Chapter 6 Heapsort

Introduction to Algorithms

Mina Gabriel

6 Heapsort

Heap sort is like merge sort in its run time $O(nlogn)$ and like insertion sort when a constant number of elements stored out side the array at a time, Heap sort combines the better attributes of the two sorting algorithms.

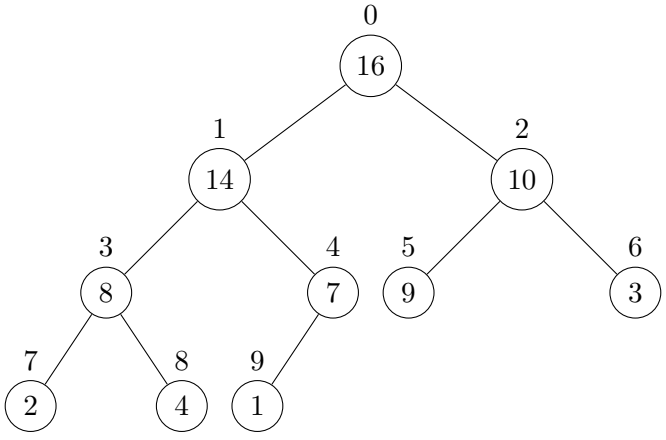
General formulas before we start:

If array index starts at 1 then:

- *PARENT(i)* = $\lfloor \frac{i}{2} \rfloor$
- *LEFT(i)* = $2i$
- *RIGHT(i)* = $2i + 1$

6.1 Heaps

The (binary) heap data structure is an array object that we can view as a nearly complete binary tree, Each node of the tree corresponds to an element of the array. An array A that represents a heap is an object with two attributes: A.length, which (as usual) gives the number of elements in the array, and A.heap-size, which represents how many elements in the heap are stored within array A.



16	14	10	8	7	9	3	2	4	1

6.2 Maintaining the heap property

In order to maintain the max-heap we call the MAX-HEAPIFY procedure:

Listing 1: MAX-HEAPIFY

```
def max_heapify(arr, n, i):
  print "Largest:", i , arr
  largest = i
  l = 2 * i + 1
  r = 2 * i + 2

  if l < n and arr[i] < arr[l]:
    largest = l

  if r < n and arr[largest] < arr[r]:
    largest = r

  #if parent is larger than swap
  if largest != i:
    arr[i],arr[largest] = arr[largest],arr[i]  # swap
    #Heapify
    print "------------>" , largest , arr
    #calling this again to go down the branch again recursively
    max_heapify(arr, n, largest)
```

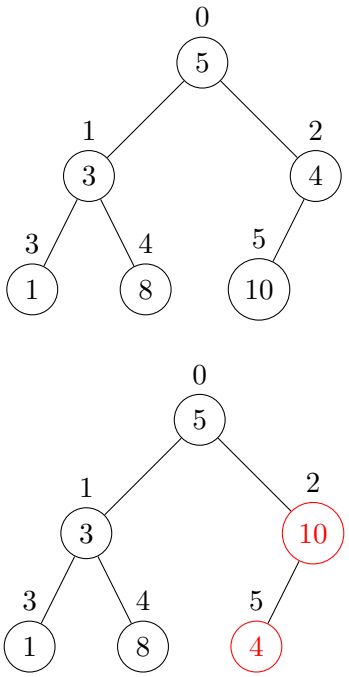
Using the previous procedure with $A = [5, 3, 10, 1, 8, 4]$

Listing 2: HEAPSORT

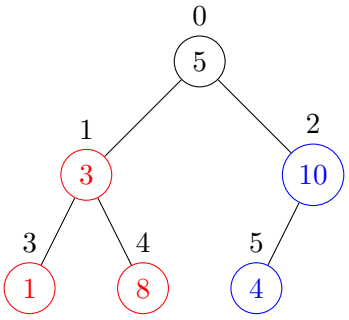
```
def heapSort(arr):
  n = len(arr)
  #initialy we need to build a maximum heap one time
  #this will visit all nodes except the ones at the leave
  for i in range(n, -1,-1):
    max_heapify(arr, n, i)

arr = [5,3,4,1,8,10]
heapSort(arr)
```

