香港中文大學(深圳)
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 9: Binary search tree, Heap

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Outline

- Binary search tree operations
  - Insert
  - Delete
- Heap
  - Motivation
  - Priority queue
  - Binary heap
  - Insert & delete
  - HeapSort

Insert

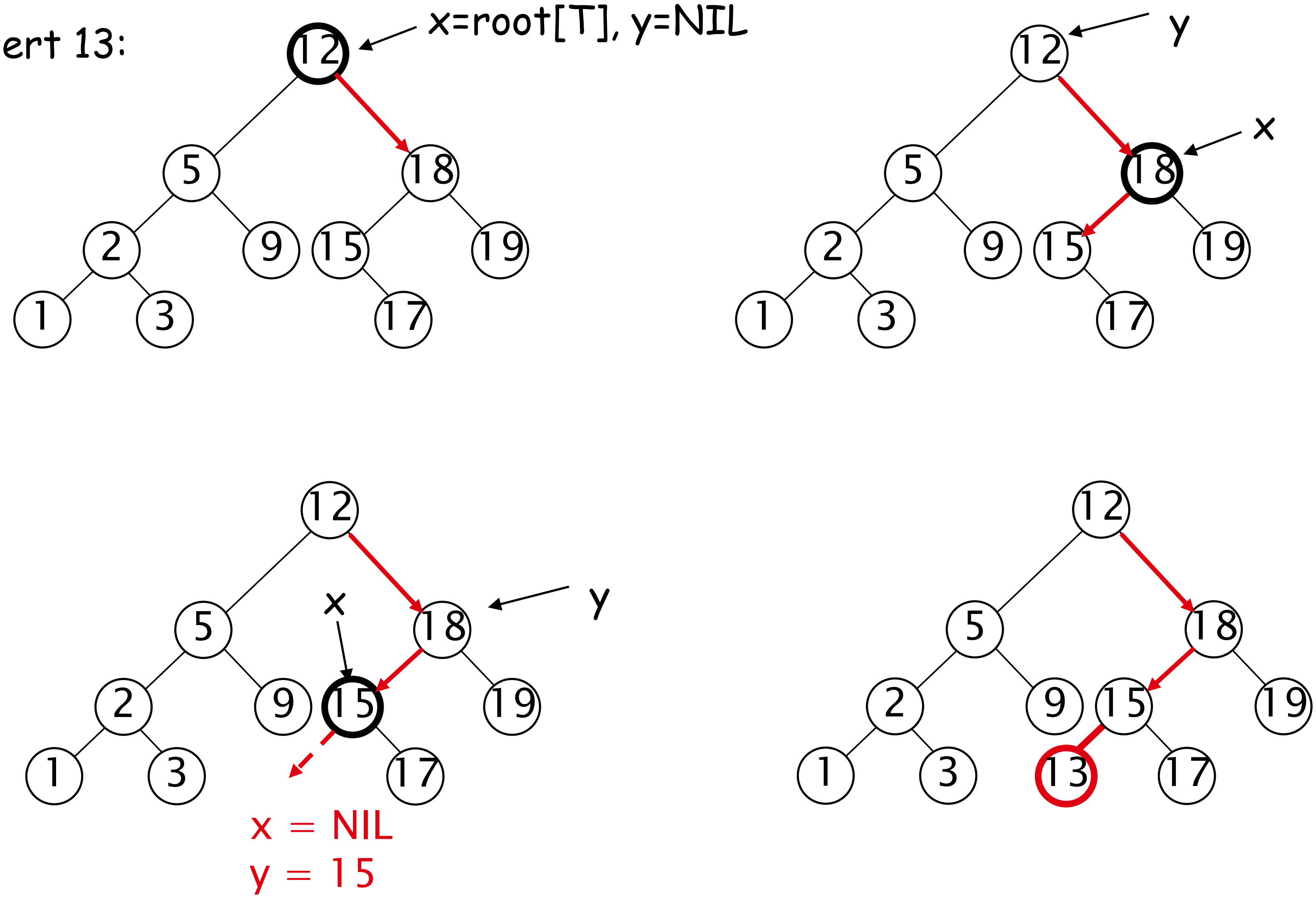
- Goal:
  - Insert value v into a binary search tree

- Find the position and insert as a leaf:
  - If key [x] < v move to the right child of x,
    else move to the left child of x
  - When x is NIL, we found the correct position
  - If v < key [y] insert the new node as y's left child

    else insert it as y's right child
  - Beginning at the root, go down the tree and maintain:
    - Pointer x : traces the downward path (current node)
    - Pointer y : parent of x ("trailing pointer")

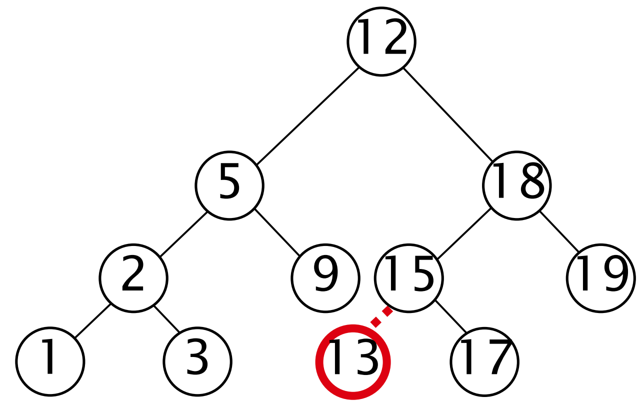
Example

Insert 13:



Insert algorithm

```
y ← NIL
x ← root [T]
while x ≠ NIL
      do y ← x
         if key [z] < key [x]
                then x ← left [x]
                else x ← right [x]
p[z] ← y
if y = NIL
     then root [T] ← z
     else if key [z] < key [y]
               then left [y] ← z
               else right [y] ← z
```



Tree T was empty

Running time: O(h)

Exercise 1

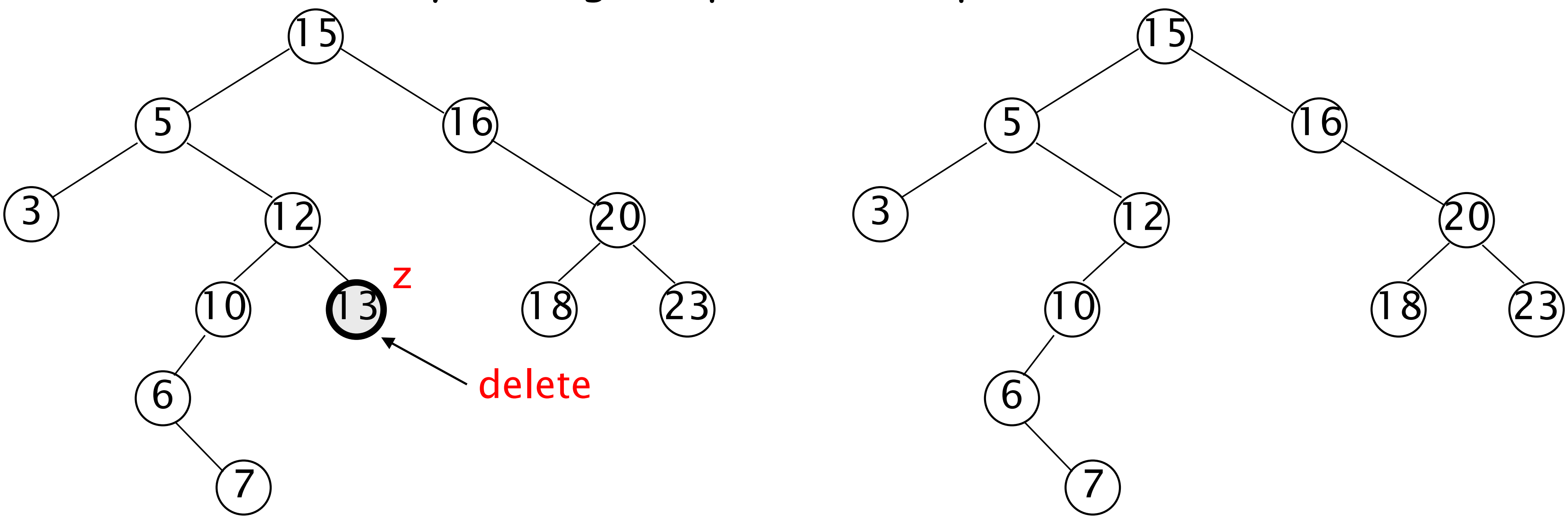
- Build a binary search tree for the following sequence

  15, 6, 18, 3, 7, 17, 20, 2, 4, 13, 9

- Build a binary search tree for the reverse of this sequence

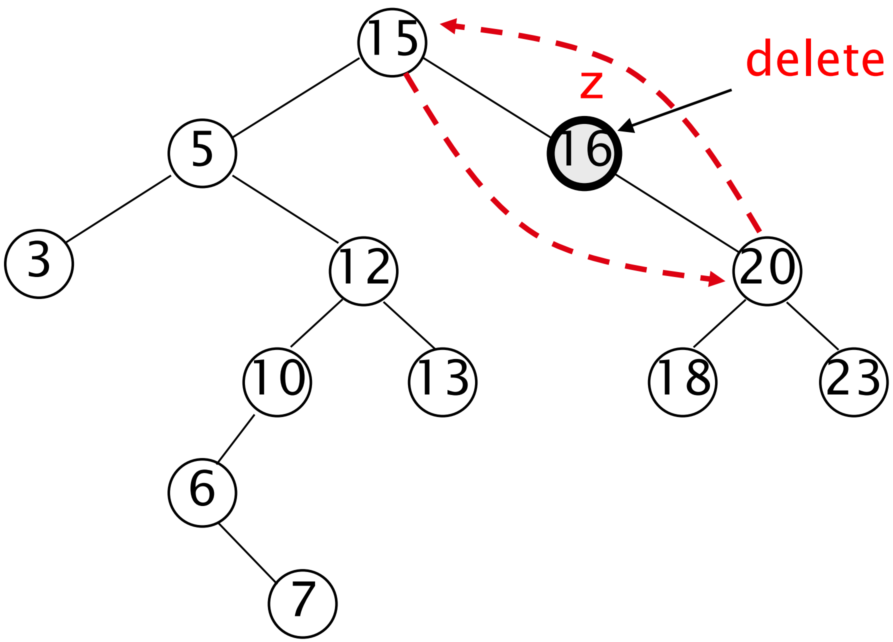
Deletion

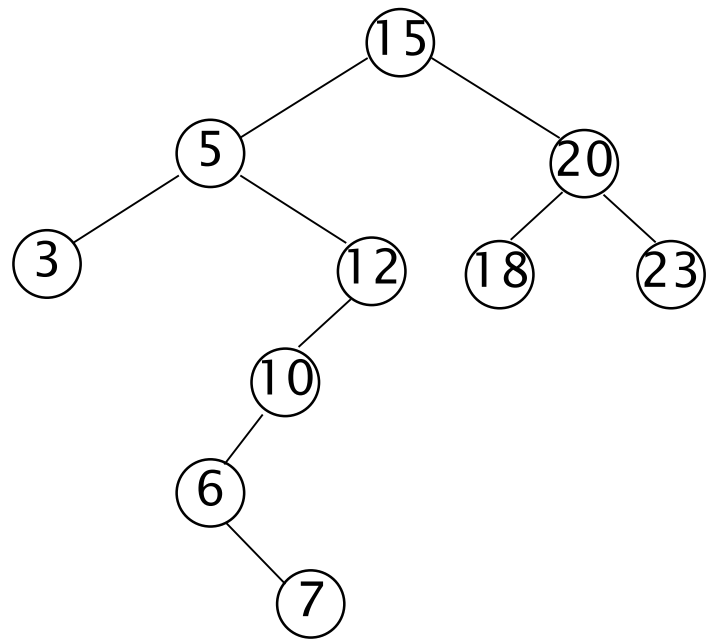
- Goal:
  - Delete a given node z from a binary search tree
- Idea:
  - **Case 1:** z has no children
    - Delete z by making the parent of z point to NIL



Deletion

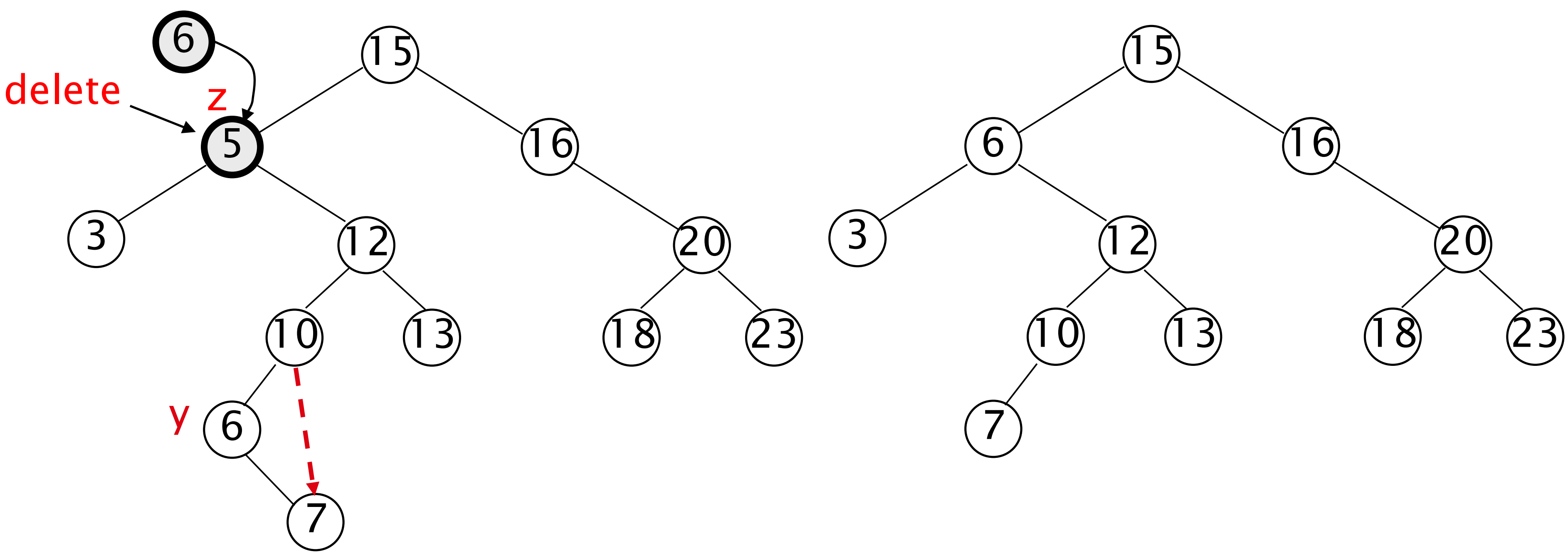
- **Case 2:** z has one child
  - Delete z by making the parent of z point to z's child, instead of to z and link the parent with the new child





Deletion

- **Case 3:** z has two children
  - Find z's *successor y* (the leftmost node in z's right subtree)
  - y has either no children or one right child (but no left child), why?
  - Delete y from the tree (via Case 1 or 2)
  - Replace z's key and satellite data with y's

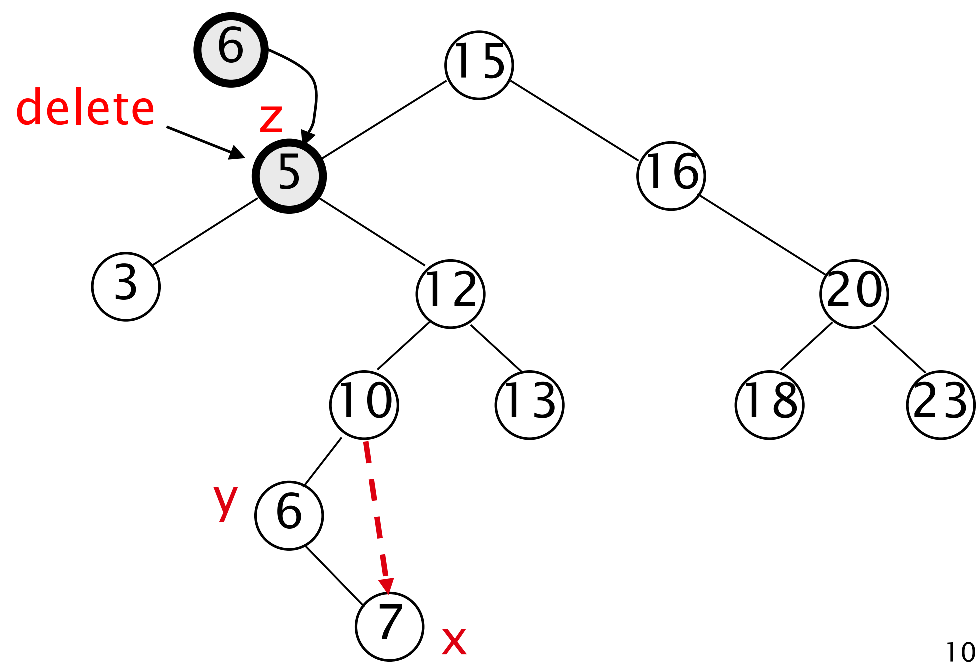


Deletion algorithm

```
if left[z] = NIL or right[z] = NIL
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ NIL
    then x ← left[y]
    else x ← right[y]
if x ≠ NIL
    then p[x] ← p[y]
```

