

Data Structures and Algorithms

Lecture 7: Priority Queue (Heap) & Heapsort



Motivating Example

3 jobs have been submitted to a printer in the order A, B, C.

Sizes: Job A – 100 pages

Job B – 10 pages

Job C – 1 page

Average waiting time with FIFO service:

$$(100 + 110 + 111) / 3 = 107 \text{ time units}$$

Average waiting time for shortest-job-first service:

$$(1 + 11 + 111) / 3 = 41 \text{ time units}$$

A queue be capable to **insert** and **deletemin**?

Priority Queue



Priority Queue

- **Priority queue** is a **data structure** which allows at least two operations
 - ◆ **insert**
 - ◆ **deleteMin**: finds, returns and removes the minimum elements in the priority queue



- Applications: external sorting, greedy algorithms

Possible Implementations

- **Linked list**

- ◆ Insert in $O(1)$
- ◆ Find the minimum element in $O(n)$, thus deleteMin is $O(n)$

- **Binary search tree** (AVL tree, to be covered later)

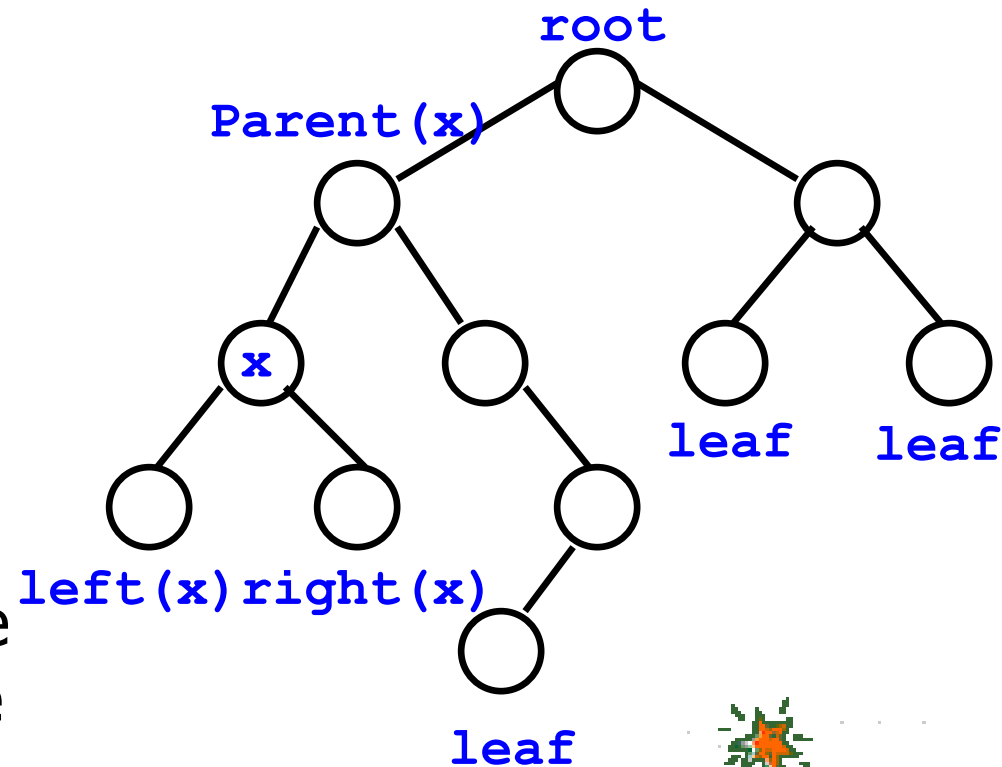
- ◆ Insert in $O(\log n)$
- ◆ Delete in $O(\log n)$
- ◆ Search tree is an overkill as it does many other operations

- Err, neither fit quite well...



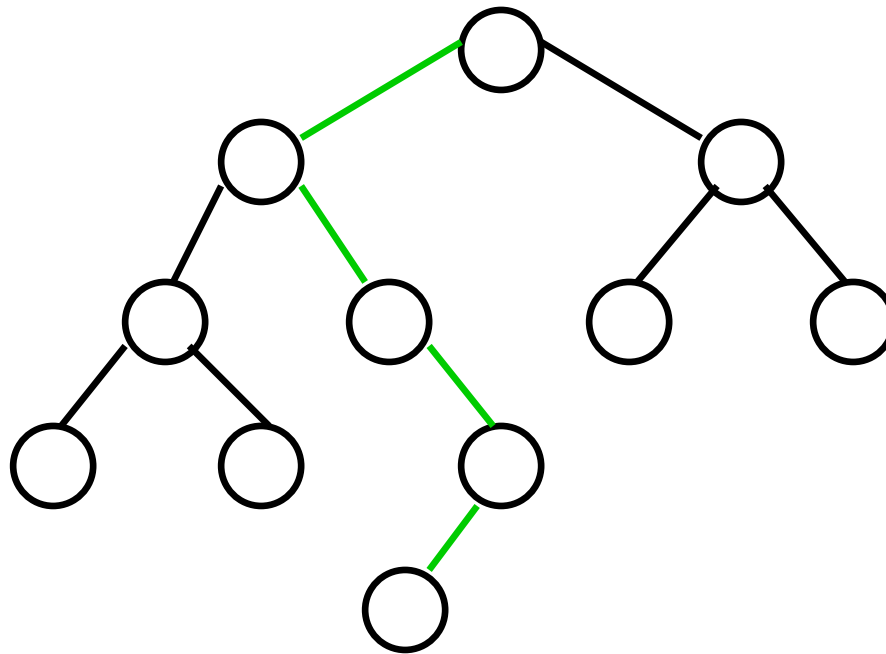
Background: Binary Trees (二叉树)

- Has a **root** at the topmost level
- Each **node** has **zero, one or two children**
- A node that has no child is called a **leaf**
- For a node x , we denote the **left child**, **right child** and the **parent** of x as $\text{left}(x)$, $\text{right}(x)$ and $\text{parent}(x)$, respectively.



Height (Depth) of a Binary Tree

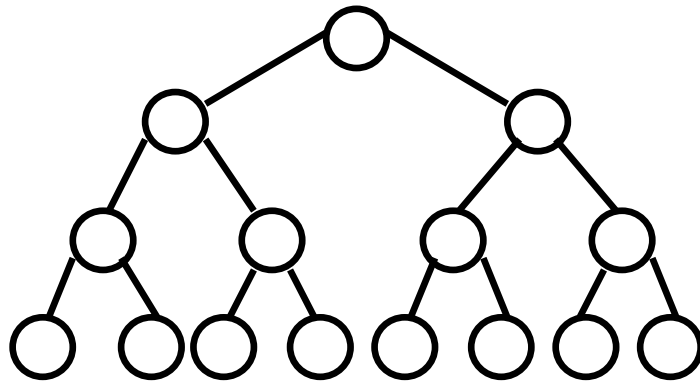
- The number of edges on the **longest** path from the root to a leaf.



Height = 4

Background: Complete Binary Trees

- A **complete binary tree** is the tree
 - ◆ Where a node can have 0 (for the leaves) or 2 children and
 - ◆ All leaves are at the same depth



height	no. of nodes
0	1
1	2
2	4
3	8
d	2^d

- No. of nodes and height
 - ◆ A complete binary tree with **N nodes** has height $O(\log N)$
 - ◆ A complete binary tree with **height d** has $2^{d+1}-1$ nodes

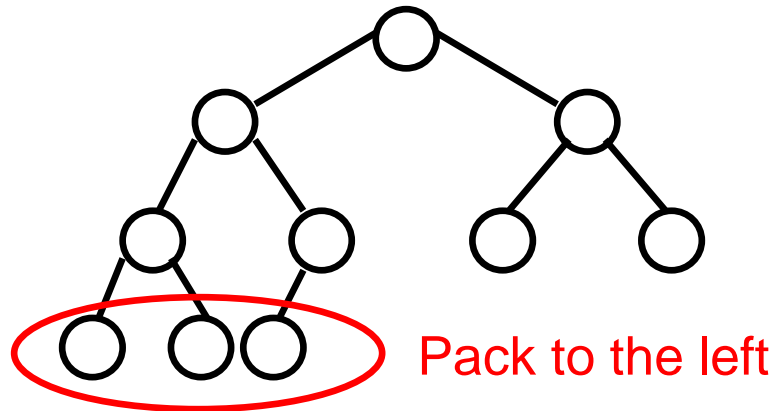
Proof: $O(\log N)$ Height

- Proof: a complete binary tree with N nodes has height of $O(\log N)$
 1. Prove by induction that number of nodes at depth d is 2^d
 2. Total number of nodes of a complete binary tree of depth d is $1 + 2 + 4 + \dots + 2^d = 2^{d+1} - 1$
 3. Thus $2^{d+1} - 1 = N$
 4. $d = \log(N+1) - 1 = O(\log N)$
- Side notes: the largest depth of a binary tree of N nodes is $O(N)$ (what is the shape of the tree?)



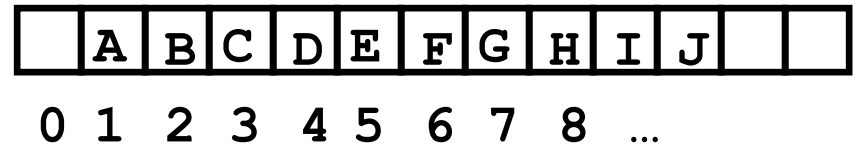
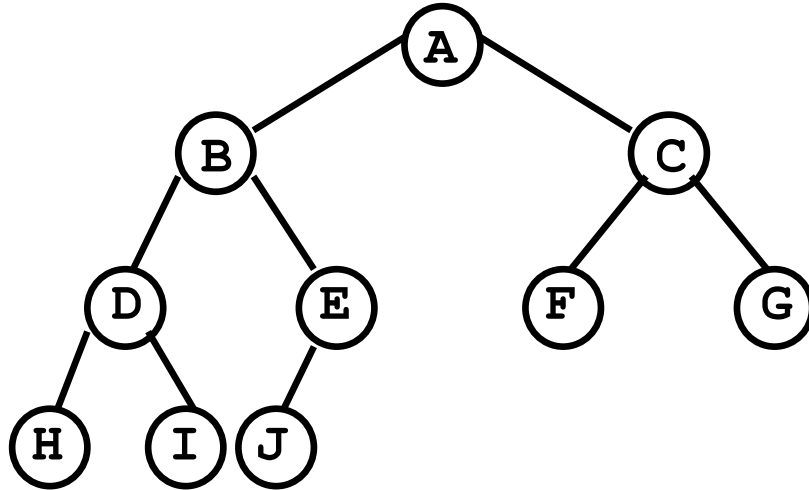
Binary Heap

- Heaps are “almost complete binary trees”
 - ◆ All levels are full except possibly the lowest level
 - ◆ If the lowest level is not full, then nodes must be packed to the left

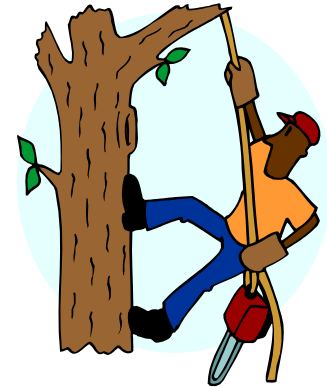


- Structure properties
 - ◆ Has 2^h to $2^{h+1}-1$ nodes with height h
 - ◆ The structure is so regular, it can be represented in an array and no links are necessary !!!