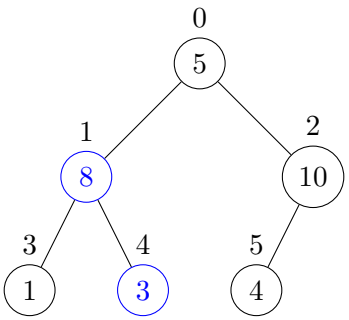
Notice: when we first call the MAX-HEAPIFY procedure i = 6, Inside the MAX-HEAPIFY l and r are equal 13 and 14 respectively, nothing will happen untill i = 2, l = 5 and r = 7 (node 7 is an element that doesn't exists)

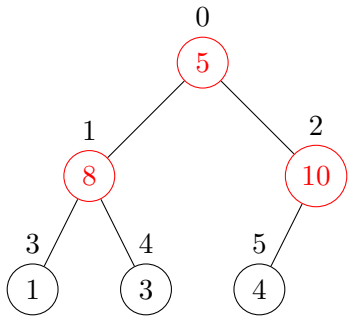


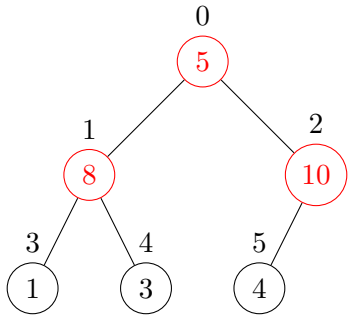
Notice: the recursive call of MAX-HEAPIFY with largest = 5 to mantin the maximum heap, in our example this will not run as node 5 is a leave node. After the recursive call, the for loop in the heapSort method will call MAX-HEAPIFY again with i minus 1 unit or i = 1.



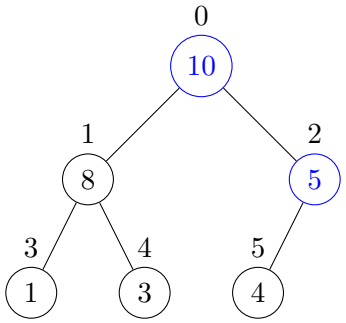
Since $A[i] \geq A[i * 2 + 1]$ and $A[i] \leq A[i * 2 + 2]$ change swap A[1] and A[4]



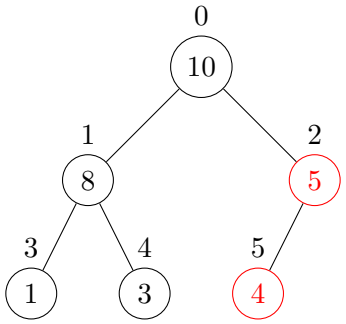




Now, i=0, l= 1 and r =2, largest will be $A[2]$ swap will take place between node 0 and node 2



Remember: the recursive call when i = 2 as largest.



Nothing to swap, now every child node in our tree is smaller than its parent

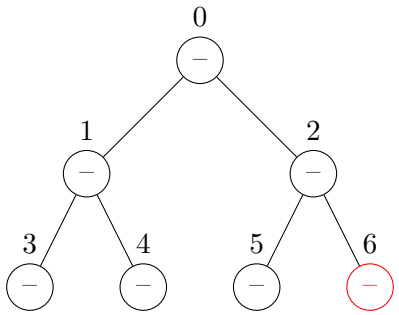
$$A[PARENT(i)] \geq A[i]$$

6.3 Building a heap

The running time of the MAX-HEAPIFY of size n - the amount of work we do to fix the relationship between the parent and it's children- is constant $\theta(1)$ plus the time we run MAX-HEAPIFY on every subtree.

To be able to compute the complexity of such algorithm, we need to maintain working on a full binary tree, and since we are counting for the upper and worst case this match our assumption perfectly.

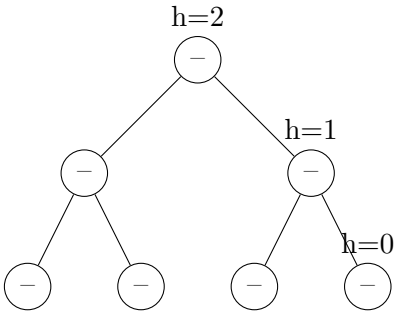
using our previous example our full tree must look like the following:



We need to look at the height of this tree in a different way, we need to count the height from the lowest level moving upwards, the bottom most level corresponding to height h is zero, something else to note here is the number of element n is 7 and not 6 to make every parent node of this tree has 2 children except the leaves.

We know from before that the height of the heap tree of n node is:

$$h = \lceil log_2 n \rceil \tag{1}$$



in the previous tree we have three different levels or height.

to find how many node per each level use the following equation - where h is the given height:

$$\left\lceil \frac{n}{2^{h+1}} \right\rceil \tag{2}$$

now to move every node from each level to the root in a worst case, every node has to travel $\lceil log_2\ n \rceil$ and since we have n nodes, then you may think all nodes travel $log_2 n * n$ this will be only true if all the nodes are on the same level. Every node has to travel from the height (level) where it is upwards, the leave node has to travel $log_2 n$ but their parents has to travel $log_2 n - 1$ and so on ...

Another way to think about this is all the leaf node don't have to do any swap since they are at height $h = 0$ and all the node of height $h = 1$ has to do 1 swap each and so one, if we add all the work each node has to do on all levels and multiply this by the amount of work each has to do $O(h)$ then we can figure out a more tight run time complexity for MAX-HEAPIFY procedure.

from equation 1 and 2 we get:

$$\sum_{h=0}^{\lceil log_2 n \rceil} \left\lceil \frac{n}{2^{h+1}} \right\rceil * O(h) \tag{3}$$

If we simplify equation 3 we can proof (page 159 Text Book) that the MAX-HEAPIFY procedures runs in $O(n)$

6.4 The heapsort algorithm

Every time the previous code runs a the largest number inside this list will be at the end of this list by swapping it with the smallest number in the heap and remove it, the heapsort then repeat this process for max-heap of size $n - 1$ down to a heap of size 2.

Listing 3: HEAP SORTING

```
def max_heapify(arr, n, i):
  print "Largest is: ", i , arr
  largest = i
  l = 2 * i + 1
  r = 2 * i + 2

  if l < n and arr[i] < arr[l]:
    largest = l

  if r < n and arr[largest] < arr[r]:
    largest = r

  #if parent is larger than swap
  if largest != i:
    arr[i],arr[largest] = arr[largest],arr[i]  # swap

    #Heapify
    print "---------->" , largest , arr
    max_heapify(arr, n, largest) #calling this again to go down the branch again r
def heapSort(arr):
  n = len(arr)
```

```
  #initialy we need to build a maximum heap one time
  #this will visit all nodes except the ones at the leave
  for i in range(n, -1,-1):
    max_heapify(arr, n, i)
  # One by one extract elements
  for i in range(n-1, 0, -1):
    arr[i], arr[0] = arr[0], arr[i]    # swap
    max_heapify(arr, i, 0)
arr = [ 5,3,4,1,8,10]
heapSort(arr)
```

The HEAPSORT procedure takes time $O(nlogn)$ since the call to building the heap takes $O(n)$ and each of the $n-1$ call to MAX-HEAPIFY takes time $O(logn)$

6.5 Priority queues

Heapsort is an excellent algorithm, but a good implementation of quicksort, presented in Chapter 7, usually beats it in practice.

the heap data structure itself has many uses. In this section, we present one of the most popular applications of a heap: as an efficient priority queue.

As with heaps, priority queues come in two forms: max-priority queues and min-priority queues. We will focus here on how to implement max-priority queues, which are in turn based on **maxheaps**.

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

- INSERT(S, x) inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$
- 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.

Now we discuss how to implement the operations of a max-priority queue. The procedure HEAP-MAXIMUM implements the MAXIMUM operation in $\Theta(1)$ time.

Algorithm 1 Priority Queue

HEAP-MAXIMUM(A)

1: return [1]

The running time of HEAP-EXTRACT-MAX is $O(log\ n)$, since it performs only a constant amount of work on top of the $O(log\ n)$ time for MAX-HEAPIFY.

Algorithm 2 Priority Queue

HEAP-EXTRACT-MAX(A)

```
1: if A:heap-size<1 then
2:    return error "heap underflow"
3: end if
4: max = A[1]
5: A[1] = A[A.heap-size]
6: A.heap-size = A.heap-size -1
7: MAX-HEAPIFY(A, 1)
8: return max
```

The procedure MAX-HEAP-INSERT implements the INSERT operation. It takes as an input the key of the new element to be inserted into max-heap A. The procedure first expands the max-heap by adding to the tree a new leaf.

Algorithm 3 Priority Queue

MAX-HEAP-INSERT(A; key)

```
1: A.heap-size = A.heap-size + 1
2: A.[A.heap-size] = key
3: MAX-HEAPIFY
```