Lecture 13: Binary Heaps – Priority Queues

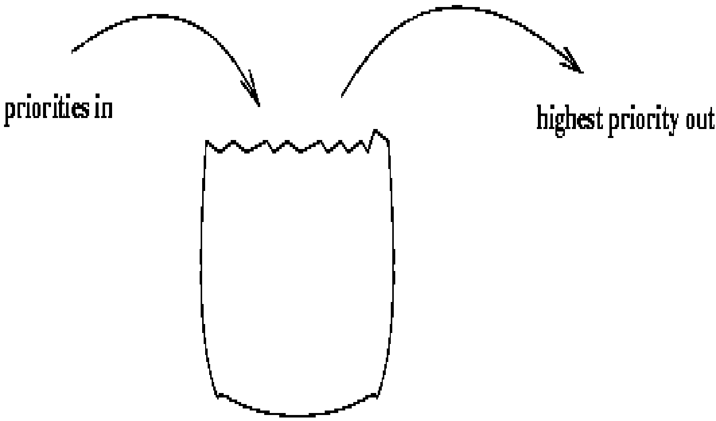
Recall Queues

- FIFO: **First-In, First-Out**

- Some contexts where this seems right?

- Some contexts where some things should be allowed to **skip ahead** in the line?

- Development of a data structure which allows efficient inserts and efficient deletes of the **minimum value** (minheap) or **maximum value** (maxheap)



Applications of the Priority Queue

- Select print jobs in order of decreasing **length**
- Forward packets on routers in order of **urgency**
- Select most frequent symbols for **compression**
- Sort numbers, picking **minimum first**

- Anything ***greedy***

Priority Queues: Specification

- **Main operations:**
  - Insert (i.e., enqueue)
    - Dynamic insert
    - specification of a priority level (0-high, 1,2.. Low)
  - deleteMin (i.e., dequeue)
    - Finds the current minimum element (read: “highest priority”) in the queue, deletes it from the queue, and returns it

Simple Implementations

- **Simple linked list:**
  - Insertion at the front (O(1)); delete minimum (O(N)), or
  - Keep list sorted; insertion O(N), deleteMin O(1)

- **Binary search tree:**
  - This gives an O(log N) average for both operations
  - But BST class supports a lot of operations that are not required

- **Array: Binary Heap**
  - Does not require links and will support both operations in O(logN) wost-case time

Binary Heap

- A **binary heap** is a **heap data structure** created by using a **binary tree**
- It is a binary tree with **two** additional **constraints**:

1. Shape property:

- A binary heap is a ***complete binary tree***
- All levels of the tree, except possibly the last one (deepest) are fully filled
- If the last level of the tree is not complete, the nodes of that level are filled from left to right.

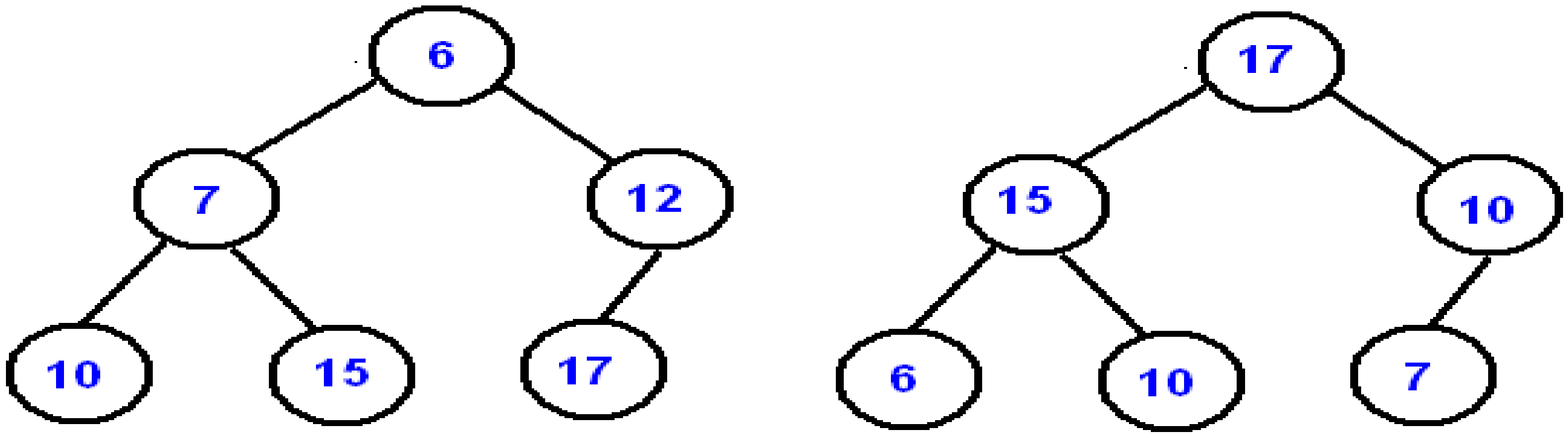
2. Heap property:

- All nodes are *either* ***greater than or equal to*** *or* ***less than or equal to*** *each of its children, according to a comparison predicate defined for the heap.*

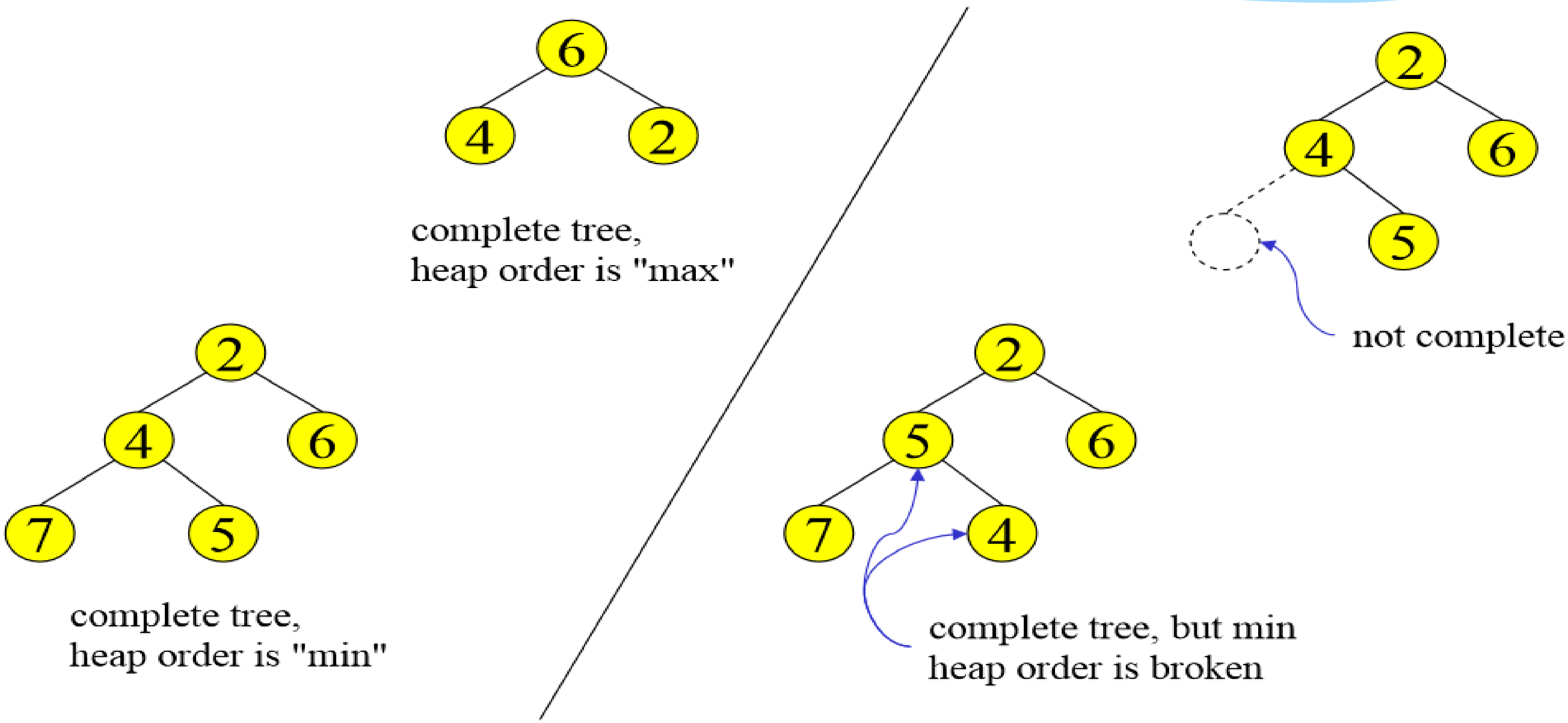
Heap Ordering Property

- ***min-heap property*:**
  - Value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root

- ***max-heap property*:**
  - Value of each node is less than or equal to the value of its parent, with the maximum-value element at the root

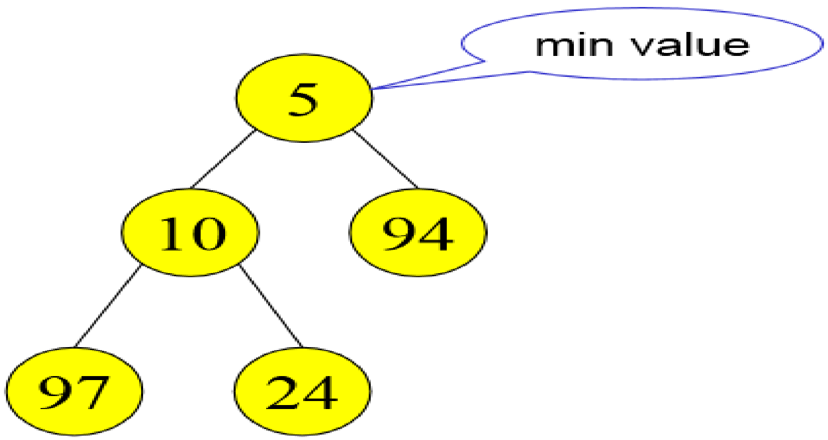


Examples


6
4
2
complete tree,
heap order is "max"
2
4
6
5
not complete
2
4
6
7
5
complete tree,
heap order is "min"
2
5
6
7
4
complete tree, but min
heap order is broken

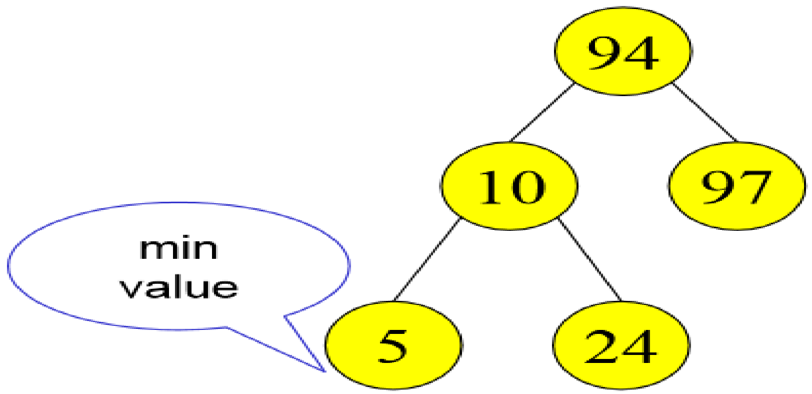
Binary Heap vs Binary Search Tree

Binary Heap



Parent is less than both left and right children

Binary Search Tree



Parent is greater than left child, less than right child

- They may look similar, but the ordering is very different
- The BST may contain some blank spots
- For a BST, the height h may vary
- You can traverse all the nodes of a tree
- In a heap, you can only look at the root

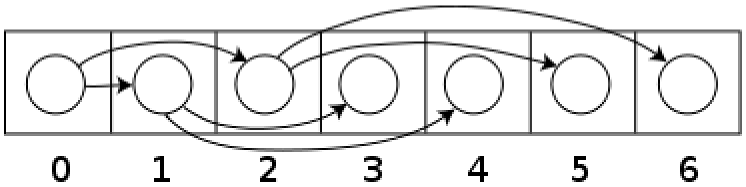
Heap Applications

- Sorting (HeapSort)

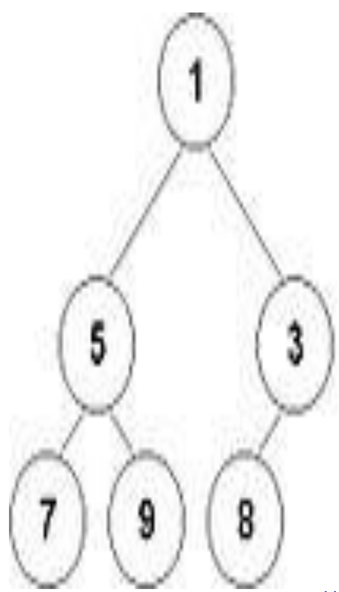
- Operating system scheduling (**priority queue**)
  - Process jobs by priority

- Graph algorithms
  - Find shortest path

Binary Heap Representation With Array



A complete binary tree stored in an array



Node	1	5	3	7	9	8
Index	0	1	2	3	4	5

Scheme 1: Root at **index 0**

- Left at 2N + 1
- Right at 2N + 2

Scheme 2: Root at **index 1**

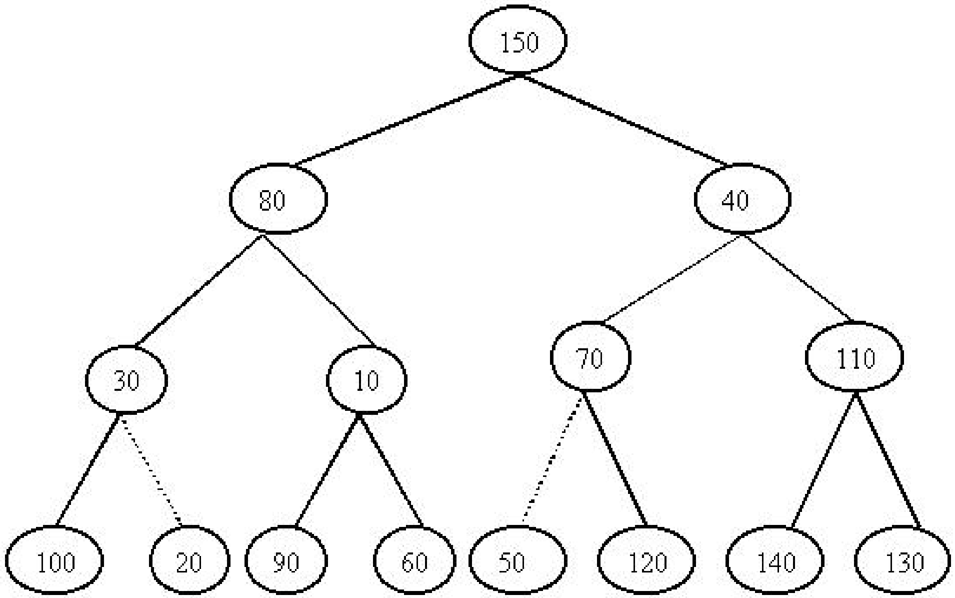
- Left at 2N
- Right at 2N + 1

Building a Heap

- **The idea:**
  - **Given an array of elements to be inserted in the heap, treat the array as a heap with order property violated, and then do operations to fix the order property**

Let the array A be:

