CSE 5392 Fall 2005 Week 4

crystal.uta.edu/~gdas/Courses/Courses/Fall2005/advAlgos/student_**slides**/Week4.ppt -

Devendra Patel
Subhesh Pradhan

Heaps

- **Definition:**
  A **heap** is a **complete** binary tree with the condition that every node (except the root) must have a value greater than or equal to its parent.

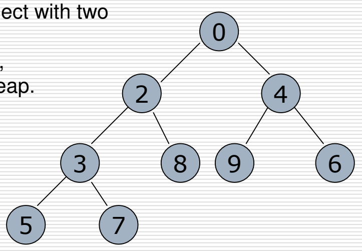
Heap representation:

A heap data structure is represented as an array A object with two attributes:

length[A] - number of elements in the array,
heap-size[A] - number of elements in the heap.
heap-size[A] ≤ length[A]

In an array A the root of the heap resides in A[1]
Consider a node with index *i*,

Index of parent is Parent(i) = $\lfloor i/2 \rfloor$
Index of left child of i is LEFT_CHILD(i) = 2 x i;
Index of right child of i is RIGHT_CHILD(i) = 2 x i+1;



A=

0	2	4	3	8	9	6	5	7	

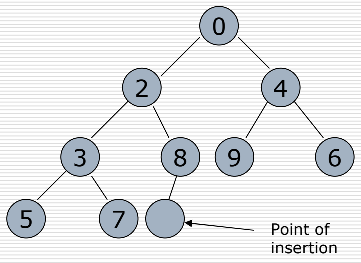
Operations on Heap

- For a dynamic set S
  - Insert (S,X)
  - delete_min (S)
  - Find_min (S)

Note: Heaps need not be represented as binary tree, they can be n-ary tree.

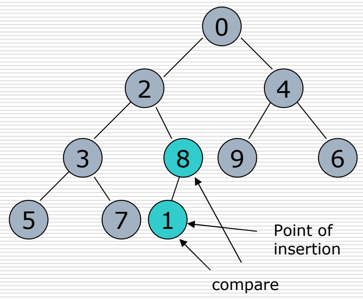
Insert (S,X)

- From the point of insertion to the root compare X with each node in the path
  - If X < node
    - Swap (X,node)



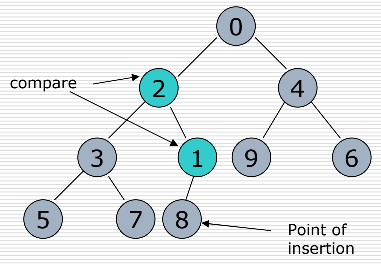
Insert (S,X)

- From the point of insertion to the root compare X with each node in the path
  - If X < node
    - Swap (X,node)



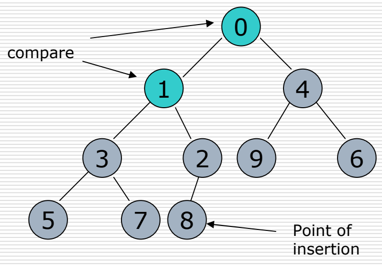
Insert (S,X)

- From the point of insertion to the root compare X with each node in the path
  - If X < node
    - Swap (X,node)



Insert (S,X)

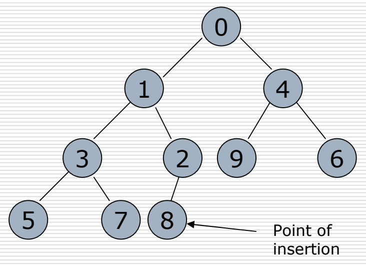
- From the point of insertion to the root compare X with each node in the path
  - If X < node
    - Swap (X,node)



Insert (S,X)

- From the point of insertion to the root compare X with each node in the path
  - If X < node
    - Swap (X,node)

Insert (S,X) = O(logn)



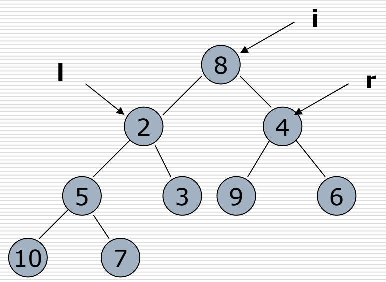
Heapify (S, i)

- **HEAPIFY**
- **HEAPIFY** is an important subroutine for maintaining heap property.
- Given a node *i* in the heap with children *l* and *r.* Each sub-tree rooted at *l* and *r* is assumed to be a heap. The sub-tree rooted at *i* may violate the heap property [***key(i) > key(l) OR key(i) > key(r)***]
- Thus Heapify lets the value of the parent node ***"float"*** down so the sub-tree at *i* satisfies the heap property.