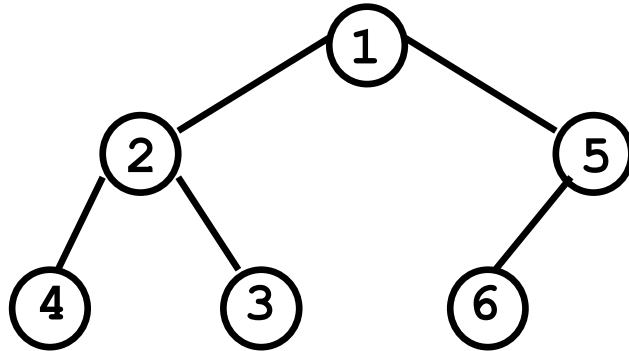
Array Implementation of Binary Heap



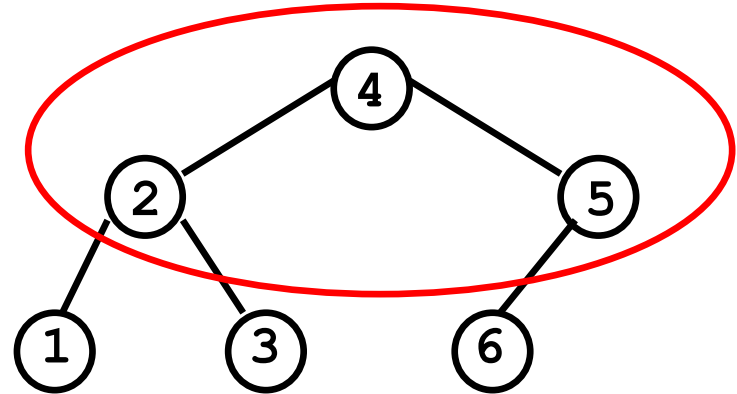
- For any element in array position i
 - ◆ The left child is in position $2i$
 - ◆ The right child is in position $2i+1$
 - ◆ The parent is in position $\text{floor}(i/2)$
- A possible problem: an estimate of the maximum heap size is required in advance (but normally we can resize if needed)
- Note: we will draw the heaps as trees, with the implication that an actual implementation will use simple arrays
- Side notes: it's not wise to store normal binary trees in arrays, coz it may generate many holes



A Minimum Priority Queue has



A minimum heap



Not a minimum heap

- Binary Heap structure
- Heap-order property: the value at each node is less than or equal to the values at both its descendants
 - ◆ The smallest node is always on the top
 - ◆ It is easy (both conceptually and practically) to perform `insert` and `deleteMin` in heap if the heap-order property is maintained
- Use of binary heap is so common for priority queue implementations, thus the word heap is usually assumed to be the implementation of the data structure

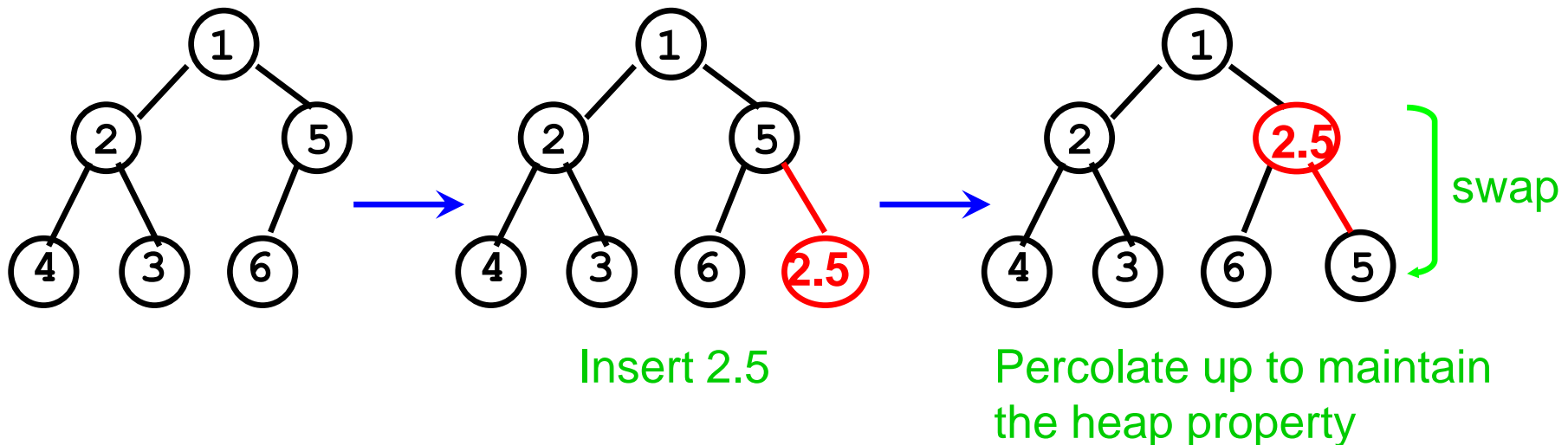
Heap Properties

- Heap supports the following operations efficiently
 - ◆ Insert in $O(\log N)$ time
 - ◆ Locate the current minimum in $O(1)$ time
 - ◆ Delete the current minimum in $O(\log N)$ time