150, 80, 40, 30, 10, 70, 110, 100, 20, 90, 60, 50, 120, 140, 130



- Fix order property, compare the nodes with their children starting with the rightmost node

a) **110** is less that its children - OK

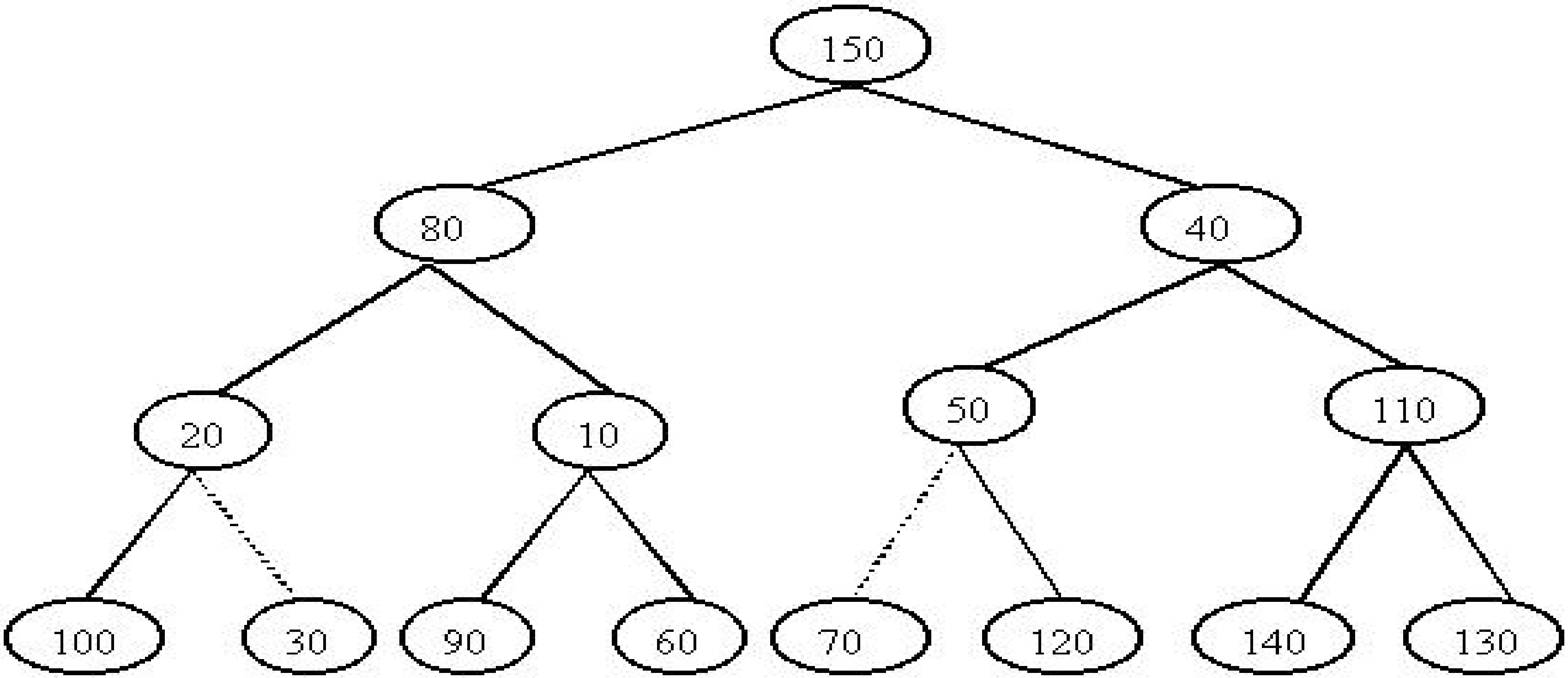
b) **70** is not less than its children . 50 is the node to go up one level.

c) **10** is OK

d) **30** is not less than its children, 20 is the node to go up

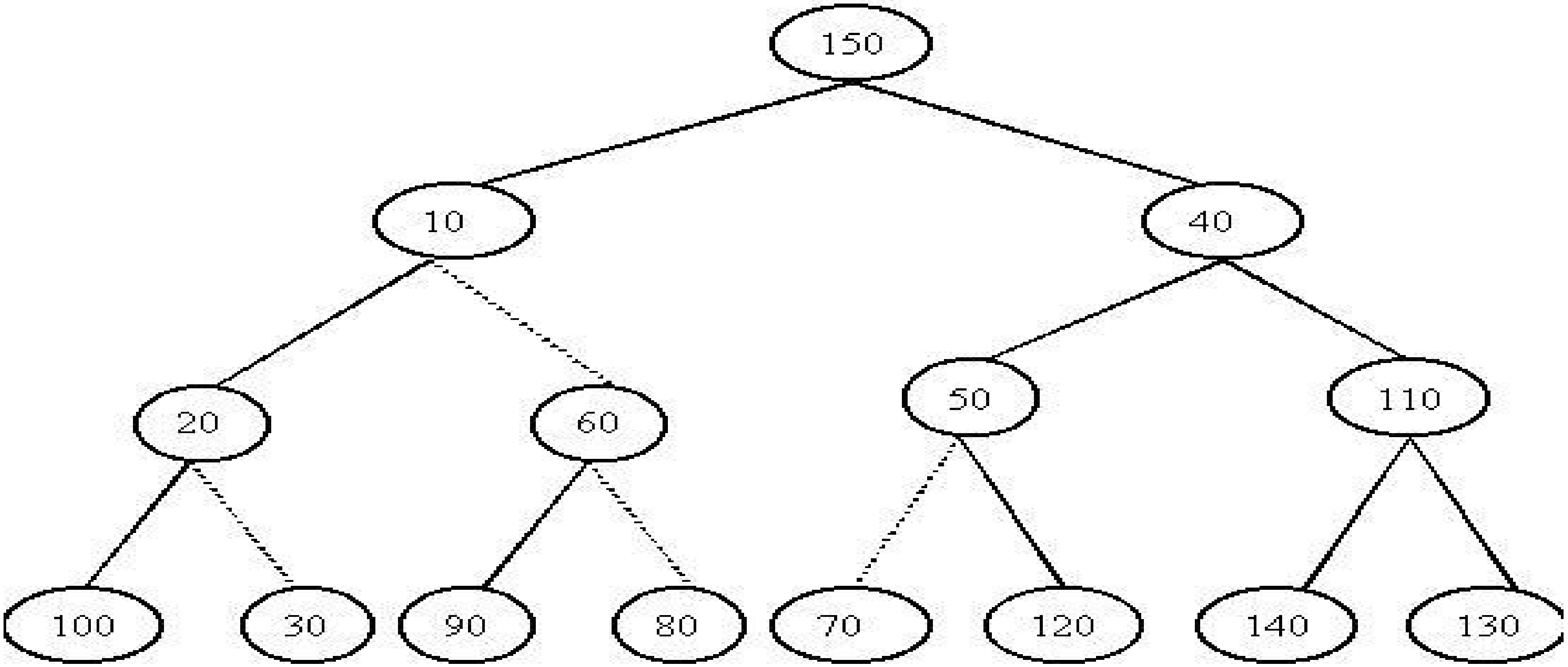
Building a Heap

- After processing :



