leftarrow \text{Array-To-PQ}(A)$

**for**  $i \leftarrow \text{length}[A]$  **downto** 1 **do**

$A[i] \leftarrow \text{PQ-Delete-Max}(H)$

We will see below that when priority queues are represented as heaps, the priority queue used by **HeapSort** can be stored within the argument array  $A$ , so that **HeapSort** can be an in-place sorting algorithm.

---

## Complete Binary Trees

The **binary address** of a node in a binary tree specifies the order in which it would be visited in a breadth first traversal. For example:

Operations on binary addresses:

- $\text{Left}(\text{address}) = 2 * \text{address}$
- $\text{Right}(\text{address}) = (2 * \text{address}) + 1$
- $\text{Parent}(\text{address}) = \text{address} \text{ div } 2$

An  $n$ -element binary tree is **complete** if the set of binary addresses of its nodes is  $\{1, 2, \dots, n\}$ . For example:

An  $n$ -element binary tree is **full** if it is a complete tree of height  $h$  with  $2^h - 1$  nodes.

*Important implementation detail:* The elements of a complete binary tree can be represented as an array. In this representation, tree navigation is performed by pointer arithmetic.

---

## Heaps

A **heap** is a complete binary tree satisfying the following heap condition:

At every node in a heap, the node value is greater than or equal to all the values in both of its subtrees.

*Example:*

---

## Representing Priority Queues as Heaps

We can represent a priority queue as a record (object) with two slots:

1. a `size` slot holding the number of elements in the priority queue
2. an `elts` slot holding an array representing the heap of elements in the priority queue.

*Example:*

---

## Heap Insertion

```
PQ-Insert(H, elt)
  size[H] ← size[H] + 1
  A ← elts[H]
  A[size[H]] ← elt ▷ Assume A is big enough to hold new element.
                  ▷ In practice, might need to increase size of array.
  Bubble-Up(A, size[H])
```

```
Bubble-Up(A, address)
  while address > 1 and lt(A[Parent(address)], A[address]) do
    swap(A, address, Parent(address))
    ▷ Can get by with fewer assignments; See CLR
    address ← Parent(address)
```

*Example:*

*Analysis:*

---

## Heap Deletion

```
PQ-Delete-Max(H)
  if size[H] < 1 then
    error "heap underflow"
  A ← elts[H]
  max ← A[1]
  A[1] ← A[size[H]]
  size[H] ← size[H] - 1
  BubbleDown(A, 1)
  return max
```

```
Bubble-Down(A, address)
  ▷ This function is called Heapify in CLR
  if Left(address) ≤ heap_size[A]
    and lt(A[address], A[Left(address)]) then
    largest ← Left(address)
  else
    largest ← address
  if Right(address) ≤ heap_size[A]
    and lt(A[largest], A[Right(address)]) then
    largest ← Right(address)
  if largest ≠ address then
    swap(A, address, largest)
    Bubble-Down(A, largest)
```

*Analysis:*

---

## Constructing a Heap

*Naive version of Array-To-PQ:*

```
Array-To-PQ(A)
  H ← PQ-Empty
  for i ← 1 to length[A] do
    ▷ Uses array slots for heap storage!
    PQ-Insert(H, A[i])
  return H
```

*Analysis:*

*Clever version of Array-To-PQ:*

```
Array-To-PQ(A)
  H ← PQ-Empty
  size[H] ← length[A]
  for i ← (length[A] div 2) downto 1 do
    Bubble-Down(A, i)
  return H
```

*Analysis:*