Insertion

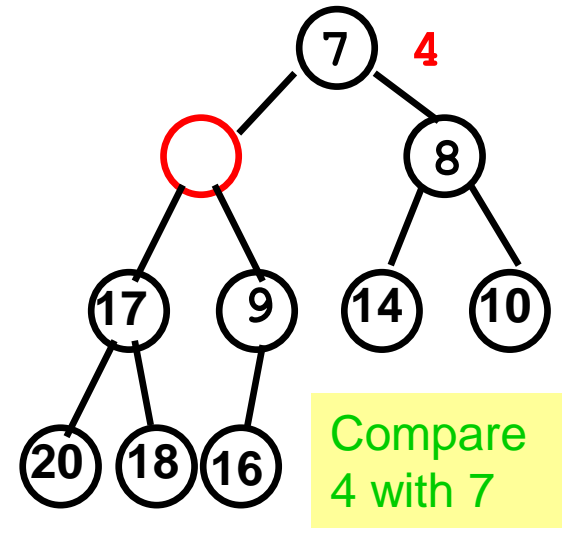
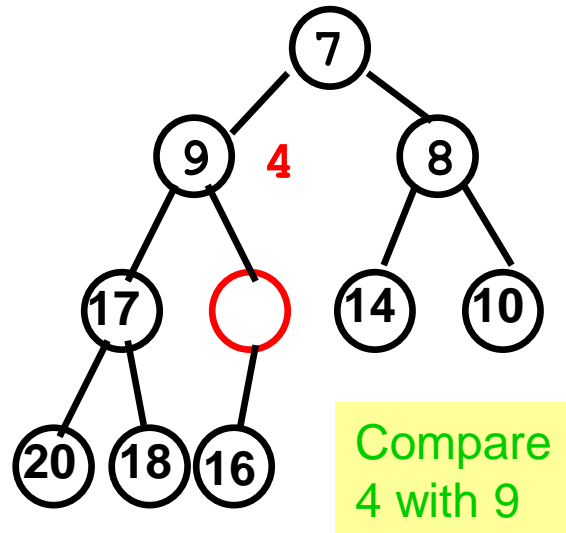
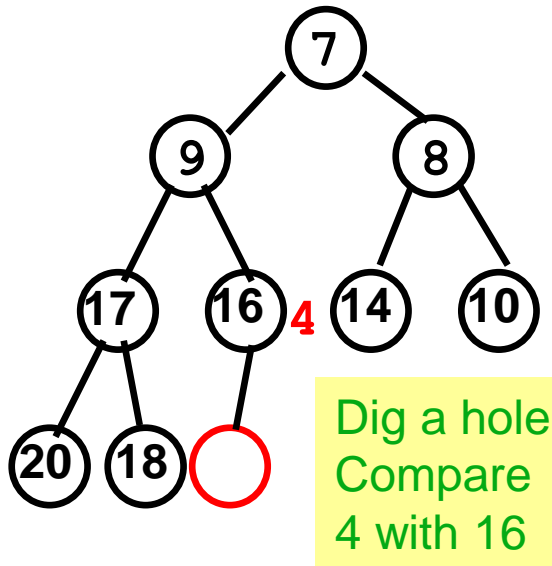
■ Algorithm

1. Add the new element to the **next available position** at the **lowest level**
2. Restore the **min-heap property** if violated
 - General strategy is **percolate up** (or bubble up): if the parent of the element is larger than the element, then interchange the parent and child.

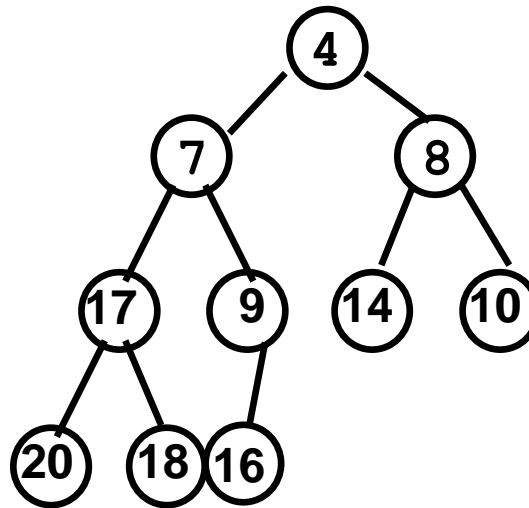


An Implementation Trick

- Implementation of percolation in the insert routine
 - ◆ by performing **repeated swaps**: 3 assignment statements for a swap. **3d assignments** if an element is percolated up **d levels**
 - ◆ An enhancement: **Hole digging** with **d+1 assignments**
- **Insert 4...**



Insertion Complexity



A heap!