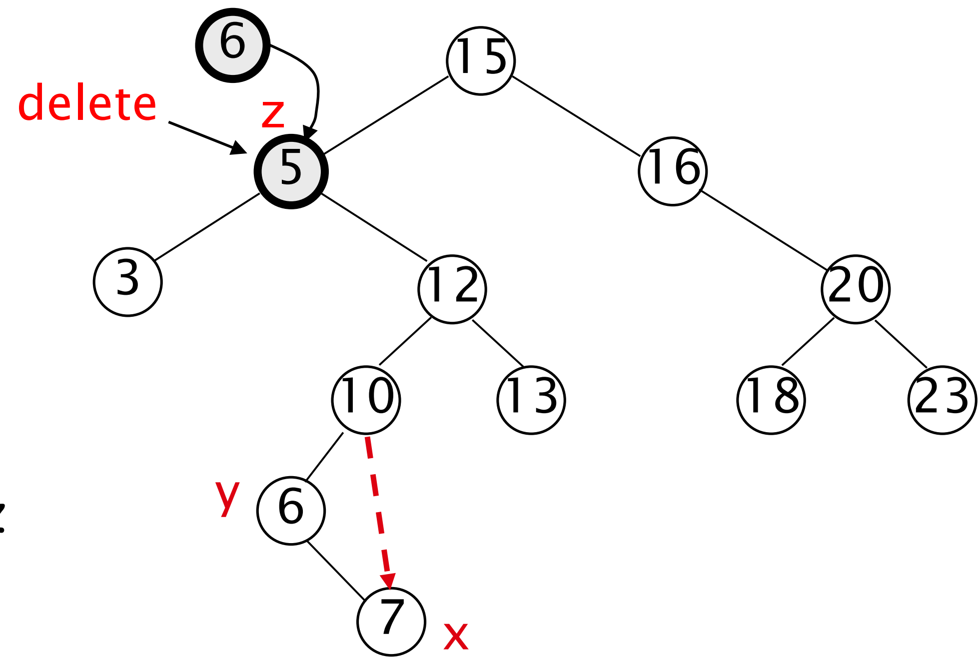
z has one child

z has 2 children



Deletion algorithm cont.

```
if p[y] = NIL
   then root[T] ← x
   else if y = left[p[y]]
           then left[p[y]] ← x
           else right[p[y]] ← x
if y ≠ z
   then key[z] ← key[y]
        copy y's satellite data into z
return y
```

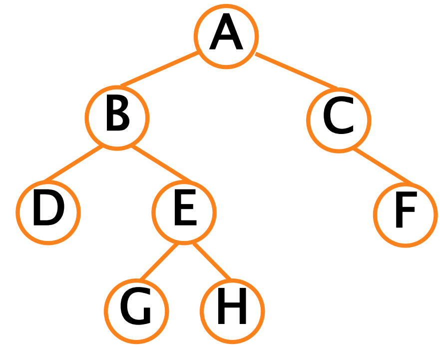


Running time: $O(h)$

Binary Tree Operations - height

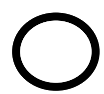
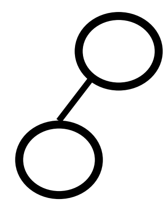
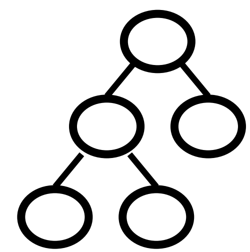
Review:

Depth of node : The depth of root(T) is zero.
The depth of any other node is one larger than his parent's depth



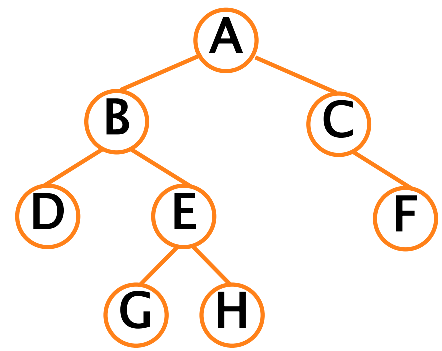
Height of a tree: The maximum depth of any leaf in the tree

Example:

Height of a NULL binary tree is 0	 Height of a tree with 1 node is 0	 Height = 1	 Height = 2

Binary Tree Operations - height

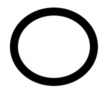
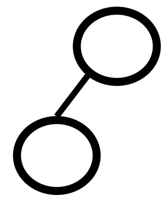
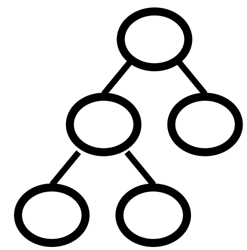
```
//To determine the height of a binary tree
int TreeNode height( )
{      int HeightOfLeftSubTree, HeightOfRightSubTree;

       if (this == NULL)
              return(0);

       if ((this->left == NULL) && (this->right == NULL))
              return(0); // the subroot is at level 0

       HeightOfLeftSubTree = this->left->height();
       HeightOfRightSubTree = this->right->height();

       if (_______________________________________)

              return_____________________________;
       else
              return_____________________________;
}
```




```
int Mytree::height()
{
       return root->height();
}
```

Binary Tree Operations - countleaves

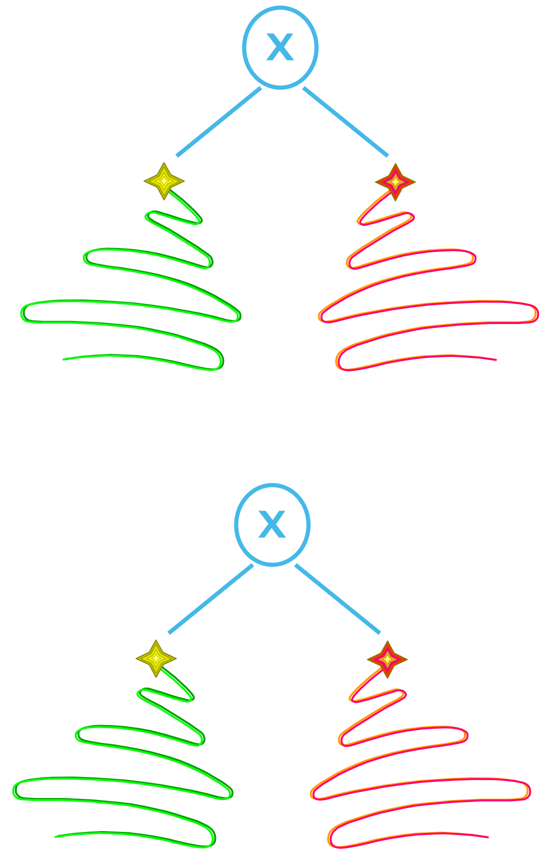
Example:

A NULL binary tree has 0 leaf node	 A tree with 1 node has 1 leaf node	 No. of leaf nodes = 1	 No. of leaf nodes = 3

```
//To count the number of leaf nodes

int Mytree::count_leaf(TreeNode* p)
{     if (p == NULL)
             return(0);
      else if ((p->left == NULL) && (p->right == NULL))
             return(1);
      else
             return(count_leaf(p->left) + count_leaf(p->right));
}
```

Binary Tree Operations - equal

```
// To compare 2 binary trees
boolean TreeNode::equal(TreeNode* TN)
{       if ((this == NULL) && (TN == NULL))
                return(true) ;
        if ((this != NULL) && (TN == NULL))
                return(false);
        if ((TN != NULL) && (this == NULL))
                return(false);
        if (this->info == TN->info)
                if (this->left->equal(TN->left) &&
                        this->right->equal(TN->right))
                            return(true);
        return(false);
}

boolean Mytree::equal(Mytree* T)
{ return root->equal(T->root);}
```



Binary Search Trees - Summary

- Operations on binary search trees:
  - SEARCH O(h)
  - PREDECESSOR O(h)
  - SUCCESOR O(h)
  - MINIMUM O(h)
  - MAXIMUM O(h)
  - INSERT O(h)
  - DELETE O(h)

- These operations are fast if the height of the tree is small - otherwise their performance is similar to that of a linked list

The issues in BST

- After a series of DELETION, the above algorithm favors making the left sub-trees deeper than the right