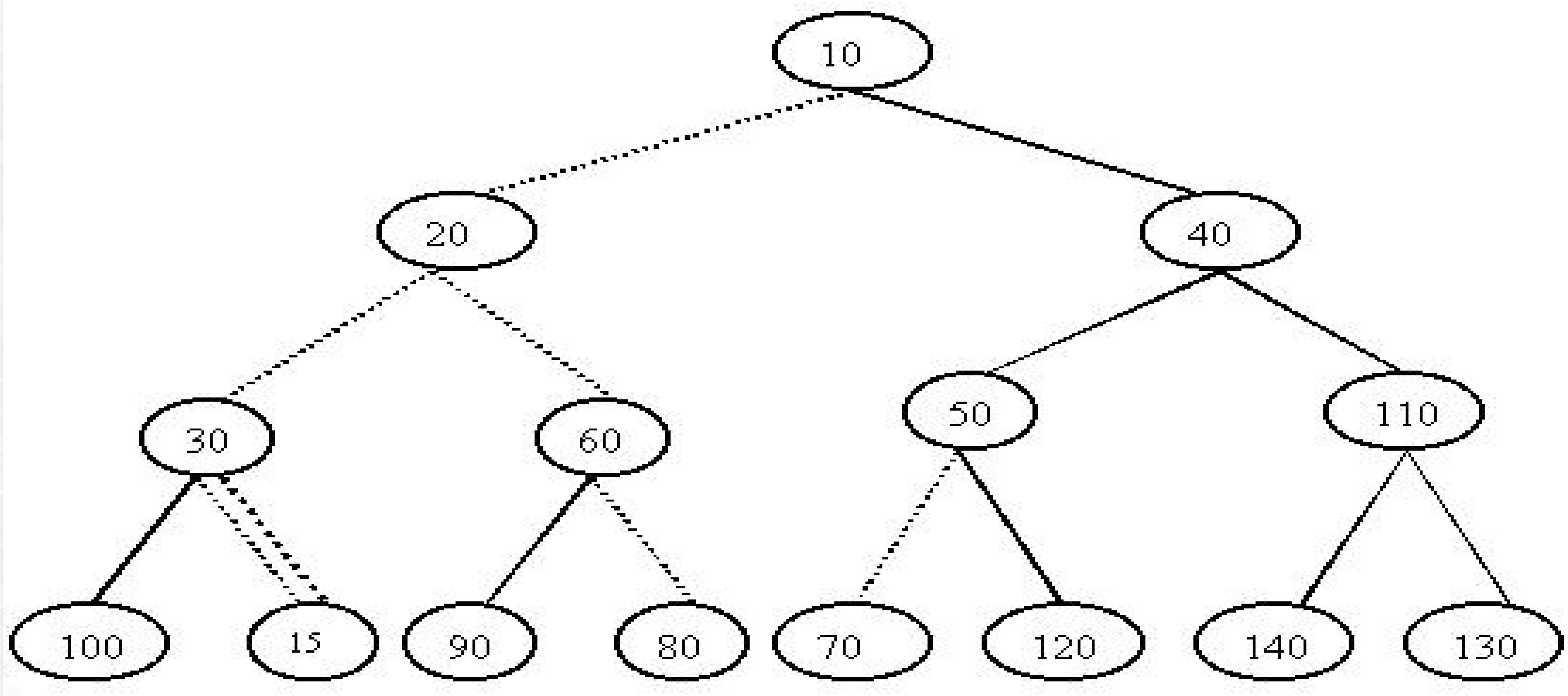
Building a Heap

a) **40** is OK
b) **80** needs to be percolated down twice



Building a Heap

a) **150** needs to be percolated down three times - until it gets to the bottom



Now the tree is in order

Insert

1) Add the element to the **bottom** level of the heap
2) **Compare** the added element with its **parent**; if they are in the correct order, stop
3) If not, **swap** the element with its **parent** and return to the previous step

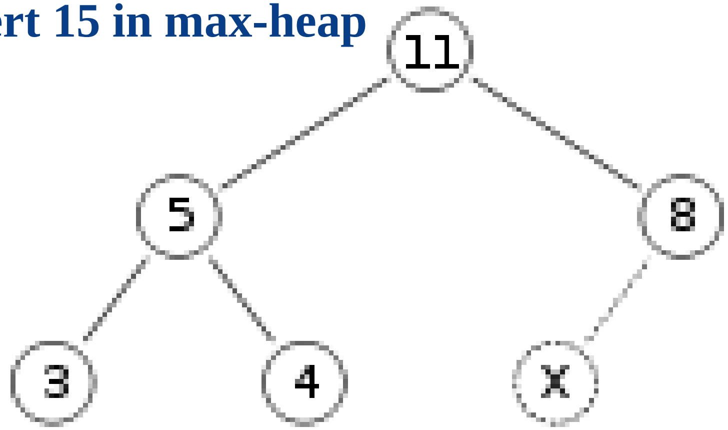
Insert function

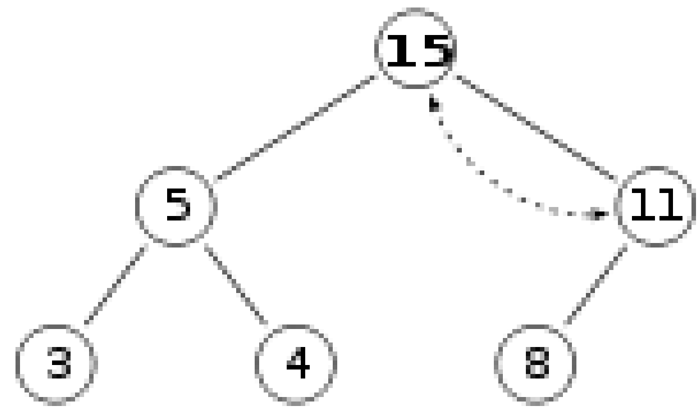
O(log N) time

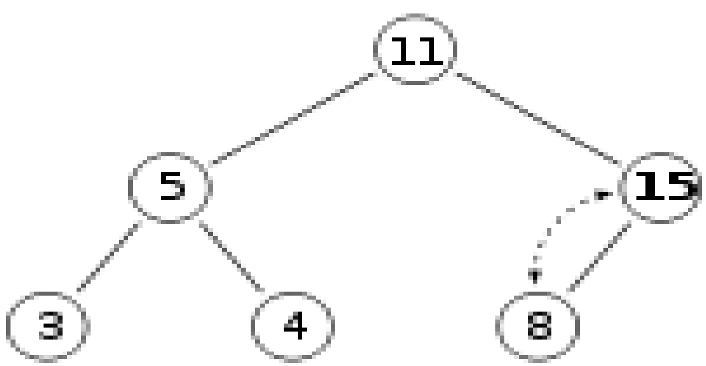
```
/**
 * Insert item x, allowing duplicates.
 */
void insert( const Comparable & x )
{
    if( currentSize == array.size( ) - 1 )
        array.resize( array.size( ) * 2 );

        // Percolate up
    int hole = ++currentSize;
    for( ; hole > 1 && x < array[ hole / 2 ]; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}
```

Insert - Example

Insert 15 in max-heap



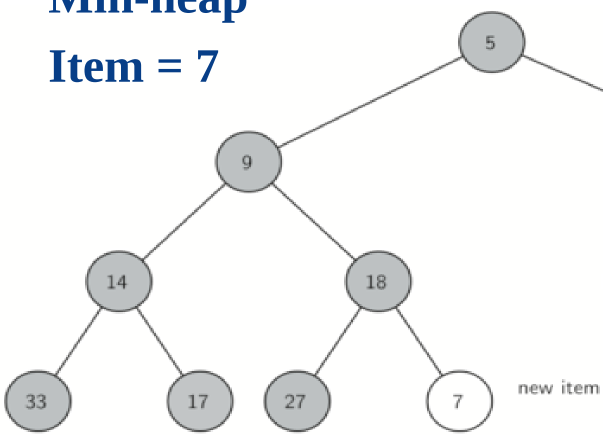


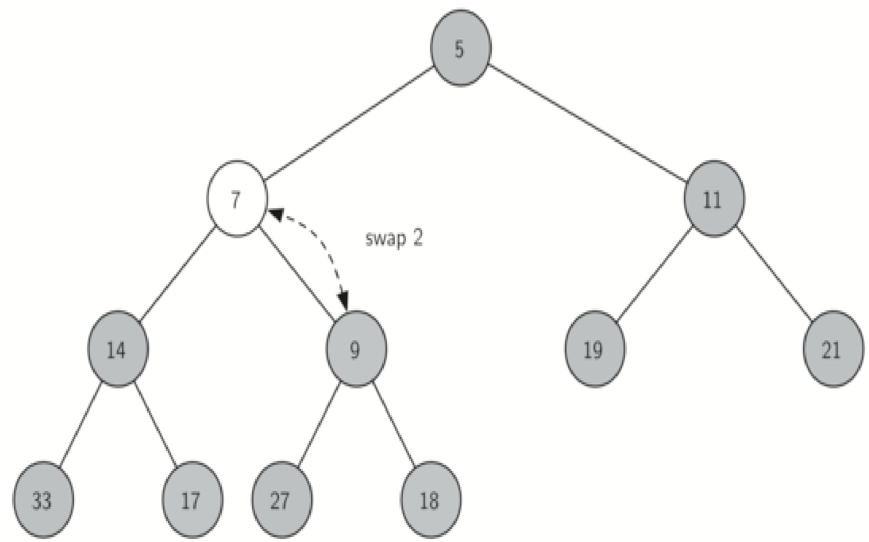


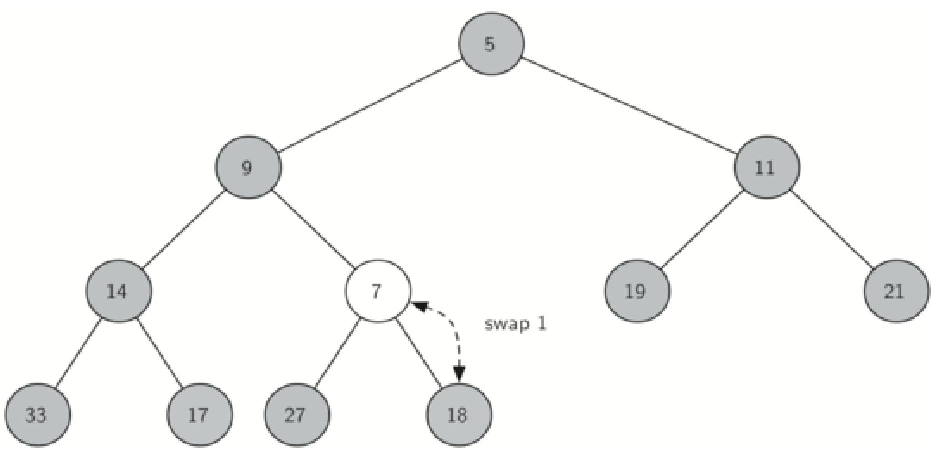
Insert - Example 1

Min-heap

Item = 7







Delete

1) Replace the **root** of the heap with the **last element** on the **last level**
2) **Compare** the new root with its **children**; if they are in the correct order, stop
3) If not, **swap** the element with one of its children and return to the previous step. (**Swap** with its **smaller child** in a **min-heap** and its **larger child** in a **max-heap**)

