Heaps

Binary heaps :

Heap Storage Rules

If the elements of a set can be compared with a *total order semantics*, then these elements can be stored in a heap. A heap is a binary tree in which two rules are followed :

1. - In a *max-heap*, the element contained by a node is greater than or equal to the elements of that node's children.

   - In a *min-heap*, the element contained by a node is lesser than or equal to the elements of that node's children.

2. The tree is a complete binary tree: thus every level, except the deepest, must contain as many nodes as possible; and at the deepest level, all nodes are as far left as possible.

Representation :

Although heaps can be represented in exactly the same manner as the *binary search tree* using dynamic memory structure, we can take advantage of the fact that they are almost *complete* trees to represent them using an array.

An array A that represents a heap is an object with two attributes: $length[A]$, which is the number of elements in the array, and *heap-size*[A], the number of elements in the heap stored within array A. That is, although $A[1 \ldots length[A]]$ may contain valid numbers , no element past A[*heap-size*[A]], where *heap-size*[A] $\leq length[A]$, is an element of the heap.

The root of the tree is A[1], and given the index i of a node, the indices of its $PARENT(i)$, left child $LEFT(i)$ and right child $RIGHT(i)$ can be computed simply:

```
function PARENT(i)
    return ⌊i/2⌋

function LEFT(i)
    return 2i

function RIGHT(i)
    return 2i + 1
```

The heap property #1 is therefore restated as :

$A[PARENT(i)] \geq A[i]$ in a *max-heap*

or

$A[PARENT(i)] \leq A[i]$ in a *min-heap*

The five basic procedures on maximal heaps are :

- $Heapify$: The procedure which maintains the *heap* property, runs in $O(\lg n)$.
- *Build-Heap*: Creates a heap from an unordered input array in linear time.
- $Heapsort$: Sorts an array in $O(n \lg n)$.
- *Extract-Max* and $Insert$: allows, in $O(\lg n)$, to use a heap as a priority queue.

Note that a minimal heap would use *Extract-Min* instead of *Extract-Max*.

In the following transparencies we will consider only the operations for a maximal heap.

Heapify:

Heapify is used to insure the heap property of the tree after a manipulation. It takes as parameter an array A and an index i into the array. *Heapify* assumes that the trees $LEFT(i)$ and $RIGHT(i)$ are heaps, but that $A[i]$ may be smaller than its children, thus violating the heap property. The function of this algorithm is therefore to let the value at $A[i]$ "flow down" in the heap so that the subtree rooted at index i becomes a heap.

```
function HEAPIFY(A, i)
    l ← LEFT(i)
    r ← RIGHT(i)
    if l ≤ heap-size[A] and A[l] > A[i]
        then largest ← l
        else largest ← i
    if r ≤ heap-size[A] and A[r] > A[largest]
        then largest ← r
    if largest ≠ i
        then
            exchangeA[i] ↔ A[largest]
            HEAPIFY(A, largest)
```

Since in the worst case the value at $A[i]$ will drift down the entire tree, the running time of the algorithm is in $O(h)$ where h is the height of the tree. Since h is $\lg n$, the algorithm runs in $O(\lg n)$.

Building a *heap* :

Having just defined the procedure $HEAPIFY$, we can use it to convert an array $A[1 \ldots n]$, where $n = length[A]$, into a *heap*.

- Since the elements in the subarray $A[(\lfloor n/2 \rfloor + 1) \ldots n]$ are all leaves of the tree, each acts as a 1-element *heap*.

- Starting from the parents of these leaves and going all the way back to the root, apply the $HEAPIFY$ algorithm.

```
function BUILD-HEAP(A)
    heap-size[A] ← length[A]
    for i ← ⌊length[A]/2⌋ downto 1 do
        HEAPIFY(A, i)
```

Although we know that there are n elements and that $HEAPIFY$ runs in $O(\lg n)$, we can deduce a closer bound then $O(n \lg n)$.

We know that the running time of $HEAPIFY$ is $O(h)$ where h is the height of the tree. However, $HEAPIFY$ is being executed on sub-trees of A, the height of which are often much smaller then $\lg n$.

When looking at a complete tree, we realise that at most $\lceil n/2^{h+1} \rceil$ nodes can be of height h. The running time of the *BUILD-HEAP* algorithm can therefore be expressed as :

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = 0 \left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

however

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$$

thus

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = 0 \left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = 0 \left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

Hence, we can build a heap from an unordered array in linear time.

Heapsort algorithm :

The *heapsort* algorithm starts by using *BUILD-HEAP* to transform the array into a heap, it then takes the biggest element (the root), outputs it, swaps it with the element at position *heap-size*(A), decrements by one this value (thus eliminating the biggest value), and applies $HEAPIFY$ to the tree. By repeating this process n times, the running time of the algorithm is $\in O(n \lg n)$.

```
function HEAPSORT(A)
    BUILDHEAP(A)
    for i ← length[A] downto 2 do
        output(A[1])
        exchange A[1] ↔ A[i]
        heapsize[A] ← heapsize[A] − 1
        HEAPIFY(A, 1)
```

Priority queues:

A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a *key*. A priority queue supports the following operations:

- $INSERT(S, x)$ inserts the element x into the set S. This operation could be written as $S \leftarrow S \cup \{x\}$.
- $MAXIMUM(S)$ returns the element of S with the largest key.
- *EXTRACT-MAX*(S) removes and returns the element of S with the largest key.

Priority queues are often used in job scheduling on shared computers. The priority queue keeps track of the jobs to be performed and their relative priorities. When a job is finished or interrupted, the highest-priority job is selected from those pending using *EXTRACT-MAX*. A new job can be added to the queue at any moment using $INSERT$.

Priority queues can easily be implemented using *heaps*. The functions *HEAP-MAXIMUM* returns the maximum heap element in $\theta(1)$ by returning $A[1]$.

```
function HEAP-EXTRACT-MAX(A)
    if heap-size[A] < 1 then
        error "heap underflow"
    max ← A[1]
    A[1] ← A[heap-size[A]]
    heap-size[A] ← heap-size[A] − 1
    HEAPIFY(A, 1)
    return max
```

The running time of *HEAP-EXTRACT-MAX* is $\in O(\lg n)$ since it performs a constant number of steps on top of the call to $HEAPIFY$.

```
function HEAP-INSERT(A, key)
    heapsize[A] ← heapsize[A] + 1
    i ← heapsize[A]
    while i > 1 and A[PARENT(i)] < key do
        A[i] ← A[PARENT(i)]
        i ← PARENT(i)
    A[i] ← key
```

Since the path traced from the new leaf to the root is of length $O(\lg n)$, this algorithm is also $\in O(\lg n)$.