Time Complexity = $O(\text{height}) = O(\log N)$

Insertion PseudoCode

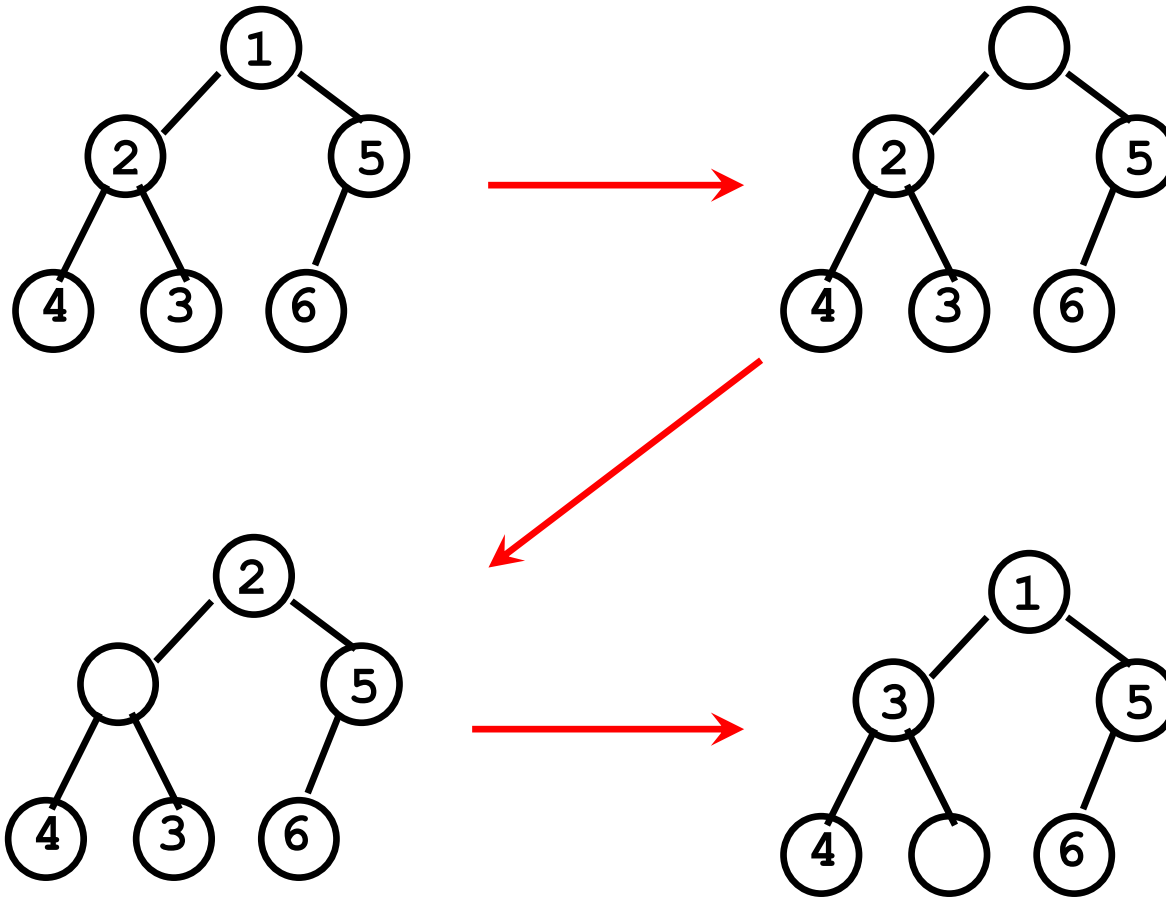
```
void insert(const Comparable &x)
{
    //resize the array if needed
    if (currentSize == array.size()-1)
        array.resize(array.size()*2)
    //percolate up
    int hole = ++currentSize;
    for (; hole>1 && x<array[hole/2]; hole/=2)
        array[hole] = array[hole/2];
    array[hole]= x;
}
```


deleteMin: First Attempt=percolate down

■ Algorithm

1. Delete the root.
2. Compare the two children of the root
3. Make the **lesser** of the two the root.
4. An empty spot is created.
5. Bring the lesser of the two children of the empty spot to the empty spot.
6. A new empty spot is created.
7. Continue

Example for First Attempt

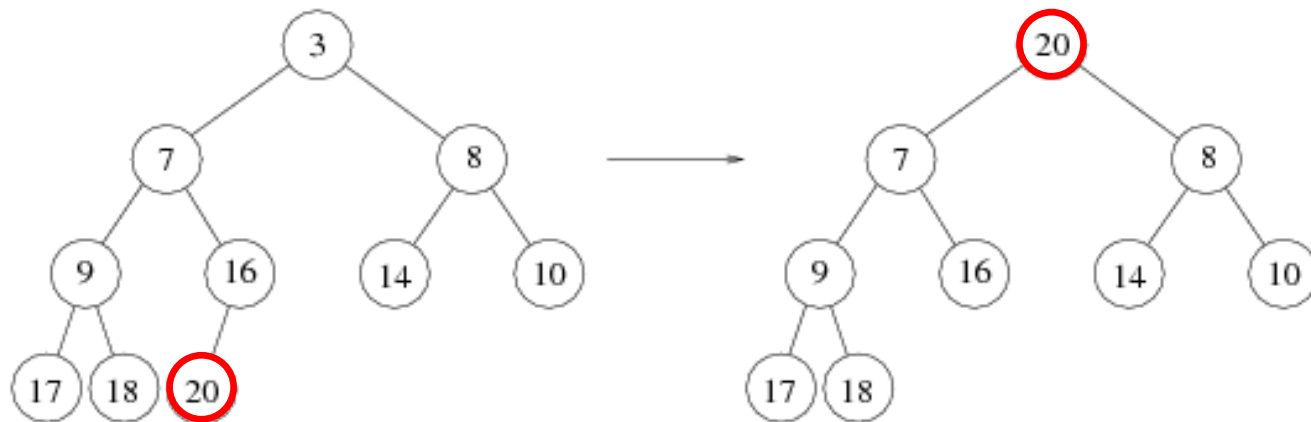


Heap property is preserved, but completeness is not preserved!