Delete function

```
/**
 * Remove the minimum item.
 * Throws UnderflowException if empty.
 */
void deleteMin( )
{
    if( isEmpty( ) )
        throw UnderflowException( );

    array[ 1 ] = array[ currentSize-- ];
    percolateDown( 1 );
}
```

```
/**
 * Remove the minimum item and place it in minItem.
 * Throws UnderflowException if empty.
 */
void deleteMin( Comparable & minItem )
{
    if( isEmpty( ) )
        throw UnderflowException( );

    minItem = array[ 1 ];
    array[ 1 ] = array[ currentSize-- ];
    percolateDown( 1 );
}
```

Delete function

```
/**
 * Internal method to percolate down in the heap.
 * hole is the index at which the percolate begins.
 */
void percolateDown( int hole )
{
    int child;
    Comparable tmp = array[ hole ];

    for( ; hole * 2 <= currentSize; hole = child )
    {
        child = hole * 2;
        if( child != currentSize && array[ child + 1 ] < array[ child ] )
            child++;
        if( array[ child ] < tmp )
            array[ hole ] = array[ child ];
        else
            break;
    }
    array[ hole ] = tmp;
}
```

Percolate down

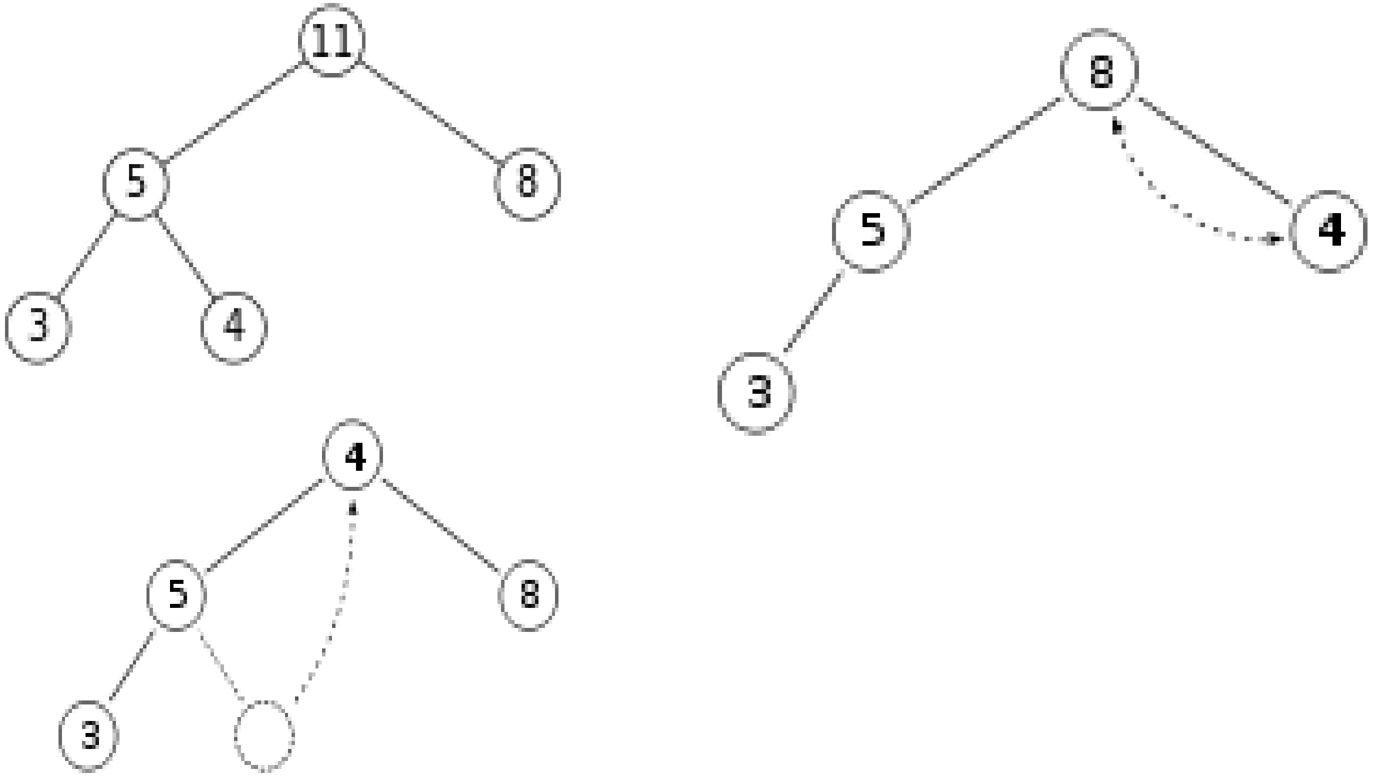
Left child

Right child

Pick child to swap with

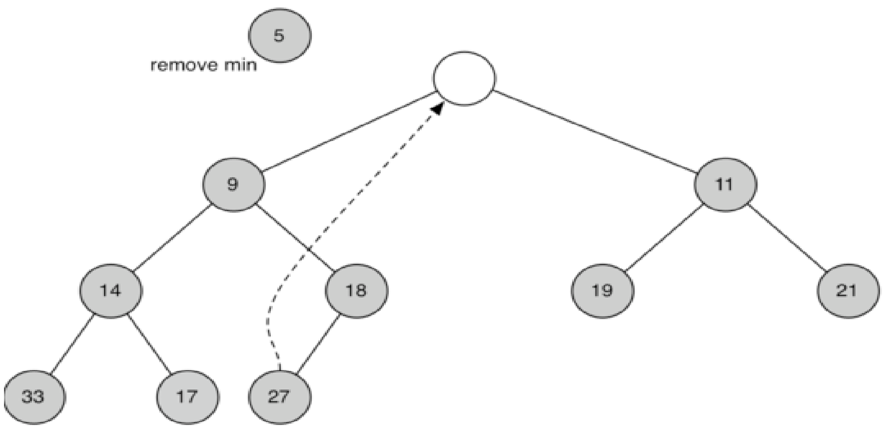
Delete - Example

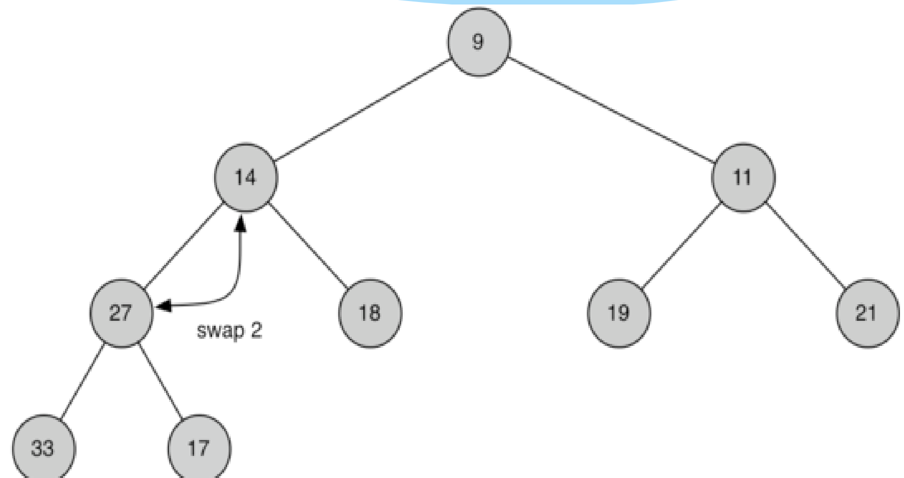
max-heap
remove 11

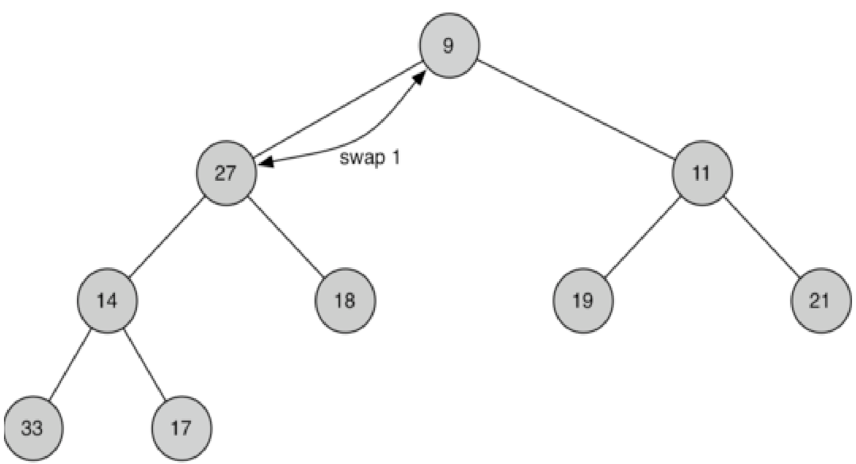


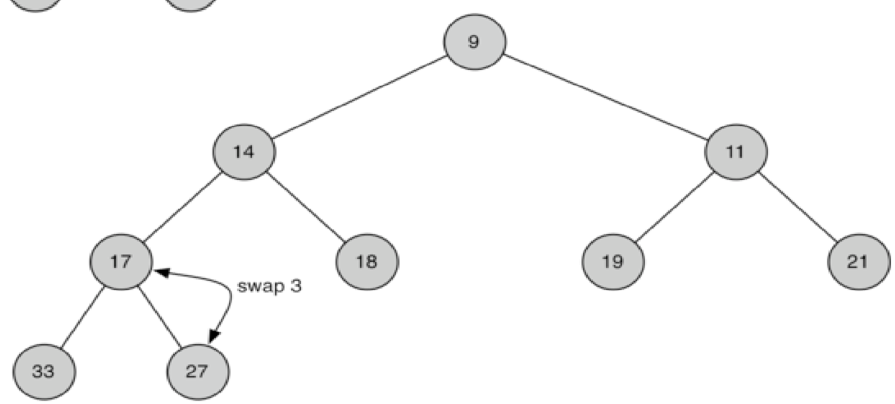
Delete - Example 1

min-heap









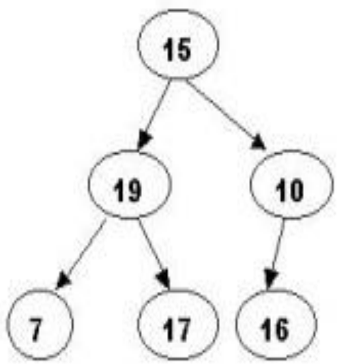
Heap Sort

- The heapsort algorithm can be **divided** into **two** parts:
  - **Step 1:**
    - A heap is built out of the data
  - **Step 2:**
    - Sorted array is created by **repeatedly removing** the **largest element** from the heap (the **root of the heap**) and inserting it into the array
    - Heap is **updated** after each **removal** to maintain the heap
    - Once **all** objects have been **removed** from the heap, the result is a **sorted array**

Heap Sort - Example

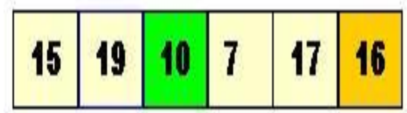
Array: **15, 19, 10, 7, 17, 16**

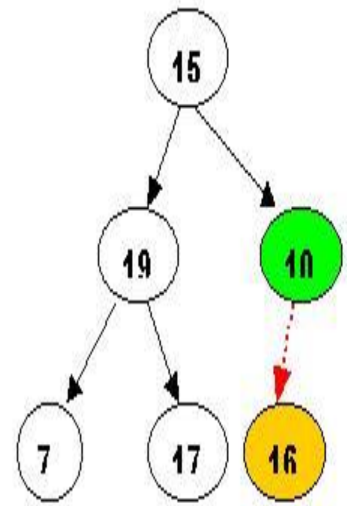
Step 1. Building the max-heap tree



The array represented as a tree, complete but not ordered

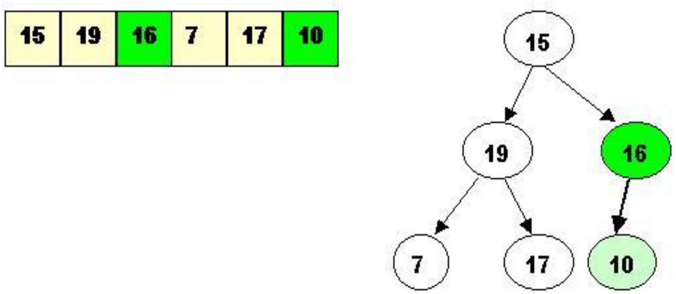
1.1 Percolate down

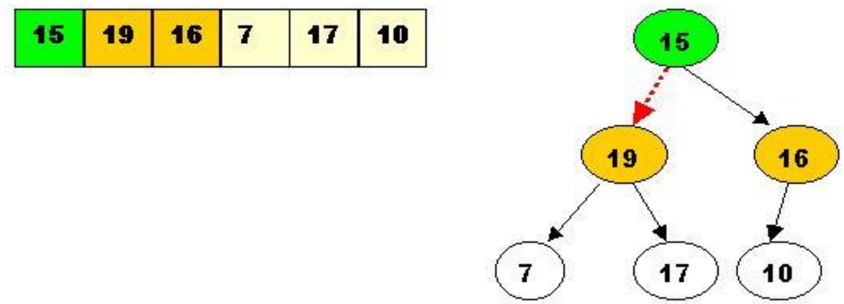


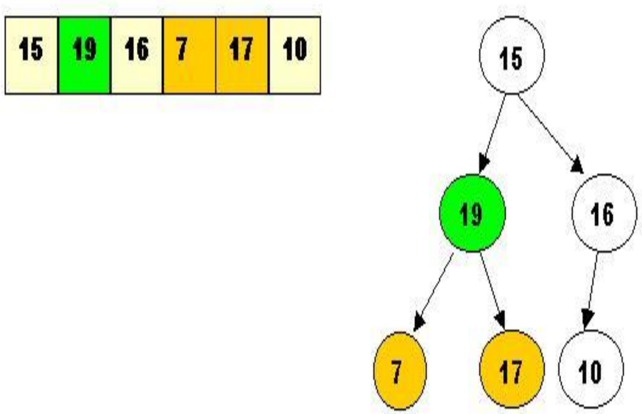


Heap Sort

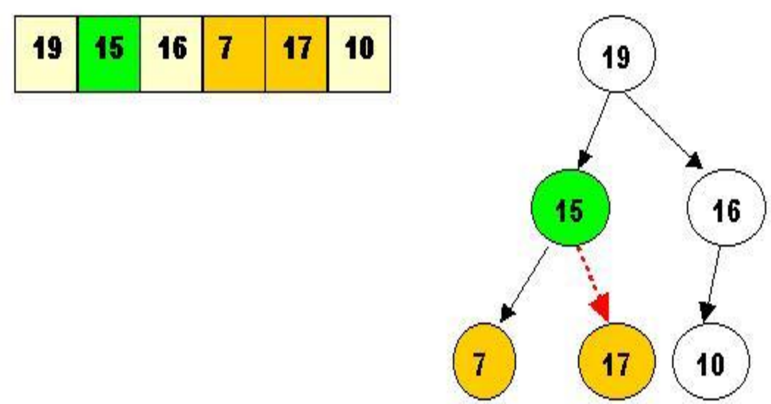
After processing array[3]