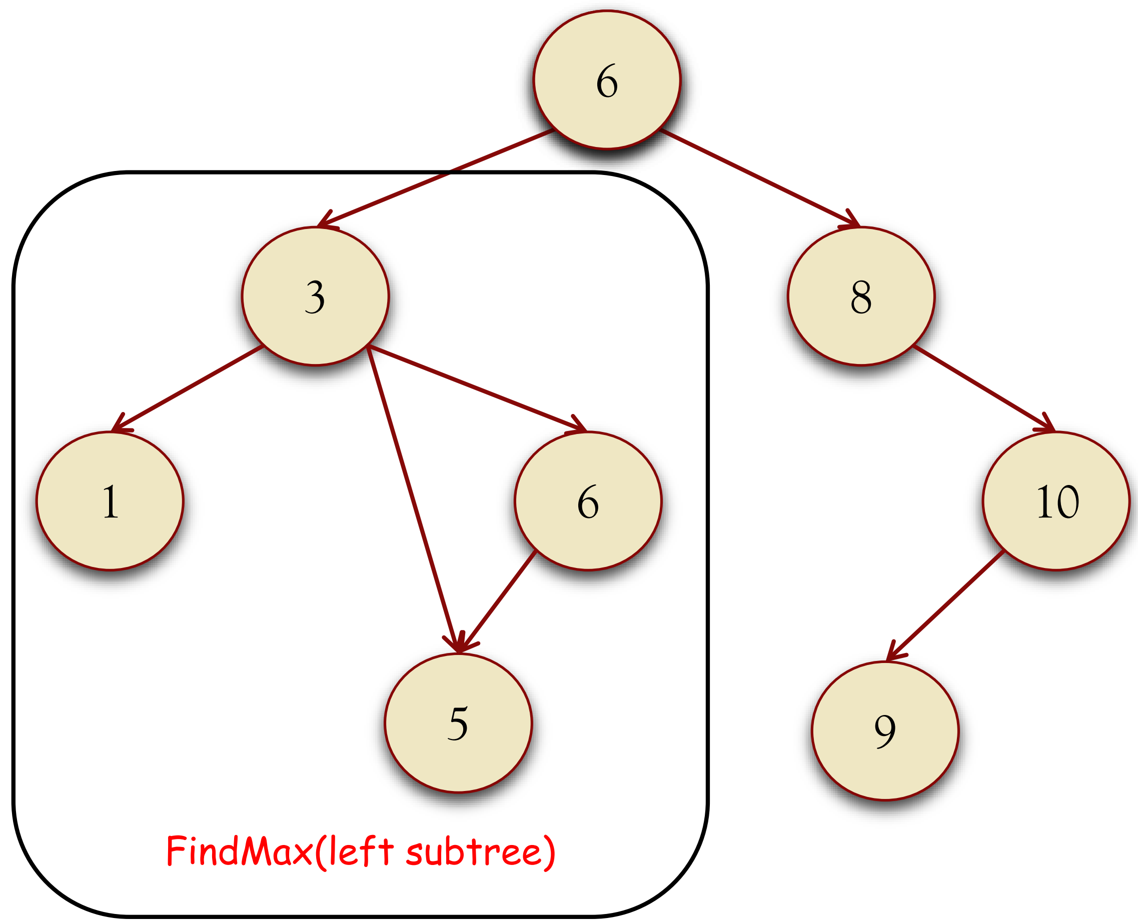
- One solution:
  - Try to eliminate the problem by randomly choosing between the smallest element in the right sub-tree and the largest in the left when replacing the deleted element (not rigorous and not prove it yet!!)

- Existing balanced BST solutions
  - Red-Black tree:height $O(\log n)$
  - AVL tree

Exercise 2: delete the node with two children

A bit complicated if we want to delete a NON-LEAF NODE with TWO children

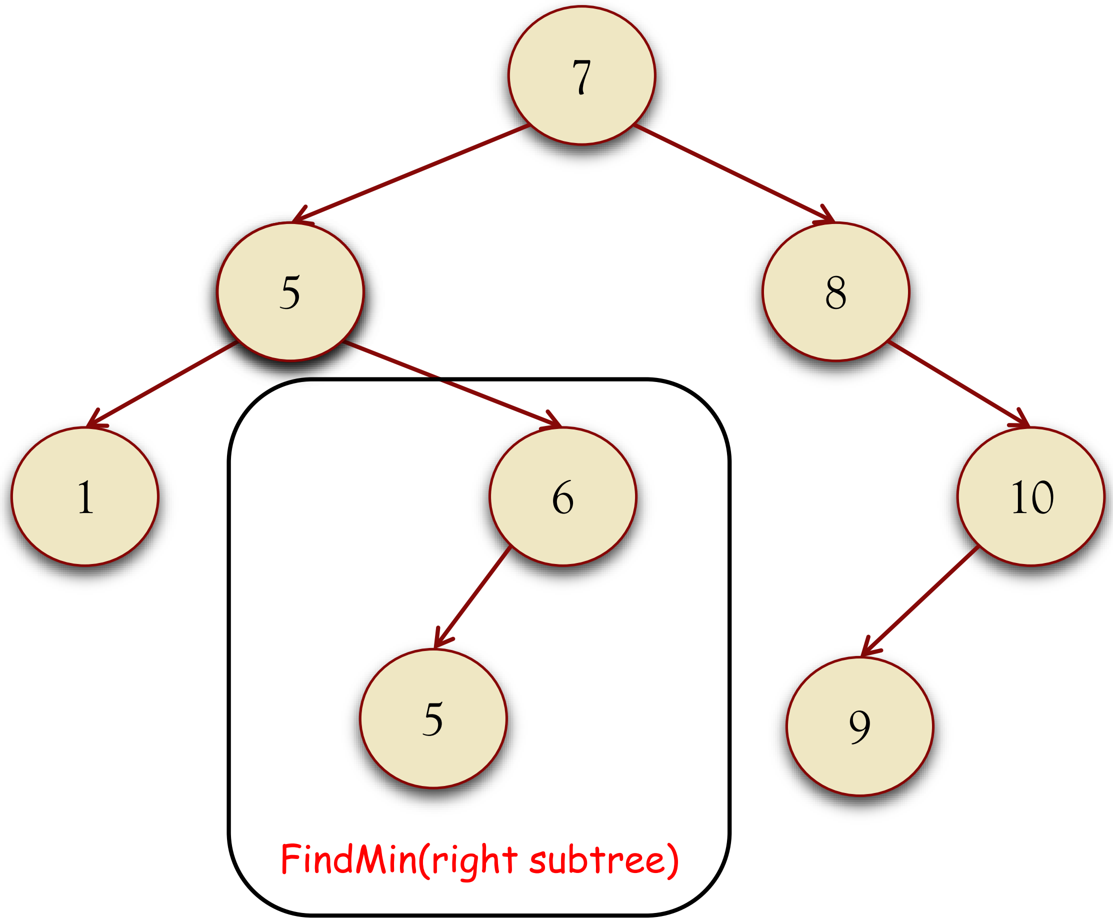
1.Locate the node
2.Find the rightmost node in its left subtree
3.Or find the leftmost node in its right subtree
4.Use the key of the node to replace its key
5.Delete the node



Exercise 3: delete the node with two children

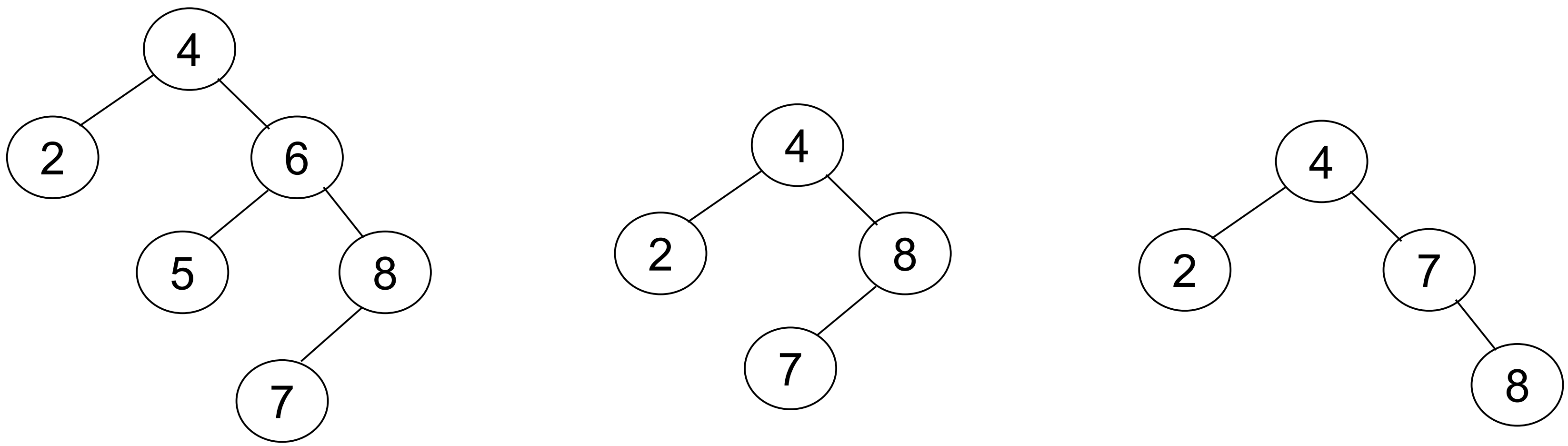
A bit complicated if we want to delete a NON-LEAF NODE with TWO children

1. Locate the node
2. Find the rightmost node in its left subtree
3. Or find the leftmost node in its right subtree
4. Use the key of the node to replace its key
5. Delete the node



Exercise 4

- In a binary search tree, are the insert and delete operations commutative?
  - insert(a) then insert(b) ⇔ insert(b) then insert(a)?
  - delete(a) then delete(b) ⇔ delete(b) then delete(a)?