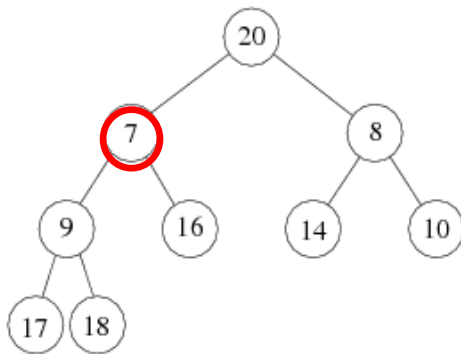


deleteMin: The second attempt

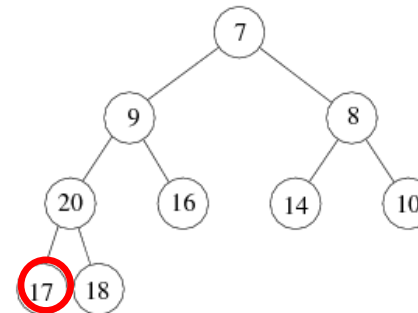
1. Copy the last number to the root (i.e. overwrite the minimum element stored there)
2. Restore the min-heap property by percolate down (or bubble down)



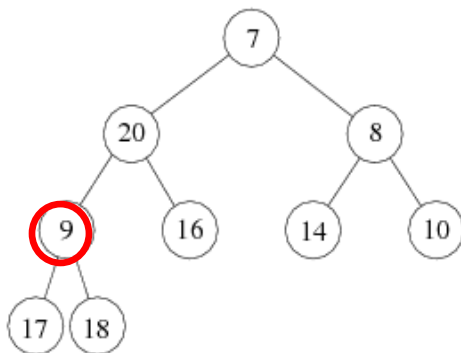
mn-heap property destroyed



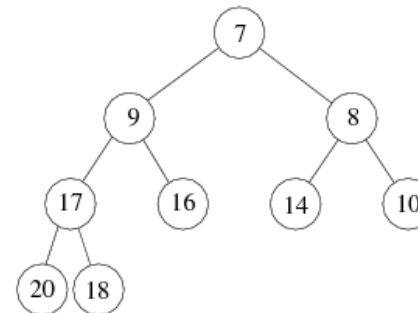
compare 20 with 7, the
smallest of its two children



compare 20 with 17, the
smallest of its two children



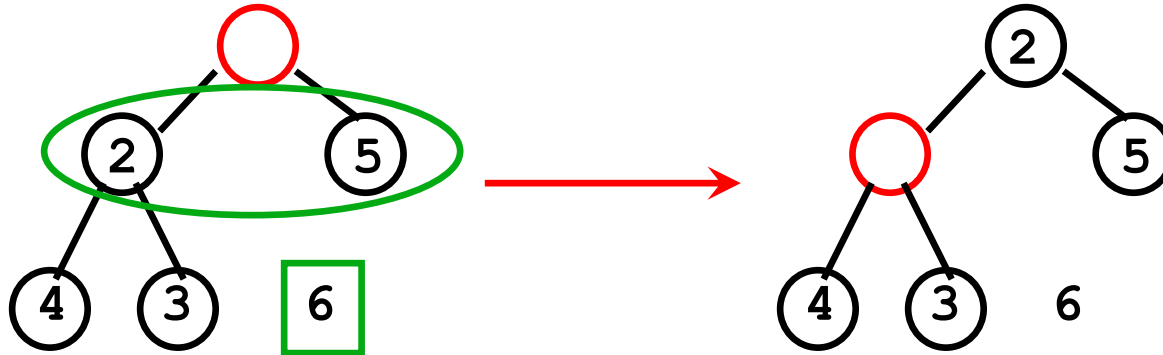
compare 20 with 9, the
smallest of its two children



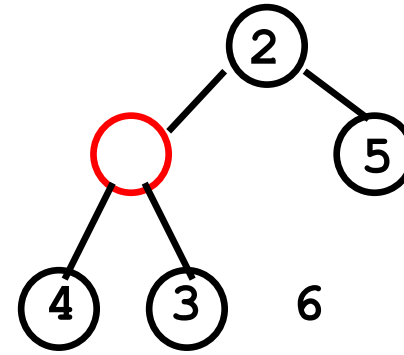
A heap!
Time complexity
 $= O(\text{height}) = O(\log n)$

The same 'hole' trick used in insertion can be used here too

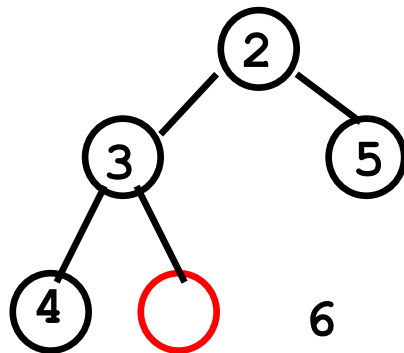
deleteMin with 'Hole Trick'



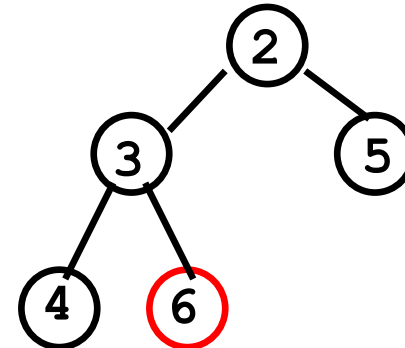
1. create hole
tmp = 6 (last element)



2. Compare children and tmp
bubble down if necessary



3. Continue step 2 until
reaches lowest level



4. Fill the hole

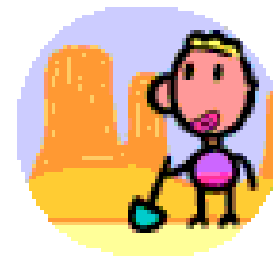
deleteMin PseudoCode

```
void deleteMin(){
    if (isEmpty()) throw UnderflowException();
    //copy the last number to the root, decrease array size by 1
    array[1] = array[currentSize- -]
    percolateDown(1); //percolateDown from root
}

void percolateDown(int hole) { //int hole is the root position
    int child;
    Comparable tmp = array[hole]; //create a hole at root
    for( ; hole*2 <= currentSize; hole=child){ //identify child position
        child = hole*2;
        //compare left and right child, select the smaller one
        if (child != currentSize && array[child+1] < array[child])
            child++;
        if(array[child] < tmp) //compare the smaller child with tmp
            array[hole] = array[child]; //bubble down if child is smaller
        else
            break; //bubble stops movement
    }
    array[hole] = tmp; //fill the hole
}
```