Algorithm:

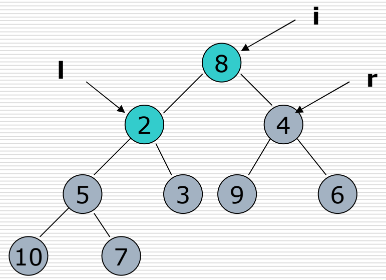
```
HEAPIFY(A, i)
1. l ← LEFT_CHILD (i);
2. r ← RIGHT_CHILD (i);
3. if l ≤ heap_size[A] and A[l] < A[i]
4.         then smallest ← l;
5.         else smallest ← i;
6. if r ≤ heap_size[A] and A[r] < A[smallest]
7.         then smallest ← r;
8. if smallest ≠ i
9.          then exchange A[i] ↔ A[smallest]
10.         HEAPIFY (A,smallest)
```

Heapify (S, i)

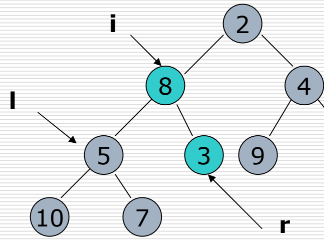


Heapify (S, i)

smallest =l

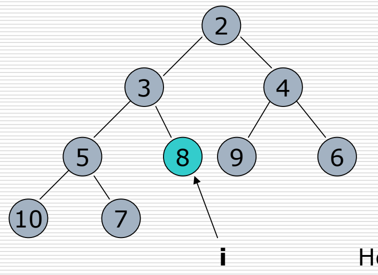


Heapify (S, i)



smallest =r

Heapify (S, i)



Heapify = O(logn)

Running Time of Heapify

- Fixing relation between *i*(a node), *l* (left child of *i*), *r* (right child of *i*) takes Θ(1) time. Let the heap at node *i* have n elements. The number of elements at subtree *l* or *r* , in worst case scenario is 2n/3 i.e. when the last level is half full.

  Or Mathematically

  $$T(n) \le T(2n/3) + \Theta(1)$$

  Applying Master Theorem (Case 2) , we can solve the above to

  $$\mathbf{T(n)=O(\log n)}$$

  Alternatively ,

  In the worst case the algorithm needs walking down the heap of height ***h= log n***. Thus the running time of the algorithm is ***O(log n)***

Build Heap

- This procedure builds a heap of the array using the Heapify algorithm
- Initially the array representing heap will have random elements
- **BUILD_HEAP(A)**

```
1.   heap_size [a] ← length [A]
2.   for i ← ⌊ length [A]/2 ⌋ downto 1 do
3.       HEAPIFY (A, i)
```

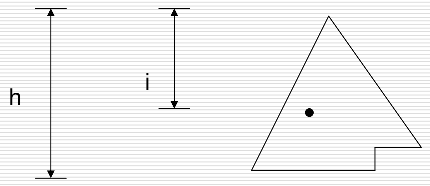
Alternative Approach

Build heap might be repeatedly calling Insert (S,X) into an initially empty heap

Running time for this approach is **O(nlogn)**

Running Time of Build_Heap

- We represent heap in the following manner



For nodes at level i , there are $\mathbf{2^i}$ nodes . And the work is done for **h-i** levels.

$$\text{Total work done} = \sum_{i=1}^{h=\log n} 2^i * (h-i)$$

$$= \sum_{i=1}^{h=\log n} 2i * (\log n - i) \quad (\text{taking } h=\log n)$$

Running Time of Build_Heap

Substituting $j = \log n - i$ we get ,

$$\text{Total work done} = \sum_{j=\log n}^{1} 2^{\log n - j} * j$$

$$= \sum_{j=1}^{\log n} (2^{\log n} / 2^j) * j$$

$$= n \sum_{j=1}^{\log n} j / 2^j$$

$$= O(n)$$

Thus running time of Build_Heap = O(n)

HeapSort

HEAPSORT(A)

```
1)   BUILD_HEAP(A)
2)   for i <--- length [A] downto 2
3)       do exchange A[1] <------> A[i]
4)           heap-size[A] ← heap-size[A]-1
5)           HEAPIFY(A,1)
```

Running time of HEAPSORT
The call to BUILD_HEAP takes O(n) time and each of the n-1 calls to MAX-HEAPIFY takes O (log n) time. Thus HEAPSORT procedure takes **O(n log n)** time.

Why doesn't Heapsort take O(log n) time as in the case of other Heap algorithms?

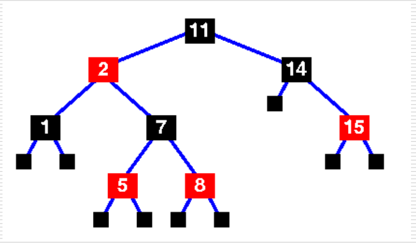
Consider the Build_Heap algorithm, a node is pushed down and since the lower part of the heap is decreasing at each step the number of leaf node operations performed decreases logarithmically. While in HeapSort the node moves upwards. Thus the decreasing lower part does not reduce the number of operations.

Performance of different Data Structures

	Unsorted Array	Unsorted Array with min variable	Sorted Array	Binary Search Tree	Heap
Insert (S, X)	O(1)	O(1)	O(n)	O(path_length)	O(logn)
Delete (S,X)	O(n)	O(n)	O(n)	O(path_length)	O(logn)
Find_min (S)	O(n)	O(1)	O(1)	O(path_length)	O(1)

Red Black Tree

- Property
  - Smallest path is no less than half of the largest path



Reference

- Some of the material is taken from Fall 2004 Presentation for Week 4 prepared by Jatinder Paul, Shraddha Rumade