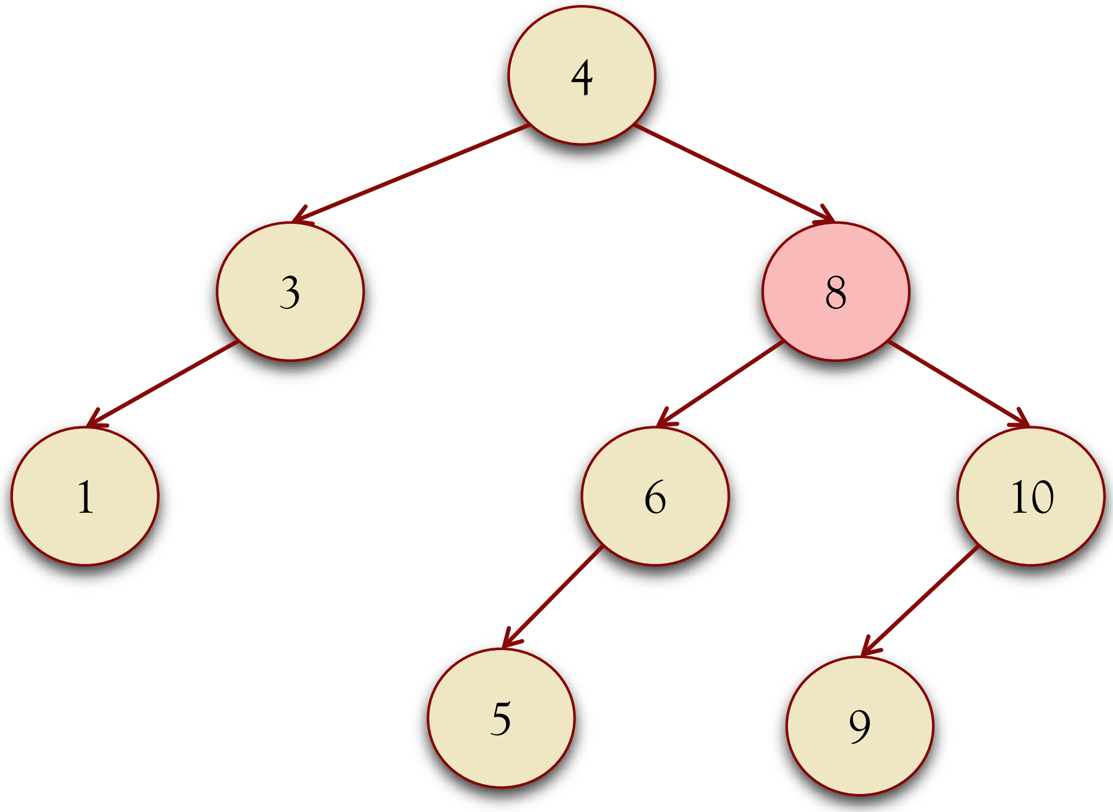


Delete 5 and then 6 or delete 6 and then 5

Exercise 5

A bit complicated if we want to delete a NON-LEAF NODE with TWO children

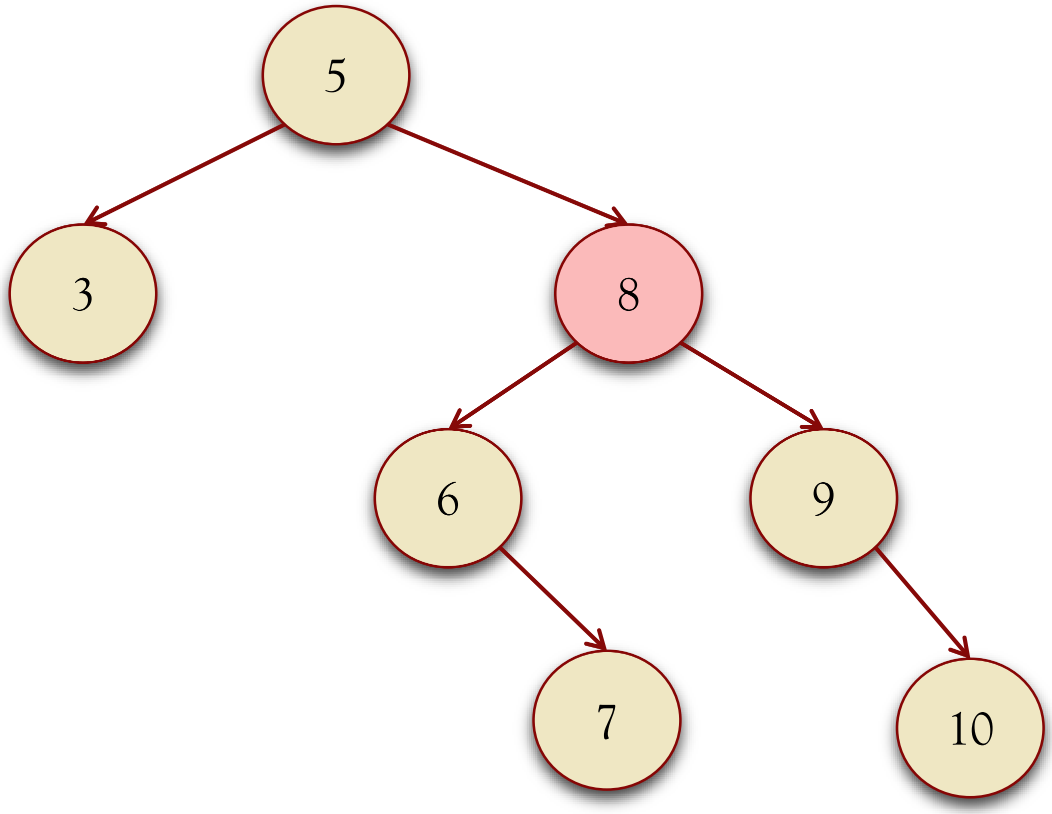
1. Locate the node
2. Find the leftmost node in its right subtree
3. Or find the rightmost node in its left subtree
4. Use the key of the node to replace its key
5. Delete the node



Exercise 6

A bit complicated if we want to delete a NON-LEAF NODE with TWO children

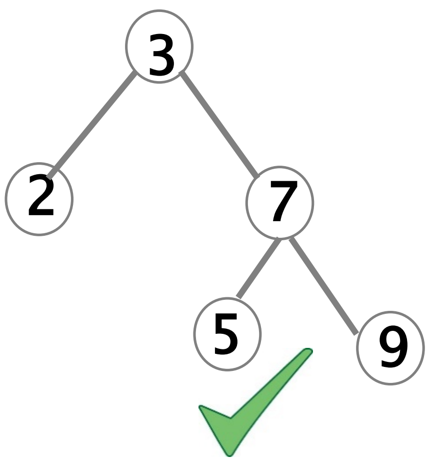
1. Locate the node
2. Find the leftmost node in its right subtree
3. Or find the rightmost node in its left subtree
4. Use the key of the node to replace its key
5. Delete the node

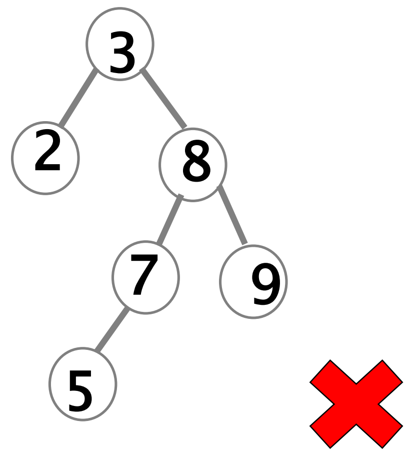


Balanced Binary Search Tree

- Goal: Keeping the height of the binary search tree low
  - Recap: A complete binary tree has height of $O(\log n)$, but the condition is too strict.
  - Solution: We relax the condition a little bit-> balanced binary search tree

