CHAPTER - 7

PRIORITY QUEUES AND HEAPS

A priority queue is a datastructure that supports the operations Insert and DeleteMin (return & remove minimum element) or DeleteMax (return & remove maximum element)

Main PQ Operations

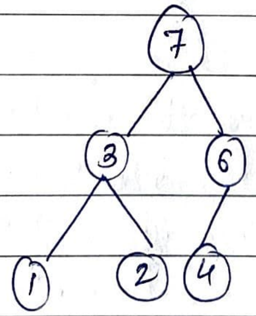
Insert (key, data) : Elements are ordered based on key.
DeleteMin / DeleteMax : Remove & return minimum/maximum element
GetMin / GetMax : Return minimum/maximum element.

Auxillary PQ Operations

k^{th} largest / k^{th} smallest
Size : Returns the number of elements in PQ
Heap Sort : Sorts PQ element based on (key).

Heaps & Binary Heaps

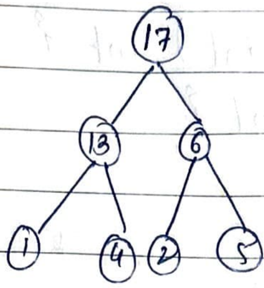
Heap : A tree with below properties
- Value of a node must be ($\geq$) or ($\leq$) to its children.
- It should form a complete binary tree.



Heap Types:

Min heap : Value of node ($\leq$) to its children
Max heap : " " " ($\geq$) " "

Binary Heaps (Each node may have upto two children)



~~17 6~~

17	13	6	1	4	2	5
0	1	2	3	4	5	6

Representation:

```
type struct Heap {
array []int
count  int   // No. of elements in Heap
capacity int // Size of the heap
heapType int // Heap type min/max
}
```

Creating a Heap

```
func Create(capacity, heapType int) *Heap {
    h := 4Heap{}
    h.array = make([]int, capacity)
    h.count = 0
    h.capacity = capacity
    h.heapType = heapType    // min = 0, max = 1
    return h
}
```

Parent of a Node: It is at location $\frac{i-1}{2}$ in the array. For example ⇒ 13 is at location 1 i.e. i = 1

$\therefore$ Parent of $13 = \frac{i-1}{2} \Rightarrow \frac{1-1}{2} \Rightarrow 0$

$\Rightarrow$ array[0] $\Rightarrow$ 17

We cannot find the parent for i = 0 & i >= h.count then.

```
func Parent (h *Heap, i int) int {
    if i <= 0 || i >= h.count {
        return -1
    }
    return (i-1/2)
}
```

Children of a Node

For a node at i^{th} position, its children are at $2*i+1$ & $2*i+2$ locations.

13 is at position => 1

Its children are at

↓

2 * i + 1 => 2 * 1 + 1 => 3 (Left Child)

2 * i + 2 => 2 * 1 + 2 => 4 (Right child)

h.array [3] = 1

h.array [4] = 4

```
func LeftChild (h *Heap, i int) int {
    left := 2 * i + 1
    if left >= h.count {
        return -1
    }
    return left
}

func RightChild (h *Heap, i int) int {
    right := 2 * i + 2
    if right >= h.count {
        return -1
    }
    return right
}
```

Maximum Element : In max heap, the max element is always at the root.

```
func Maximum (h *Heap, i) int {
        if h.count ==0 {
            return -1
        }
        return h.array [0]
}
```

Heapifying an Element

1. To heapify an element, find the maximum of its children & swap with the element.
2. Continue the above process until, the element satisfies the heap properties.

Algorithm

1. Accept heap & i th position from where heapifying should start.
2. Get the left & right child of i.
3. If left child exists & it is greater than the value at i then

    max = left child

    else

    max = i
4. If right child exists & it is greater than the value at max then

    max = r
5. If max is not equal to i, swap element at i with element at max.

6. Call PercolateDown(h, max)
7. DONE

```
func PercolateDown (h *Heap, i int) {
    l ~~LeftChild~~ := LeftChild (h, i)
    r := RightChild (h, i)
    max := 0
    // ~~When to~~ Algorithm Step 3
    if l != -1 && h.array[l] > h.array[i] {
        max = l
    } else {
        max = i
    }
    // Algorithm Step 4
    if r != -1 && h.array[r] > h.array[max] {
        max = r
    }
    // Algo. Step 5
    if max != i {
        h.array[i], h.array[max] = h.array[max], h.array[i]
    }
    PercolateDown (h, max)
}
```

Time Complexity : $O(\log n)$

Deleting an Element

Algorithm:

1. Delete the first (root) element i.e. replace it with heap's last element.
2. Delete last element
3. Change count variable
4. Heapify the element ⇒ PercolateDown (h, 0)
5. Done

```
func Delete (h *Heap) int {
if h.count==0        max := h.array [0]
return -1            h.array[0] := h.array [count -1]
                     h.array = h.array [:count -1]
                     h.count -= 1
                     PercolateDown (h, 0)
                     return max
}
```

Time Complexity: O (log n)

Inserting an Element

Algorithm :

1. Insert element at last index.
2. Start heapifying from last index.
3. Increase heap capacity before insert.

```
func Resize (h *Heap) {
    oldArray := h.array
    newCapacity := 2 * h.capacity
    h.array := make ([] int, newCapacity)

    for i,v := range oldArray {
        h.array [i] = ~~oldArray~~ v
    }
}
```

```
func Insert (h *Heap, data int) {
    if h.count == h.capacity {
        Resize (h)
    }
    h.count++
    i = h.count -1
```

// Percolate Up
~~while (i~~ for i >= 0 && data > h.array[i-1/2] {
    h.array[i] = h.array[i-1/2]
    i = i-1/2
}
h.array[i] = data

}