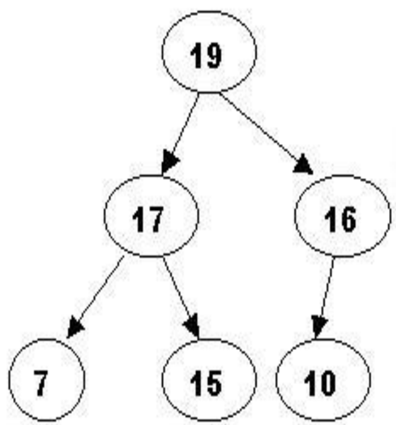


After processing



Heap Sort

19	17	16	7	15	10



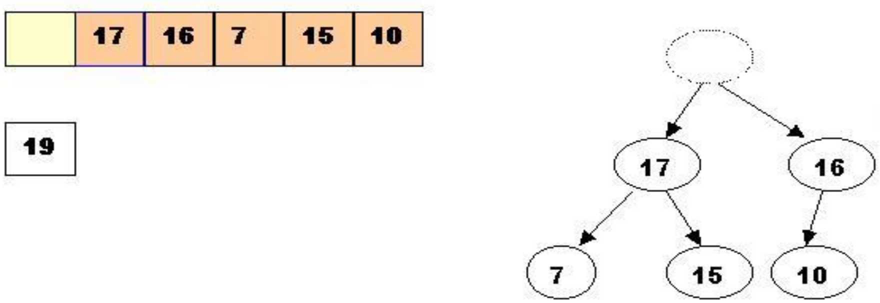
Now the tree is ordered, and the binary heap is built

Heap Sort

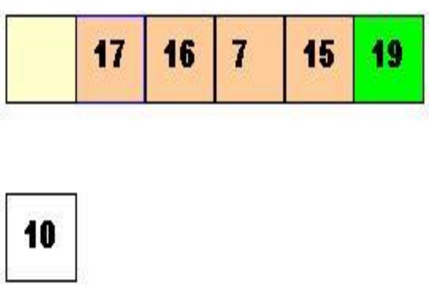
Step 2. Sorting - performing deleteMax operations:

2.1 Delete the top element 19

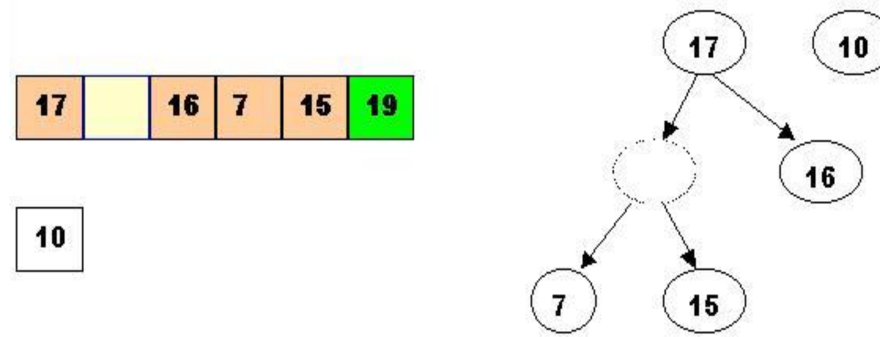
a. Store 19 in a temporary place



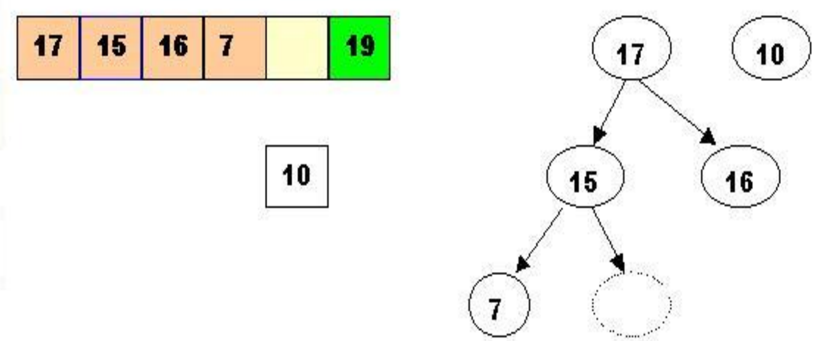
b. Swap 19 with the last element of the heap



c. Percolate down



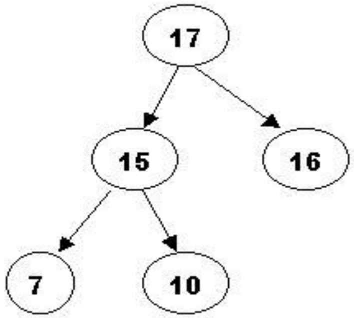
d. Percolate once more (**10** is less that **15**, so it cannot be inserted here