- A binary search tree is balanced if:
  - For every node in the tree, the height of the left subtree differs from the height of the right subtree by at most 1



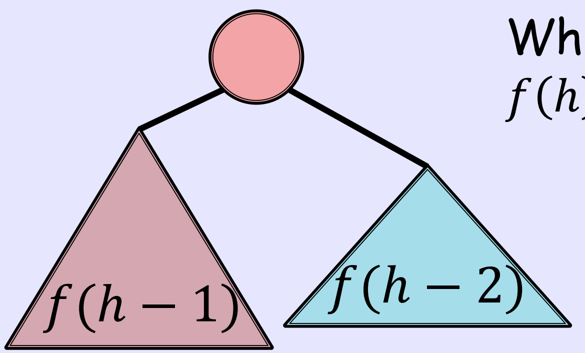


Height of Balanced Binary Search Tree

Theorem 1: Given a balanced binary search tree T of n nodes, the height, or equivalently the depth, of T is $O(\log n)$.

Proof: Let $f(h)$ be the minimum number of nodes of a balanced binary search tree of height h. Then, it is easy to verify that $f(1) = 2, \mathrm{f}(2) = 4$.

For any $h \geq 3$, we have that $f(h) = f(h-1) + f(h-2) + 1$



When h is even number:

$$\begin{aligned} f(h) &> f(h-1) + f(h-2) \\ &> 2f(h-2) \\ &> 4f(h-4) \\ &\dots \\ &> 2^{\frac{h}{2}-1} \cdot f(2) = 2^{\frac{h}{2}} \end{aligned}$$

When h is odd number:

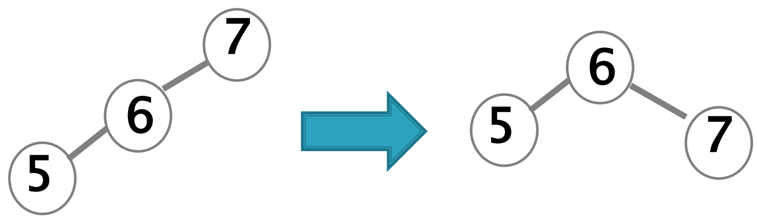
$$\begin{aligned} f(h) &> f(h-1) \\ &> 2^{\frac{h-1}{2}} \end{aligned}$$

Therefore, given a balanced BST of n nodes of height h, we have:

$$n > 2^{\frac{h-1}{2}} \Rightarrow h < 2\log_2 n + 1 \Rightarrow h = O(\log n)$$

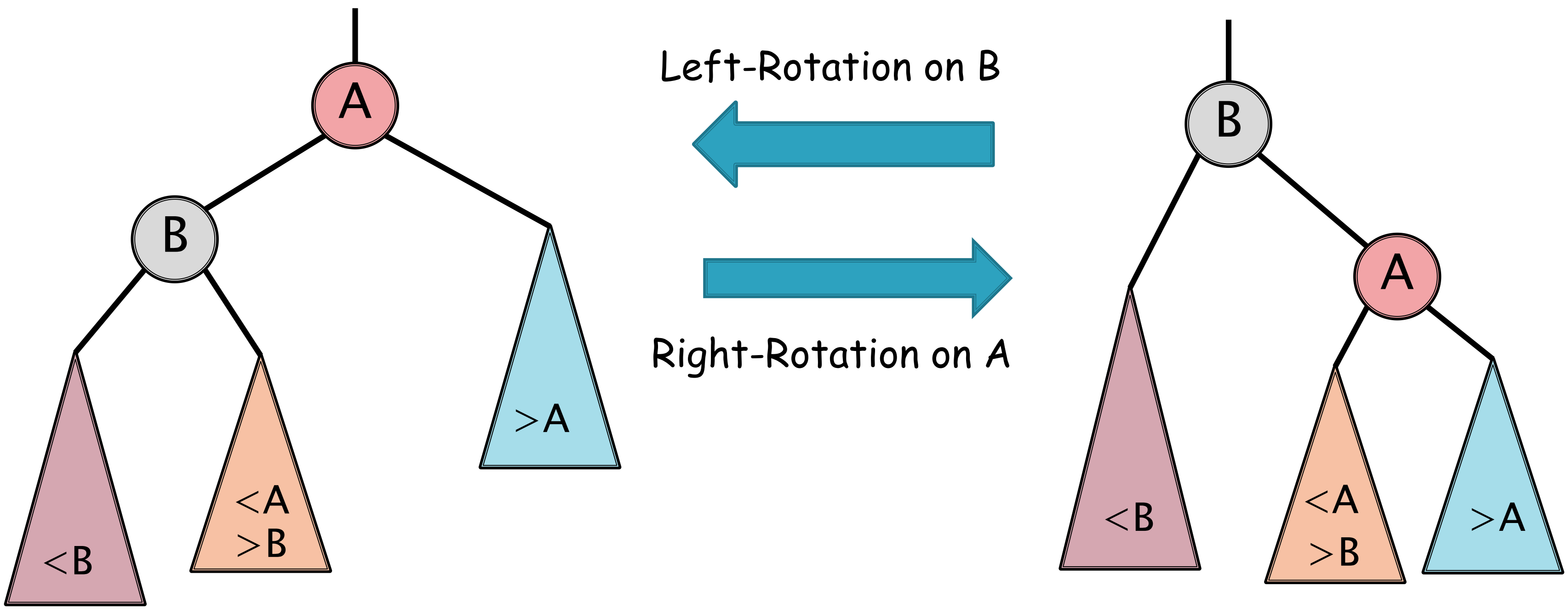
How to Keep a BST Balanced?

- Dynamic Rebalancing
  - After updating the BST, we rebalance the BST using rotation

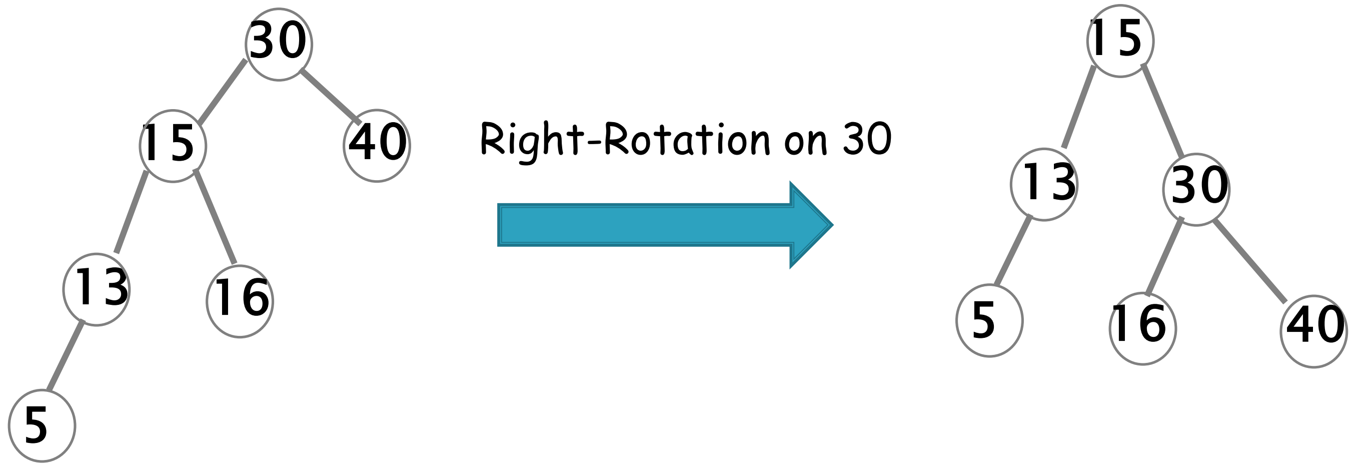


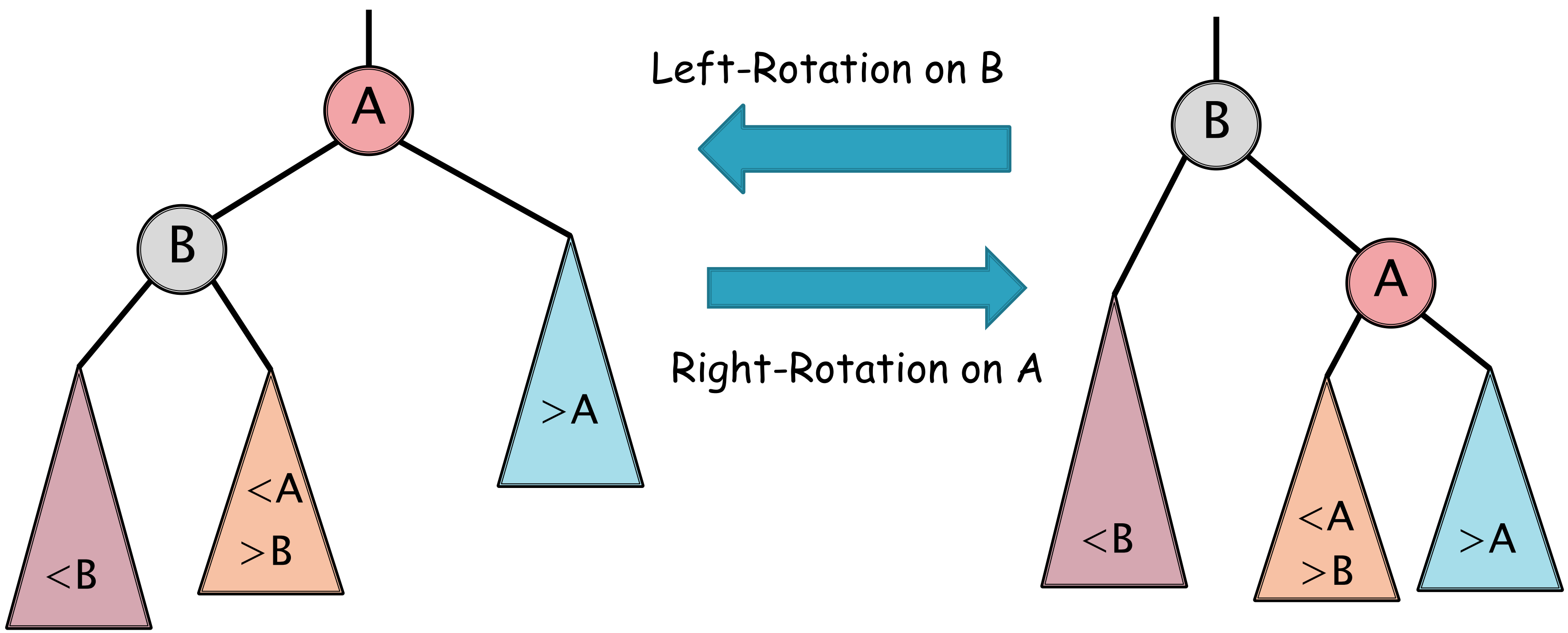
Rotations

- Rotation allows us to change the structure without violating the binary search tree property

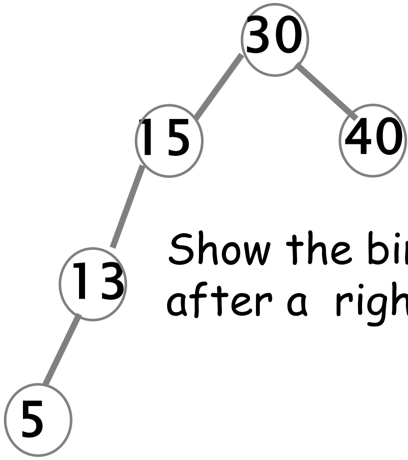


Rotation Examples

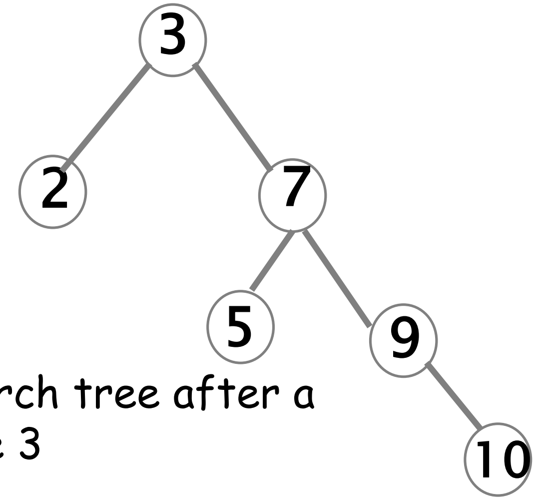




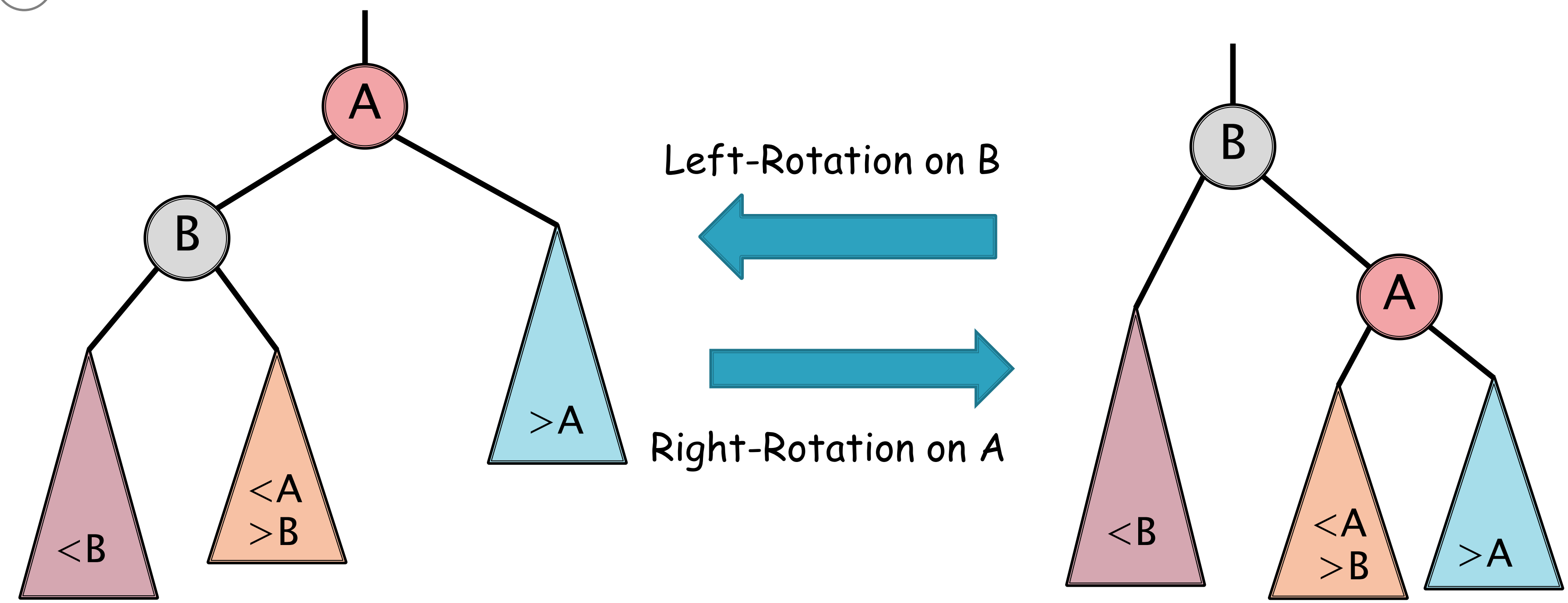
Practice



Show the binary search tree after a right rotation on node 30



Show the binary search tree after a left rotation on node 3



Priority queue, heap

Motivation

- Have you ever been jammed by a huge job while you are waiting for just one-page printout ?
- This is a typical situation for a simple first-in first-out (FIFO) queue
- Can there be a "smarter" printer?

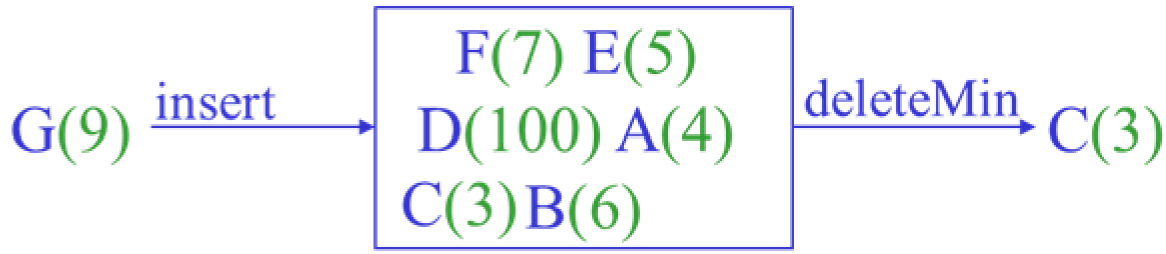
Motivation (Cont.)

