Priority Queue

A priority queue is a genernalisation of a queue where each element is assigneed a priority and elements come out in order by priority

Operations:

insert(val): this function will insert an element into the priority queue

extractMax(): this function will get the element with the highest priority out of the queue

remove(it): removes an elemnet pointed by an iterator

getMax(): returns an element with the maximum priority without extracting it out of the queue

change_Priority(it,p): changes the priority of an element pointed by 'it' to 'p'\

Implementations:

There are many ways to implement this data structure.

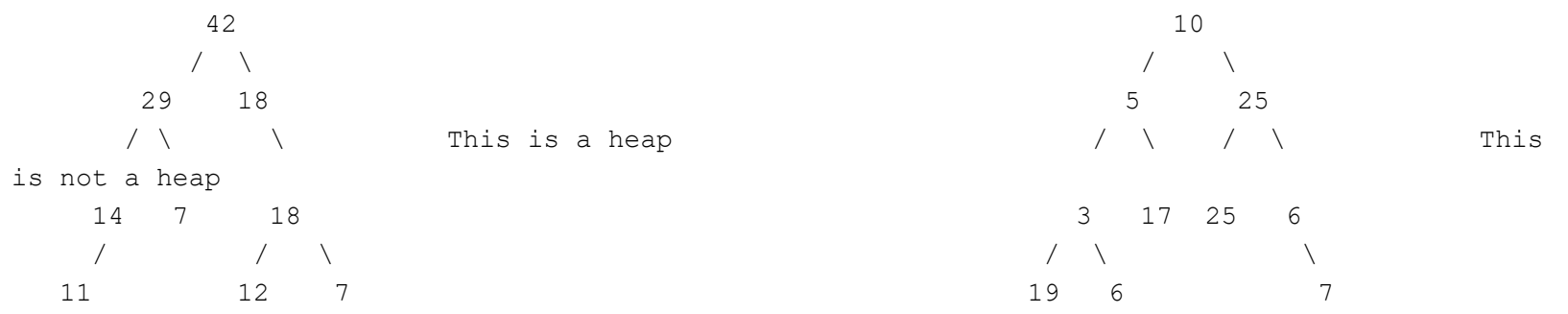
The naive way to do it is by using unsorted array or list to implement it. The insert() operation will take O(1) time, however it is not sufficient because the operations extractMax() will take O(n) times complexity.

The other way is to use sorted array, then extractMax() will take O(1) time complexity, but insert(val) will take possibly O(n) time. The insert(val) operation will first find a position for val (O(logn)) by using binary search. Then shift all elements to the right of it by 1 (O(n)). If we use sorted list the result will be the same

Optimal solution for implement priority queue is by Binary Heap

Binary max-heap:

is a binary tree (each node has zero, one or two children) where the value of each node is at least the values of its children. Max element is at the root and children's value cannot be greater than its parent.



Baisc operations and Pseudocode:

maxSize: is the maximum number of element in the heap size: is the size of the Heap H[1...maxSize] is an array of length maxSize where the heap occupies the first element

siftUp(i) This function will swap the problematic node with its parent until the property is satisfied

```
while i >1 and H[parent(i)] < H[i]:

  swap H[parent(i)] and H[i]

  i <- parent(i)
```

siftDown(i) This function will swap the problematic node with larger child until the heap property is satisfied

```
maxIndex <- implement

l <- Leftchild(i)

if l <= size and H[l] > H[maxIndex]:

  maxIndex <- l

r <- Rightchild(i)

if r <= size and H[r] > H[maxIndex]:

  maxIndex < -r

if i != maxIndex:

  swap H[i] and H[maxIndex]

  siftDown(maxIndex)
```

insert(val) This function will insert a value to a heap without changing it properties

```
if size = maxSize:

  return Error

size <- size +1

H[size] <-p

siftUp(size)
```

extractMax() This function will return the value with maximum priority

```
result <- H[0] # root value

H[1] <- H[size]

size <- size -1

siftDown(1)

return result
```

remove(i) Remove a specific element without the chaning the heap properties

```
H[i] <- oo  # set it value to infinity

siftUp(i)

extractMax() # remove the max value
```

change_Priority(i,p) Change the priority of a node

```
old_p = H[i]

H[i] <- p

if p > old_p:

  siftUp(i)

else: siftDown(i)
```

We also can create a heap from an array by using the function BuildHeap().

Build_heap(A[1..n])

```
size <- n

for i from [n/2] downto 1:

  siftDown(i)
```

And finally a Heap Sort algorithm can be implemented by using this Heap structure.

Heap_Sort(A)

```
Build_heap(A)

repeat (n-1) times:

  swap A[1] and A[size]

  size <- size -1

  siftDown(1)
```

In [ ]: