

## CE2001/ CZ2001: Algorithms

### Heapsort

Ke Yiping, Kelly

### Learning Objectives

At the end of this lecture, students should be able to:

- Explain the definition and properties of a heap
- Describe how Heapsort works
- Explain how to construct a heap from an input array
- Analyse the time complexity of Heapsort

2

### Introduction to Heapsort

### Heapsort

Heapsort is based on a heap data structure.

▪ **The definition of a heap includes:**

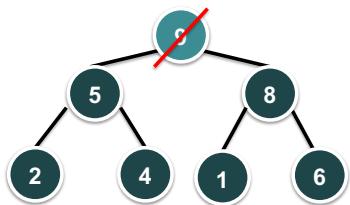
- a description of the structure.
- a condition on the data in the nodes (of a binary tree) called **partial order tree property**.

▪ **Partial order tree property**

A tree  $T$  is a (**maximising**) partial order tree if and only if each node has a key value **greater than or equal to** each of its child nodes (if it has any).

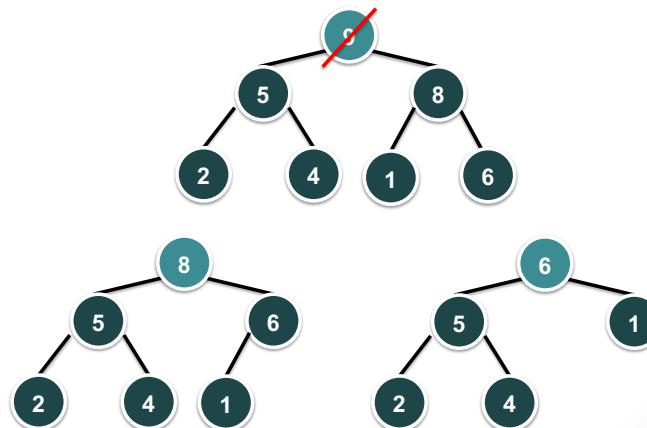
4

## Partial Order Tree Property



5

## Partial Order Tree Property



6

## Heapsort

Heapsort is based on a heap data structure.

- **The definition of a heap includes:**

- a description of the structure.
- a condition on the data in the nodes (of a binary tree) called **partial order tree property**.

- **Partial order tree property**

A tree T is a **(maximising)** partial order tree if and only if each node has a key value **greater than or equal to** each of its child nodes (if it has any).

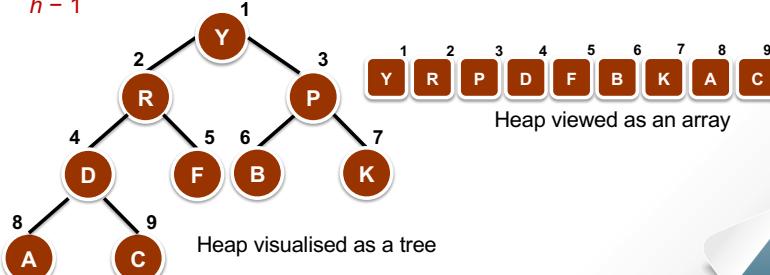
- For a **minimising** partial order tree, the key value of every parent node is **less than or equal to** the value of each of its child nodes.

7

## Heap Structure

## Heap Structure

- A binary tree  $T$  with height  $h$  is a heap structure if and only if it satisfies the following conditions:
  - $T$  is complete at least through depth  $h - 1$
  - all leaves are at depth  $h$  or  $h - 1$
  - all paths to a leaf of depth  $h$  are to the left of all paths to a leaf of depth  $h - 1$

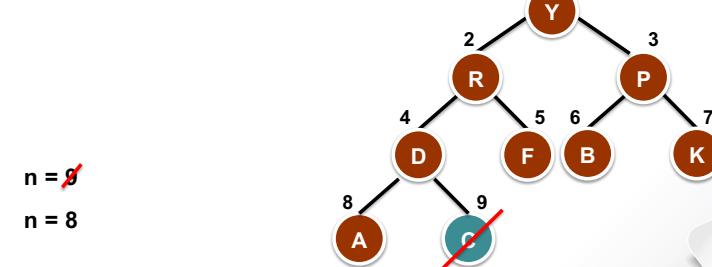


9

## Heap Structure

- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.
- Implementing the tree with  $n$  nodes by an array:**

1) Entry  $i$  in the array is a **tree node** if  $1 \leq i \leq n$



10

## Heap Structure

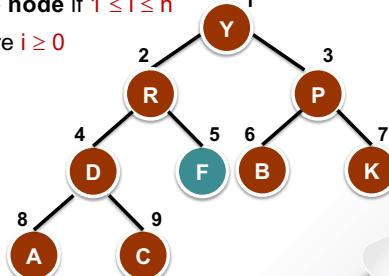
- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.

### Implementing the tree with $n$ nodes by an array:

1) Entry  $i$  in the array is a **tree node** if  $1 \leq i \leq n$

2) Parent ( $i$ ): return  $\lfloor i/2 \rfloor$  where  $i \geq 0$

$i = 5$   
Parent (5) =  $\lfloor 5/2 \rfloor = 2$



11

## Heap Structure

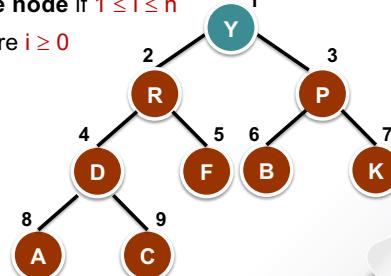
- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.

### Implementing the tree with $n$ nodes by an array:

1) Entry  $i$  in the array is a **tree node** if  $1 \leq i \leq n$

2) Parent ( $i$ ): return  $\lfloor i/2 \rfloor$  where  $i \geq 0$

$i = 1$   
Parent (1) =  $\lfloor 1/2 \rfloor = 0$



12

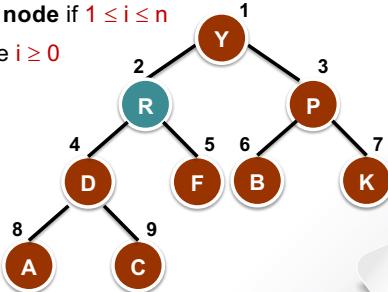
## Heap Structure

- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.

- Implementing the tree with n nodes by an array:**

- Entry  $i$  in the array is a tree node if  $1 \leq i \leq n$
- Parent ( $i$ ): return  $\lfloor i/2 \rfloor$  where  $i \geq 0$
- Left subtree of  $i$ : return  $2i$

$$\begin{aligned} i &= 2 \\ \text{Left subtree (i)} &= 2 \times 2 = 4 \end{aligned}$$



13

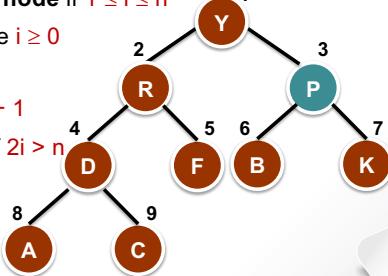
## Heap Structure

- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.

- Implementing the tree with n nodes by an array:**

- Entry  $i$  in the array is a tree node if  $1 \leq i \leq n$
- Parent ( $i$ ): return  $\lfloor i/2 \rfloor$  where  $i \geq 0$
- Left subtree of  $i$ : return  $2i$
- Right subtree of  $i$ : return  $2i + 1$
- array[i] is a leaf if and only if  $2i > n$

$$\begin{aligned} i &= 3, \quad n = 9 \\ 2 \times 3 &\text{ is not } > 9 \\ \text{Node 3 is not a leaf node.} \end{aligned}$$



15

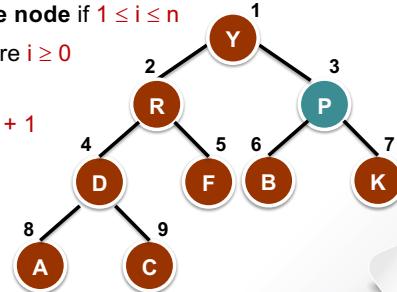
## Heap Structure

- This means that every successive level of the tree must fill up from left to right. Further, an entire level must be full before any nodes at that level can have children nodes.

- Implementing the tree with n nodes by an array:**

- Entry  $i$  in the array is a tree node if  $1 \leq i \leq n$
- Parent ( $i$ ): return  $\lfloor i/2 \rfloor$  where  $i \geq 0$
- Left subtree of  $i$ : return  $2i$
- Right subtree of  $i$ : return  $2i + 1$

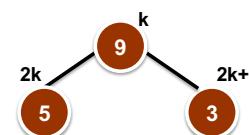
$$\begin{aligned} i &= 3 \\ \text{Right subtree (i):} &= (2 \times 3) + 1 = 7 \end{aligned}$$



14

## Heap Structure

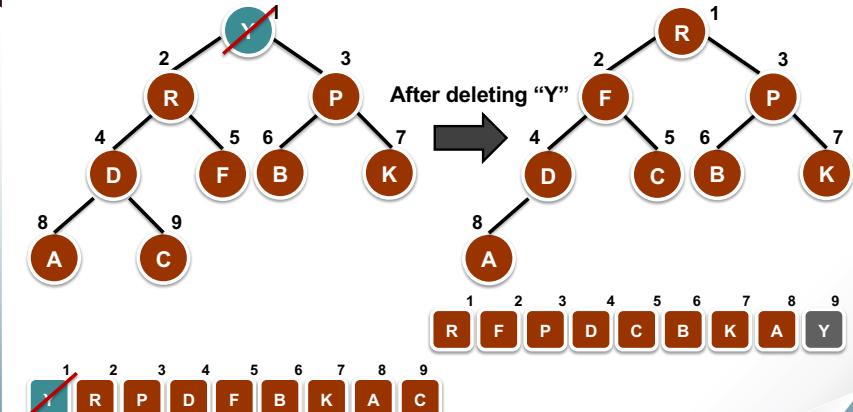
- Therefore the partial order tree property requires that for all positions  $k$  in the list, the key at  $k$  is at least as large as the keys at  $2k$  and  $2k + 1$  (if these positions exist).



16

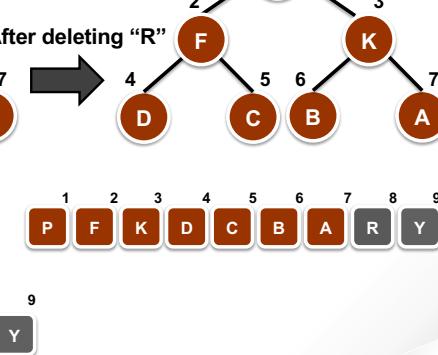
## Heapsort (Example)

## Heapsort (Example)



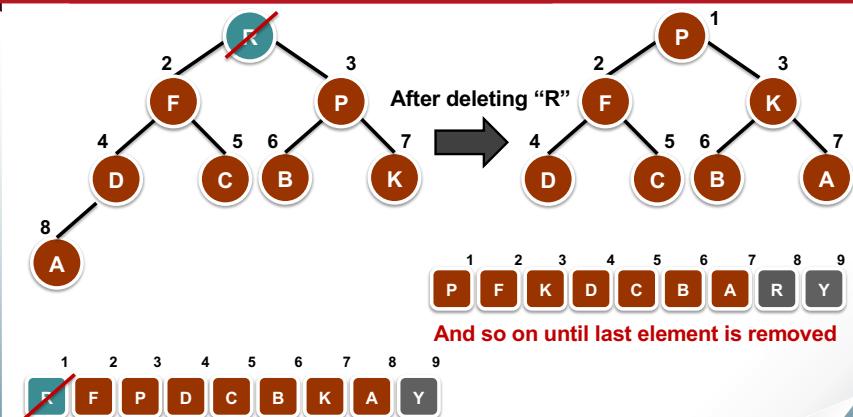
18

## Heapsort (Example)



19

## Heapsort (Example)



20

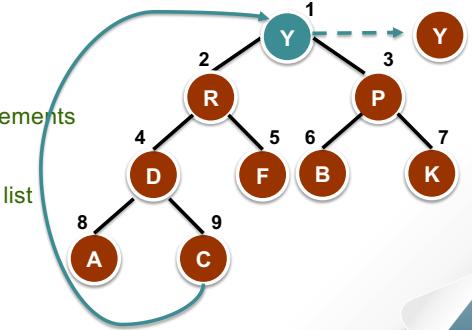
## Heapsort Method

## Heapsort Method

### heapSort (array, n)

```
{
    construct heap H from array with n elements;
    for (i = n; i >= 1; i--)
        {
            curMax = getMax(H);
            deleteMax(H);
            // as result, H has i - 1 elements
            array[i] = curMax;
            // insert curMax in sorted list
        }
}
```

Take out last  
and re-insert



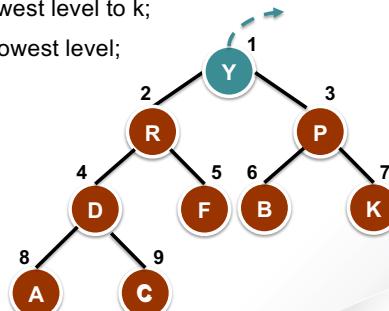
22

## Heapsort Method

### deleteMax(H)

```
{
    copy the rightmost element at the lowest level to k;
    delete the rightmost element at the lowest level;
    fixHeap(H, k);
}
```

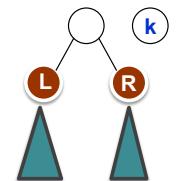
k



23

## fixHeap

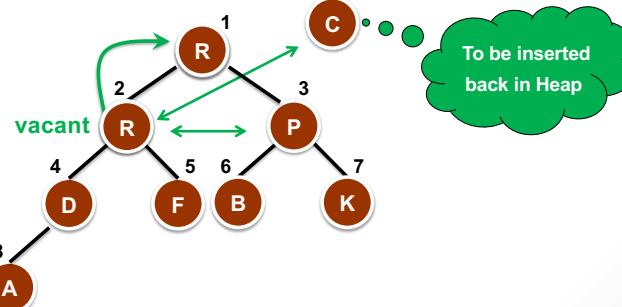
```
fixHeap(H, k) { // recursive
    if (H is a leaf)
        insert k in root of H;
    else {
        compare left child with right child;
        largerSubHeap = the larger child of H;
        if ( k >= key of root(largerSubHeap) )
            insert k in root of H;
        else {
            insert root(largerSubHeap) in root of H;
            fixHeap(largerSubHeap, k);
        }
    }
}
```



24

## fixHeap

**fixHeap(H,C)**



R > P and R is also > C; so R is inserted into Root, and the original slot of R becomes vacant.

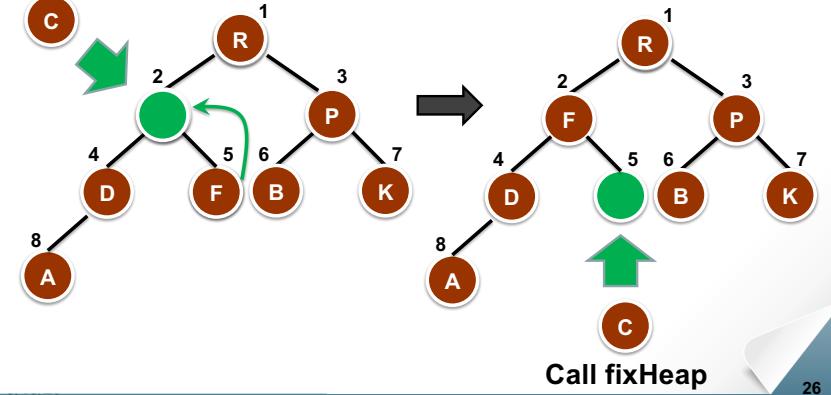
fixHeap is called again to reinsert C into the sub-heap.

25

## fixHeap

At this point, the subtree is a leaf.  
Hence, C is inserted.

Call fixHeap

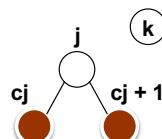


Call fixHeap

26

## fixHeap

```
fixHeap(H, k)      // iterative
{
    int j = 1,          // root of the heap
    cj = 2;            // left child of the root
    while (cj <= currentSize)
    { // cj should be the larger child of j
        if (cj < currentSize && H[cj] < H[cj+1]) cj++;
        if (k >= H[cj]) break; // should put k in H[j]
        H[j] = H[cj]; // move larger child to H[j]
        j = cj; // move down one level
        cj = 2 * j; // cj is the left child of j
    }
    H[j] = k;
}
```



27

## Heap Construction

## Heap Construction

### Construct a heap from an array

Start by putting all elements of the array in a heap structure in arbitrary order; then, “heapifying” the heap structure.

**constructHeap(array, H)**

```
{
    put all elements of array into a heap structure H in
    arbitrary order;
    heapifying(H);
}
```

Uses the `fixheap` function
 mentioned earlier

29

## Heap Construction

**heapifying(H)**

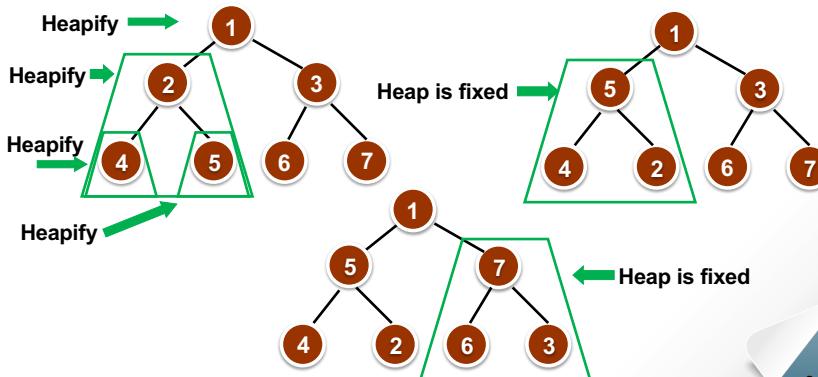
```
{
    if (H is not a leaf) {
        heapifying(left subtree of H);
        heapifying(right subtree of H);
        k = root(H);
        fixHeap(H, k);
    }
}
```

Post-order traversal  
of a binary tree

30

## Heap Construction

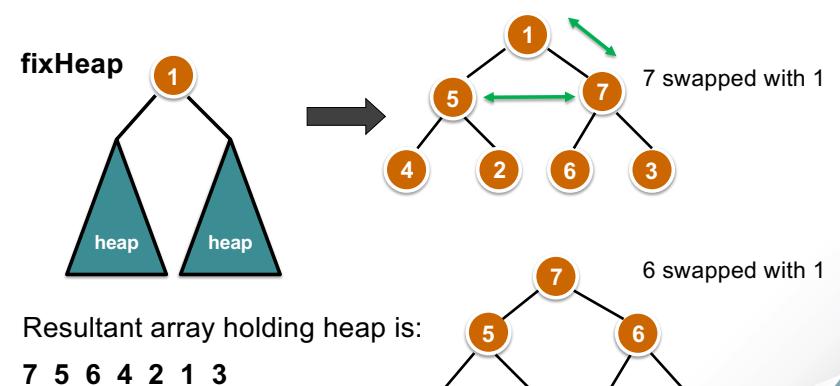
Assume elements in initial arbitrary order: 1 2 3 4 5 6 7



31

## Heap Construction

**fixHeap**



32

## Time Complexity of Heapsort

## Time Complexity of fixHeap

```

fixHeap(H, k)           // recursive
{
    if (H is a leaf)      // Heap has just one node      O(1)
        insert k in root of H;                         O(1)
    else {
        LargerSH = Sub-Heap at larger child of H's root;   O(1)
        if (k >= LargerSH's root key)                      O(1)
            insert k in root of H;                         O(1)
        else {
            insert LargerSH's root key in root of H;       O(1)
            fixHeap(LargerSH, k);                           O(1)
        }
    }
}

```

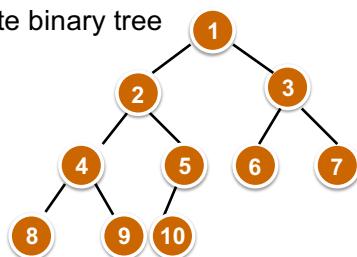
Each recursive call moves down a level  
 Total no. of key comparisons  $\leq 2 \times$  tree height

34

## Time Complexity of fixHeap

**Recall:** A heap is a nearly complete binary tree

**Note:** A complete binary tree of  $k$  levels has  $2^k - 1$  nodes (prove by mathematical induction)



35

## Time Complexity of fixHeap

A heap with

1 level has  $\leq 1 (= 2^1 - 1)$  node;

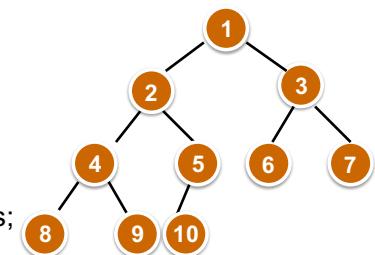
2 levels has  $\leq 3 (= 2^2 - 1)$  nodes;

3 levels has  $\leq 7 (= 2^3 - 1)$  nodes;

4 levels has  $\leq 15 (= 2^4 - 1)$  nodes;

$k - 1$  levels has  $\leq 2^{k-1} - 1$  nodes;

$k$  levels has  $\leq 2^k - 1$  nodes.



36

## Time Complexity of fixHeap

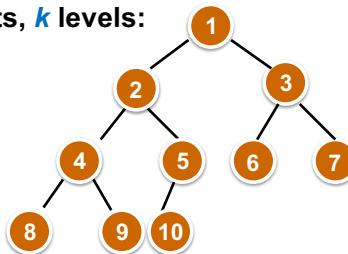
Assume the heap has  $n$  elements,  $k$  levels:

$$2^{k-1} - 1 < n \Rightarrow 2^{k-1} \leq n$$

$$2^{k-1} \leq n \leq 2^k - 1$$

$$\Rightarrow k - 1 \leq \lg n < k$$

$$\Rightarrow k - 1 = \lfloor \lg n \rfloor$$



Height of a heap with  $n$  nodes is  $O(\lg n)$ .

Worst-case time complexity of fixHeap is  $O(\lg n)$ .

37

## Time Complexity of heapifying

**heapifying(H)**

{

if ( $H$  is not a leaf)

{

    heapifying(left subtree of  $H$ );

    heapifying(right subtree of  $H$ );

$k = \text{root}(H)$ ;

    fixHeap( $H$ ,  $k$ );

}

}

 $W(n)$  $O(1)$  $W((n-1)/2)$  $W((n-1)/2)$  $O(1)$  $2 \lg n$ 

38

## Time Complexity of heapifying

- Assume a heap is a **full** binary tree, i.e.  $n = 2^d - 1$  for some non-negative integer  $d$ . The worst-case time complexity of heapifying(), i.e.  $W(n)$ , satisfies:

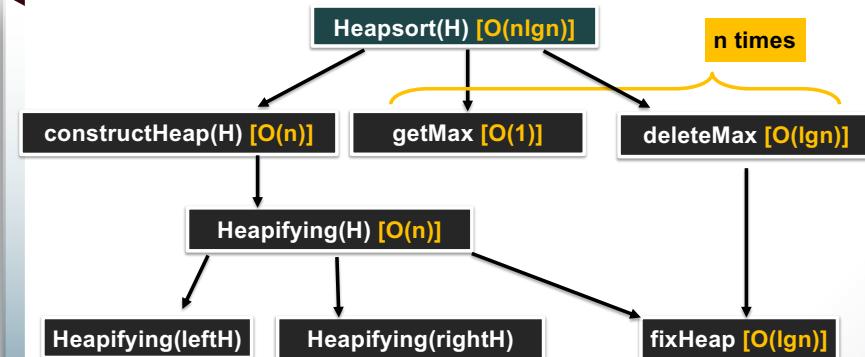
$$W(n) = 2W((n-1)/2) + 2\lg n$$

- Solving this equation gives  $W(n) = O(n)$  comparisons of keys in the worst-case.

(How to solve the recurrence equation is not required)

39

## Heapsort Performance



40

## Priority Queues

### Priority Queues (Optional, for self-learning)

- A priority queue is a data structure for maintaining a set S of elements, each with a key value. This key is considered as the 'priority' of the element in S.
- Priority queues are frequently used in job scheduling, simulation systems etc.
- A priority queue supports the following operations:
  - insert(x)** inserts the element x into a priority queue pq.
  - Maximum(pq)** returns largest key from pq.
  - extractMax(S)** removes largest key and re-arranges pq.
- Using a heap allows an efficient way of implementing a priority queue.

41

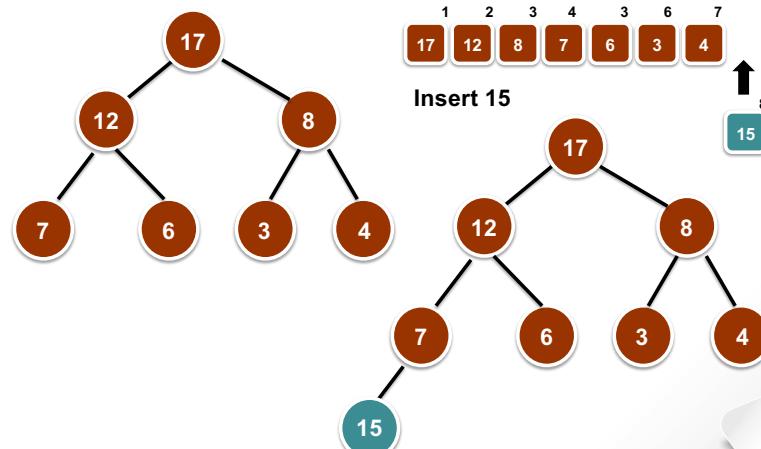
## Priority Queues

### Class pq // Java code

```
{
  private:
    ALIST pq;
    int N;           // size of priority queue
  public:
    // initialisation & other methods such as EMPTY omitted
    void insert (item i)
    { pq[ ++N ] = i; fixUp(pq,N); }
    item extractMax()
    { swap(pq[1], pq[N]); fixDown(pq, 1, N - 1);
      return pq[ N -- ]; }
}
```

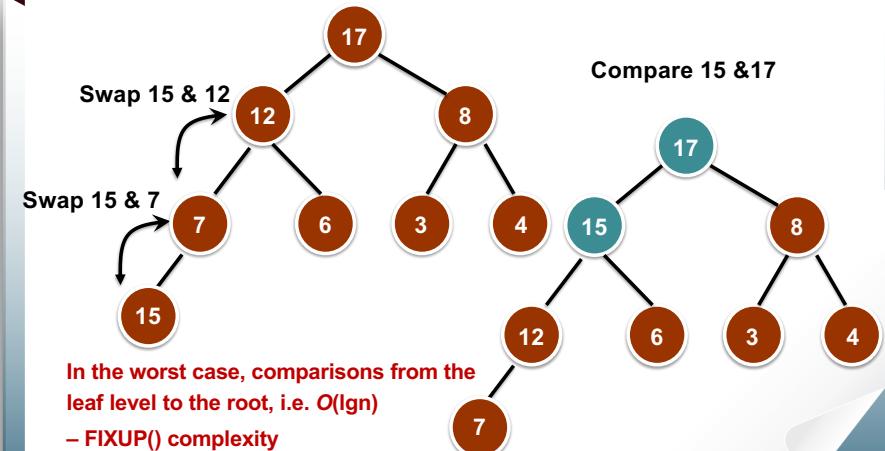
42

## Action of Fixup



43

## Action of Fixup



44

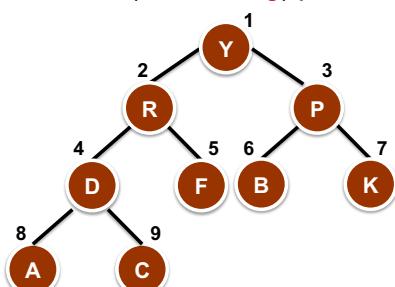
## Action of Fixup

- The running time of `insert()` on an  $n$ -element heap is  $O(\lg n)$  – same as `fixUp()`
- The running time of `extractMax()` on an  $n$ -element heap is  $O(\lg n)$  – same as `fixHeap()`
- The running time of `Maximum(pq)` (i.e. `getMax()`) on an  $n$ -element heap is  $O(1)$  - simply gets `pq[1]`
- So a heap can support any priority queue operation on a set of  $n$  elements in  $O(\lg n)$  time

45

## Summary

- Heapsort is a sorting algorithm using the data structure of heap.
- A (**maximising**) heap is an almost complete binary tree that satisfies the (**maximising**) partial order tree property.

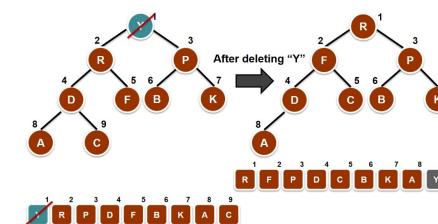


47

## Heapsort (Summary)

## Summary

- Heapsort is a sorting algorithm using the data structure of heap.
- A (**maximising**) heap is an almost complete binary tree that satisfies the (**maximising**) partial order tree property.
- Heapsort works by repeatedly deleting the root (maximum node) of the heap, and repair the damage (`fixHeap`).



48

## Summary

- Heapsort is a sorting algorithm using the data structure of heap.
- A (**maximising**) heap is an almost complete binary tree that satisfies the (**maximising**) partial order tree property.
- Heapsort works by repeatedly deleting the root (maximum node) of the heap, and repair the damage (**fixHeap**).
- Heap is constructed by recursively calling **fixHeap** in a post-order traversal of the binary tree.
- In **worst-case**, **heapsort** takes time  $\Theta(n \lg n)$ , and **heap construction** takes linear time  $\Theta(n)$ .

49

## Comparison of Sorting Algorithms

### Time complexity comparison:

	Best	Average	Worst
Insertion	$n$	$n^2$	$n^2$
Merge	$n \log n$	$n \log n$	$n \log n$
Quick	$n \log n$	$n \log n$	$n^2$
*Radix	$n$	$n$	$n$
Heap	$n \log n$	$n \log n$	$n \log n$

\* Radix sort is not required

51

## Comparison of Sorting Algorithms

## Empirical Comparison

### Compared by time (in milliseconds)

	Up	Down	100,000 numbers
Insertion	0.1	168	342
Merge	2.0	2.3	2.2
Quick	0.7	0.9	0.7
*Radix	1.6	1.6	1.6
Heap	3.4	3.5	3.6

Reference: Shaffer, C. A. (2001). *A practical introduction to data structures and algorithm analysis*. Upper Saddle River, NJ: Prentice Hall.

'UP' and 'DOWN' columns show the performance for inputs of size 10,000 where the numbers are in ascending (sorted) and descending (reversely sorted) order. Figures are timings obtained using workstation running UNIX.

52