Heap Sort

Now **10** can be inserted

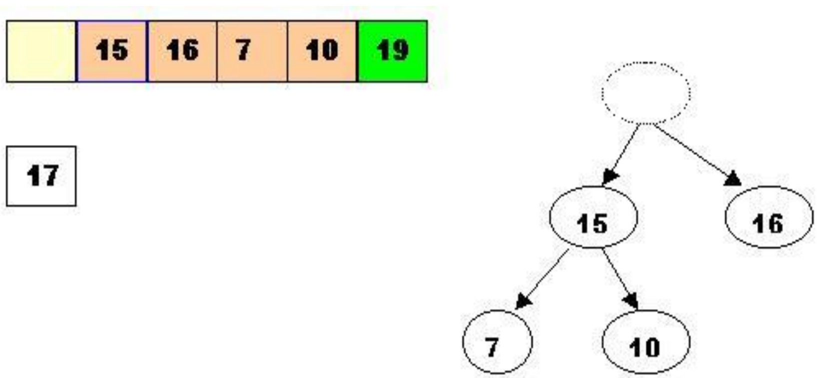
17	15	16	7	10	19



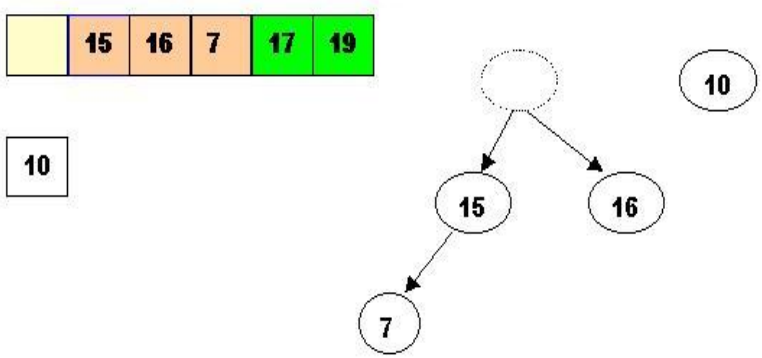
Heap Sort

2.2 DeleteMax the top element 17

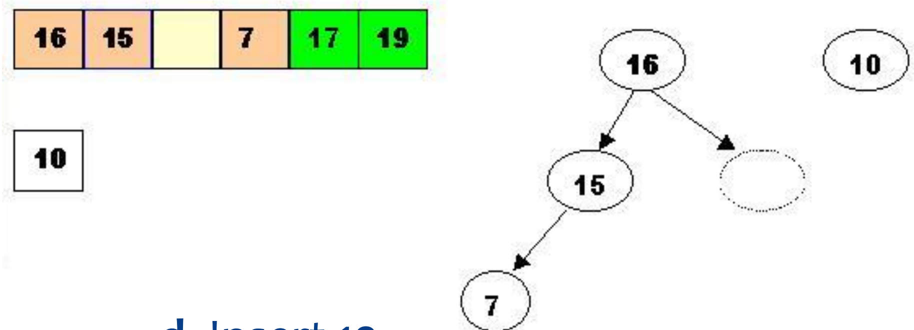
a. Store 17 in a temporary place



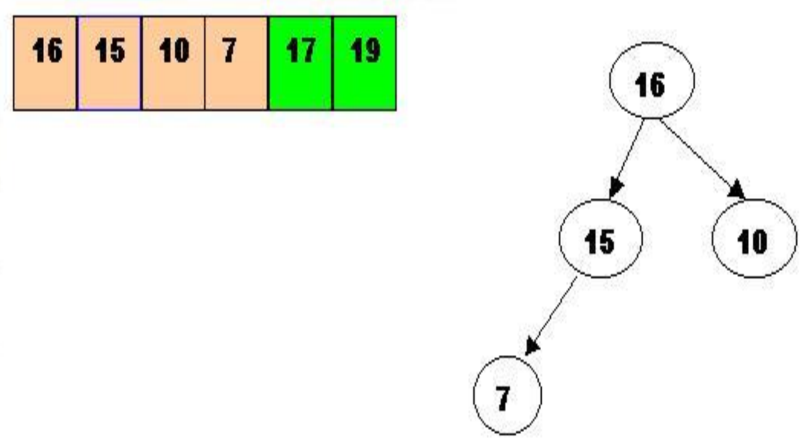
b. Swap 17 with last element of heap



c. Percolate down



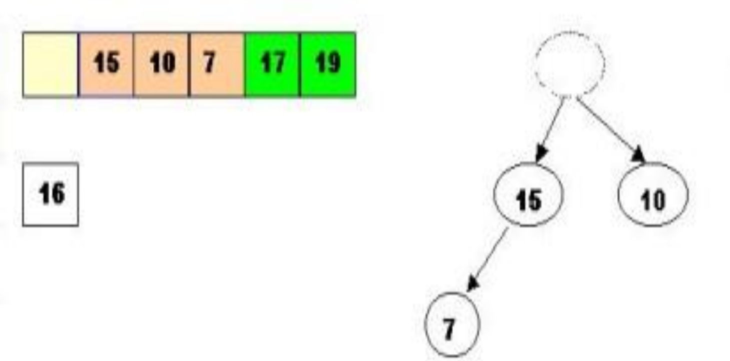
d. Insert **10**



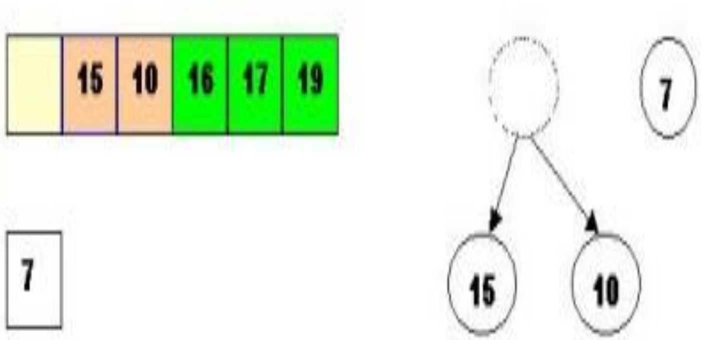
Heap Sort

2.3 DeleteMax 16

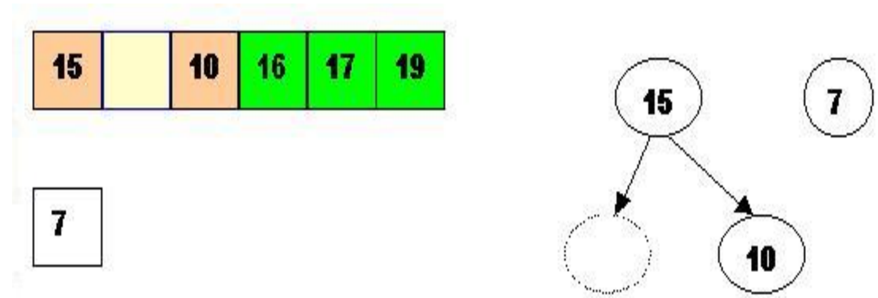
a. Store 16 in a temporary place



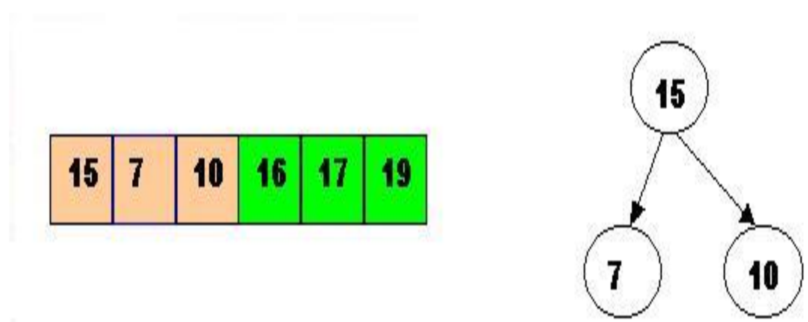
b. Swap 16 with last element of heap



c. Percolate down



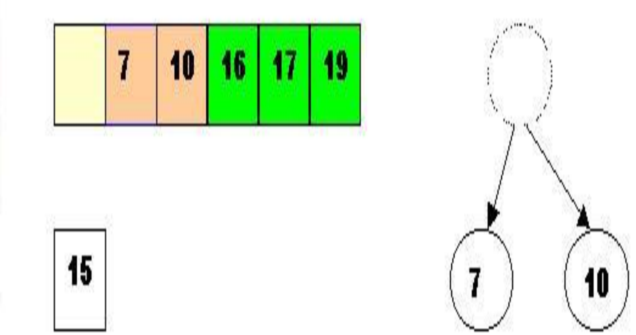
d. Insert **7**



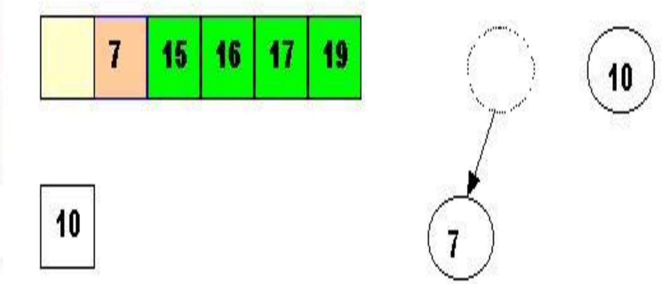
Heap Sort

2.4 DeleteMax the top element 15

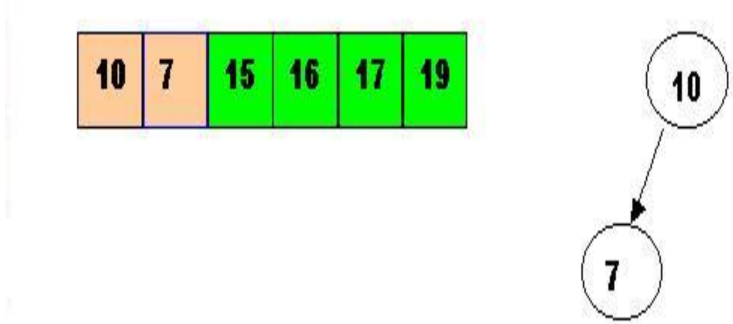
a. Store 15 in a temporary place



b. Swap 15 with last element of heap



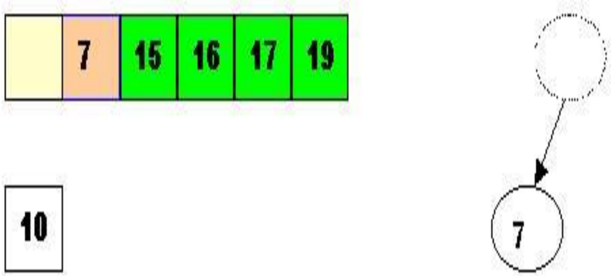
c. Insert **10**



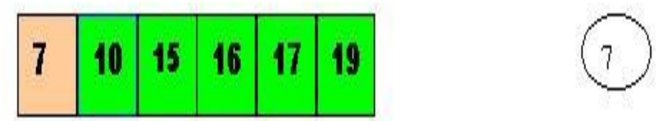
Heap Sort

2.5 DeleteMax the top element 10

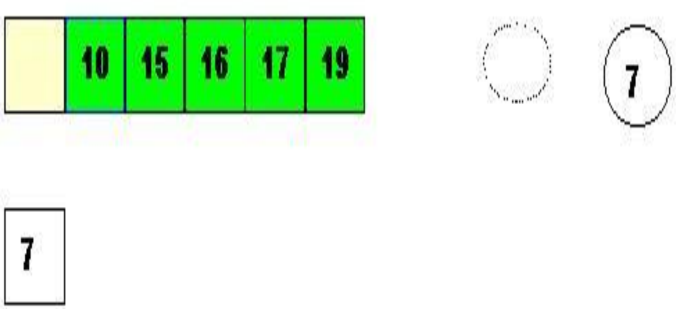
a. Remove 10 from the heap and store into temporary location



c. Insert **7**



b. Swap 10 with last element of heap



7 is the last element from the heap, so now the array is sorted