Heapsort

- (1) Build a binary heap of N elements
 - ◆ the minimum element is at the top of the heap
- (2) Perform N DeleteMin operations
 - ◆ the elements are extracted in sorted order
- (3) Record these elements in a second array and then copy the array back



Build Heap

- **Input:** N elements
- **Output:** A heap with heap-order property
- **Method 1:** obviously, N successive insertions
- **Complexity:** $O(N \log N)$

Heapsort – Running Time Analysis

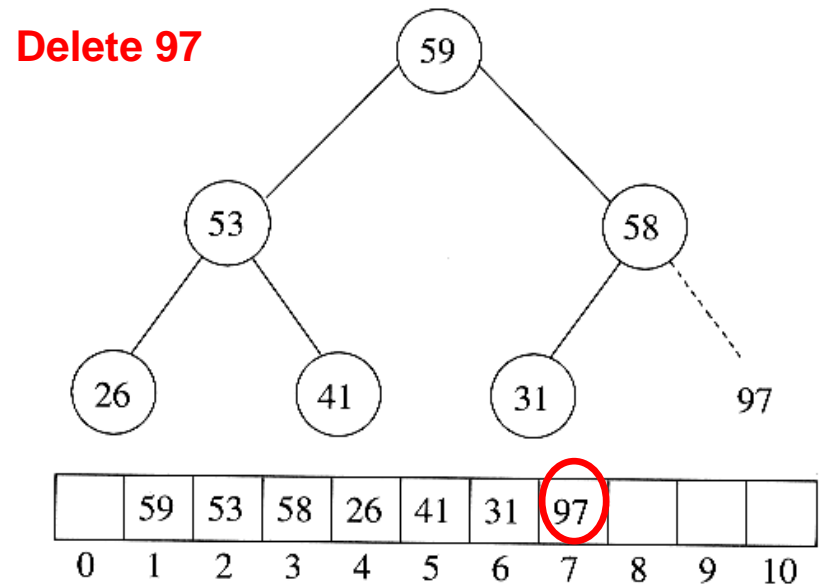
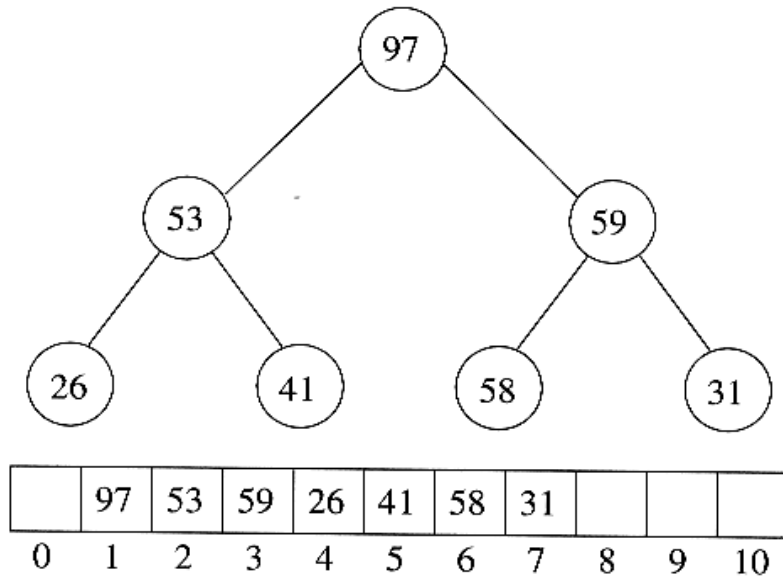
- (1) Build a binary heap of N elements
 - ◆ repeatedly insert N elements $\Rightarrow O(N \log N)$ time
(there is a more efficient way, check textbook p223 if interested)
 - (2) Perform N DeleteMin operations
 - ◆ Each DeleteMin operation takes $O(\log N) \Rightarrow O(N \log N)$
 - (3) Record these elements in a second array and then copy the array back
 - ◆ $O(N)$
- Total time complexity: $O(N \log N)$
 - Memory requirement: uses an extra array, $O(N)$

Heapsort: No Extra Storage

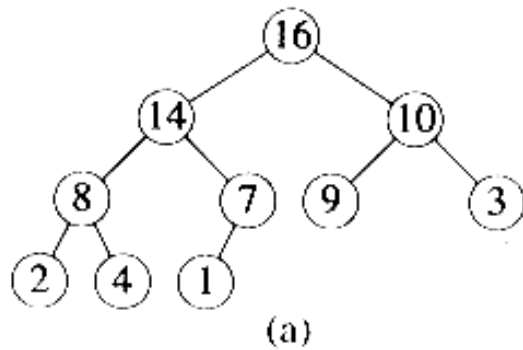
- Observation: after each deleteMin, the size of heap shrinks by 1
 - ◆ We can use the last cell just freed up to store the element that was just deleted
 - ⇒ after the last deleteMin, the array will contain the elements in decreasing sorted order
- To sort the elements in the decreasing order, use a minimum heap
- To sort the elements in the increasing order, use a maximum heap
 - ◆ the parent has a larger element than the child

Heapsort Example: No Extra Storage

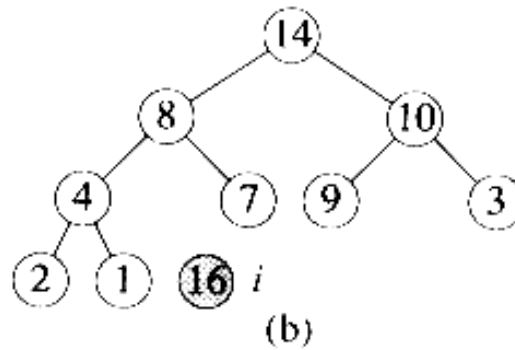
Sort in increasing order: use maximum heap



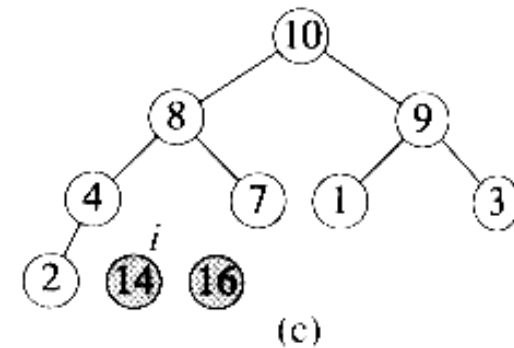
Another Heapsort Example



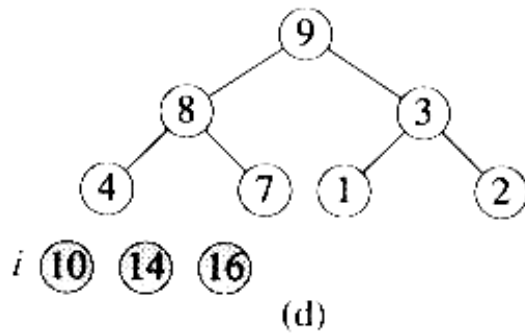
Delete 16



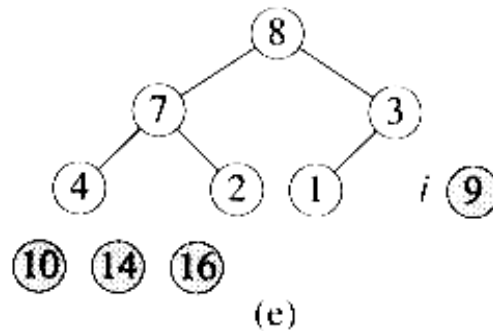
Delete 14



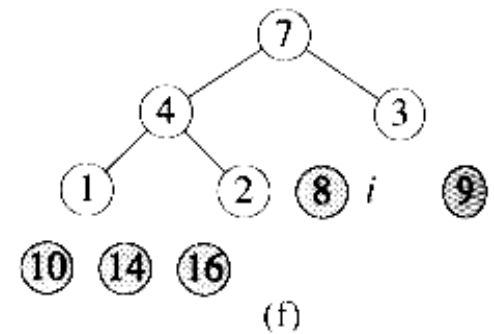
Delete 10



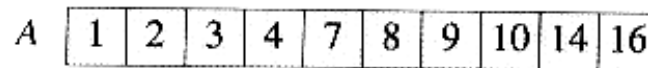
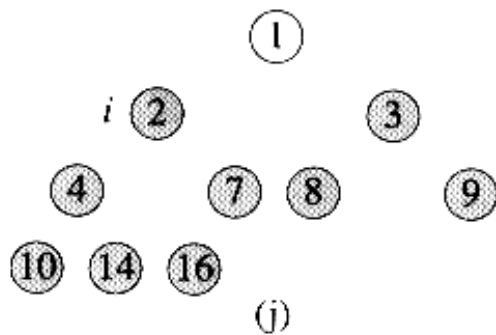
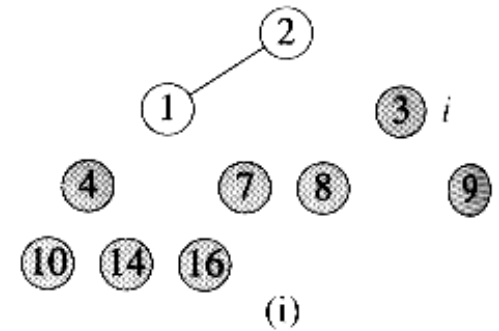
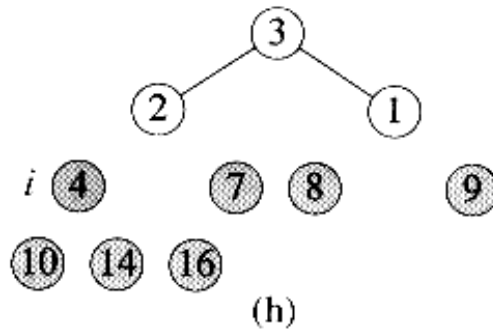
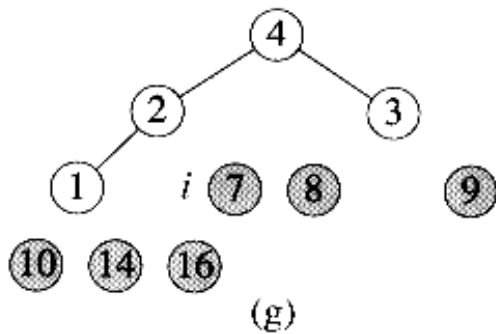
Delete 9



Delete 8



Example (cont'd)



(k)

Preliminary Heap ADT

```
class BinaryHeap{
public:
    BinaryHeap(int capacity=100);
    explicit BinaryHeap(const vector &items);

    bool isEmpty() const;

    void insert(const float &x);
    void deleteMin();
    void deleteMin(float &minItem);
    void makeEmpty();

private:
    int currentSize; //number of elements in heap
    vector<float> array; //the heap array

    void buildHeap();
    void percolateDown(int hole);
}
```