Abstract Data Types

**Abstract Data Type (ADT):** A description of *information* and a collection of *operations* on that information.

The information is accessed *only* through the operations.

We can have various *realizations* of an ADT, which specify:

- How the information is stored (*data structure*)
- How the operations are performed (*algorithms*)
Priority Queue ADT

**Priority Queue:** An ADT consisting of a collection of items (each having a *priority*) with operations

- *insert:* inserting an item tagged with a priority
- *deleteMax:* removing the item of *highest priority*

`deleteMax` is also called *extractMax*.

Applications: typical “todo” list, simulation systems

The above definition is for a *maximum-oriented* priority queue. A *minimum-oriented* priority queue is defined in the natural way, by replacing the operation *deleteMax* by *deleteMin*.

Realizations of Priority Queues

**Attempt 1:** Use *unsorted arrays*

- *insert:* $O(1)$
- *deleteMax:* $O(n)$

Using unsorted linked lists is identical.

**Attempt 2:** Use *sorted arrays*

- *insert:* $O(n)$
- *deleteMax:* $O(1)$

Using sorted linked-lists is identical.
Heaps

A max-heap is a binary tree with the following two properties:

1. Structural Property: All the levels of a heap are completely filled, except (possibly) for the last level. The filled items in the last level are left-justified.

2. Heap-order Property: For any node $i$, key (priority) of parent of $i$ is larger than or equal to key of $i$.

A min-heap is the same, but with opposite order property.

Lemma: Height of a heap with $n$ nodes is $\Theta(\log n)$. 
Insertion in Heaps

- Place the new key at the first free leaf
- The heap-order property might be violated: perform a bubble-up:

```
bubble-up(v)
v: a node of the heap
1. while parent(v) exists and key(parent(v)) < key(v) do
2. swap v and parent(v)
3. v ← parent(v)
```

The new item bubbles up until it reaches its correct place in the heap.

Time: $O(\text{height of heap}) = O(\log n)$.

deleteMax in Heaps

- The maximum item of a heap is just the root node.
- We replace root by the last leaf (last leaf is taken out).
- The heap-order property might be violated: perform a bubble-down:

```
bubble-down(v)
v: a node of the heap
1. while v is not a leaf do
2. u ← child of v with largest key
3. if key(u) > key(v) then
4. swap v and u
5. v ← u
6. else
7. break
```

Time: $O(\text{height of heap}) = O(\log n)$.
Priority Queue Realization Using Heaps

**heapInsert(A, x)**

A: an array-based heap, x: a new item

1. \(\text{size}(A) \leftarrow \text{size}(A) + 1\)
2. \(A[\text{size}(A) - 1] \leftarrow x\)
3. \(\text{bubble-up}(A, \text{size}(A) - 1)\)

**heapDeleteMax(A)**

A: an array-based heap

1. \(\text{max} \leftarrow A[0]\)
2. \(\text{swap}(A[0], A[\text{size}(A) - 1])\)
3. \(\text{size}(A) \leftarrow \text{size}(A) - 1\)
4. \(\text{bubble-down}(A, 0)\)
5. \(\text{return} \ max\)

Insert and deleteMax: \(O(\log n)\)

Storing Heaps in Arrays

Let \(H\) be a heap (binary tree) of \(n\) items and let \(A\) be an array of size \(n\). Store root in \(A[0]\) and continue with elements level-by-level from top to bottom, in each level left-to-right.

It is easy to find parents and children using this array representation:

- the *left child* of \(A[i]\) (if it exists) is \(A[2i + 1]\),
- the *right child* of \(A[i]\) (if it exists) is \(A[2i + 2]\),
- the *parent* of \(A[i]\) \((i \neq 0)\) is \(A[\lfloor \frac{i-1}{2} \rfloor]\) \((A[0]\) is the root node).
## Building Heaps

**Problem statement:** Given $n$ items (in $A[0 \cdots n-1]$) build a heap containing all of them.

**Solution 1:** Start with an empty heap and insert items one at a time:

```
heapify1(A)
A: an array
1. initialize $H$ as an empty heap
2. for $i \leftarrow 0$ to $\text{size}(A) - 1$ do
3. heapInsert($H, A[i]$)
```

This corresponds to going from $0 \cdots n-1$ in $A$ and doing *bubble-ups*.

Worst-case running time: $\Theta(n \log n)$.

---

## Building Heaps

**Problem statement:** Given $n$ items (in $A[0 \cdots n-1]$) build a heap containing all of them.

**Solution 2:** Using *bubble-downs* instead:

```
heapify(A)
A: an array
1. $n \leftarrow \text{size}(A) - 1$
2. for $i \leftarrow \lfloor n/2 \rfloor$ downto 0 do
3. bubble-down($A, i$)
```

A careful analysis yields a worst-case complexity of $\Theta(n)$.

A heap can be built in linear time.
Using a Priority Queue to Sort

\[ PQ - Sort(A) \]
1. initialize \( PQ \) to an empty priority queue
2. for \( i \leftarrow 0 \) to \( n - 1 \) do
3. \( PQ.insert(A[i], A[i]) \)
4. for \( i \leftarrow 0 \) to \( n - 1 \) do
5. \( A[n - 1 - i] \leftarrow PQ.deleteMax() \)

HeapSort

\[ HeapSort(A) \]
1. initialize \( H \) to an empty heap
2. for \( i \leftarrow 0 \) to \( n - 1 \) do
3. \( heapInsert(H, A[i]) \)
4. for \( i \leftarrow 0 \) to \( n - 1 \) do
5. \( A[n - 1 - i] \leftarrow heapDeleteMax(H) \)

\[ HeapSort(A) \]
1. \( heapify(A) \)
2. for \( i \leftarrow 0 \) to \( n - 1 \) do
3. \( A[n - 1 - i] \leftarrow heapDeleteMax(A) \)

Running time of HeapSort: \( O(n \log n) \)
Selection

**Problem Statement:** The \( k \)th-max problem asks to find the \( k \)th largest item in an array \( A \) of \( n \) numbers.

**Solution 1:** Make \( k \) passes through the array, deleting the maximum number each time.

**Complexity:** \( \Theta(kn) \).

**Solution 2:** First sort the numbers. Then return the \( k \)th largest number.

**Complexity:** \( \Theta(n \log n) \).

**Solution 3:** Scan the array and maintain the \( k \) largest numbers seen so far in a min-heap

**Complexity:** \( \Theta(n \log k) \).

**Solution 4:** Make a max-heap by calling `heapify(A)`. Call `deleteMax(A)` \( k \) times.

**Complexity:** \( \Theta(n + k \log n) \).