- Other applications
  - Scheduling CPU jobs
  - Emergency room admission processing

- Problems using FIFO queue
  - Short jobs may go first
  - Most urgent cases should go first
  - Task with highest priority/lowest priority should go first

Priority Queue ADT

- Priority Queue operations
  - insert
  - deleteMin
  - create
  - isEmpty
  - …



- Priority Queue property:
  - for two elements in the queue, ***x*** and ***y***, if ***x*** has a ***lower priority*** value than ***y***, ***x*** will be deleted before ***y***

Simple Implementations

- Multiple possibilities for the implementation
  - Simple linked list (Suggestion 1)
    - Insert at the front in $O(1)$
    - Delete minimum in $O(N)$
  - Sorted linked list (Suggestion 2)
    - Insert in $O(N)$
    - Delete minimum in $O(1)$
  - Sorted Array (Suggestion 3)
    - Insert in $O(N)$
    - Delete minimum in $O(N)$

Simple Implementations

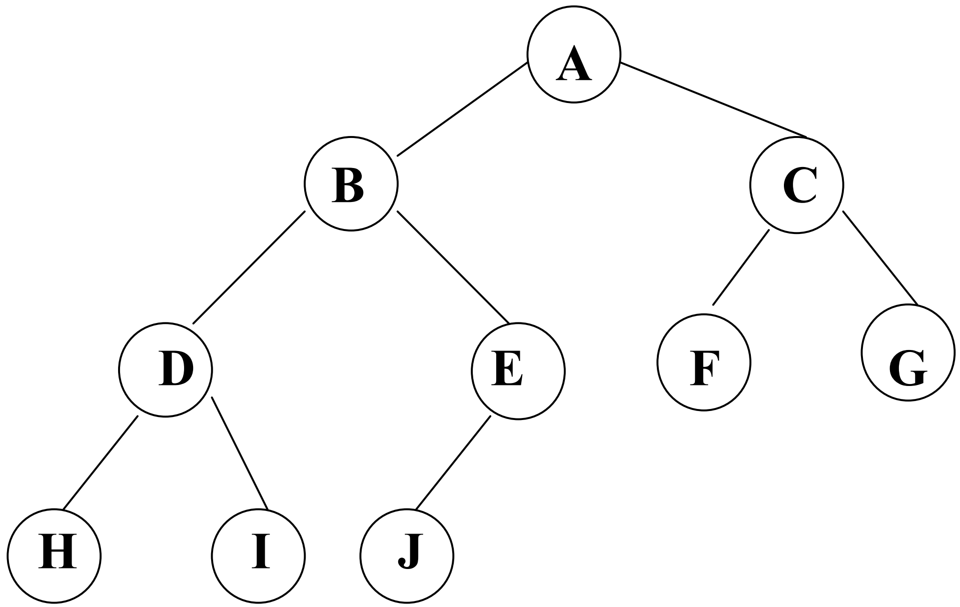
- Binary heap (Best solution)
  - O (log N) for insert and deleteMin for worst cases
  - Two properties:
    - Structure property
    - Heap order property

- Other solution: binary search tree (to be explained later)
  - O (log N) on average for both operations

Binary Heap (Heap)

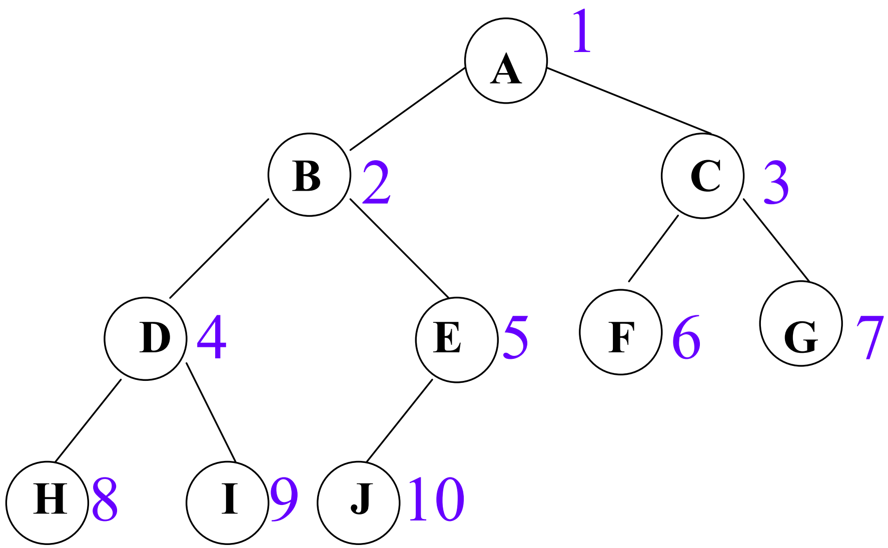
- (A) Structure Property
  - A heap is a binary tree that is completely filled, except at the bottom level, which is filled from left to right
  - A complete binary tree of height h has between 2^h and $2^{h+1} - 1$ nodes
  - The height of a complete binary tree = $\lfloor \log N \rfloor$
    - round down, e.g., $\lfloor 2.7 \rfloor = 2$

Binary Heap (Heap)



- A complete binary tree can be represented in an array without using pointers

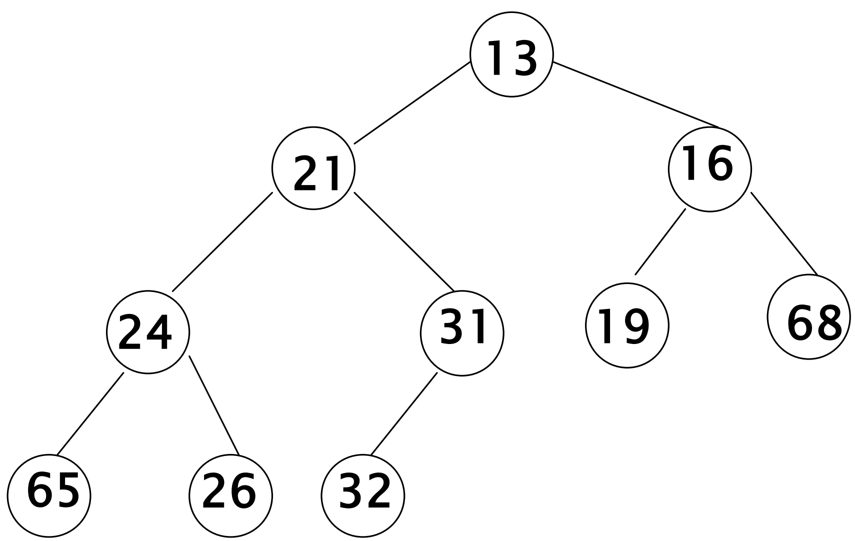
	A	B	C	D	E	F	G	H	I	J					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

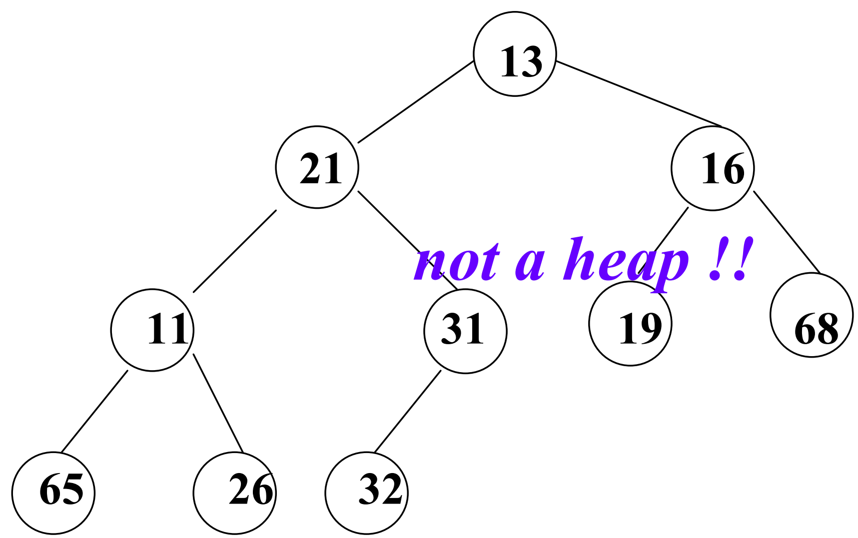


- The root is at position 1 (reserve position 0 for the *implementation purpose*)
- For an element at position *i*,
  - its left child is at position 2*i*
  - its right child at 2*i* +1; its parent is at floor $\lfloor i/2 \rfloor$

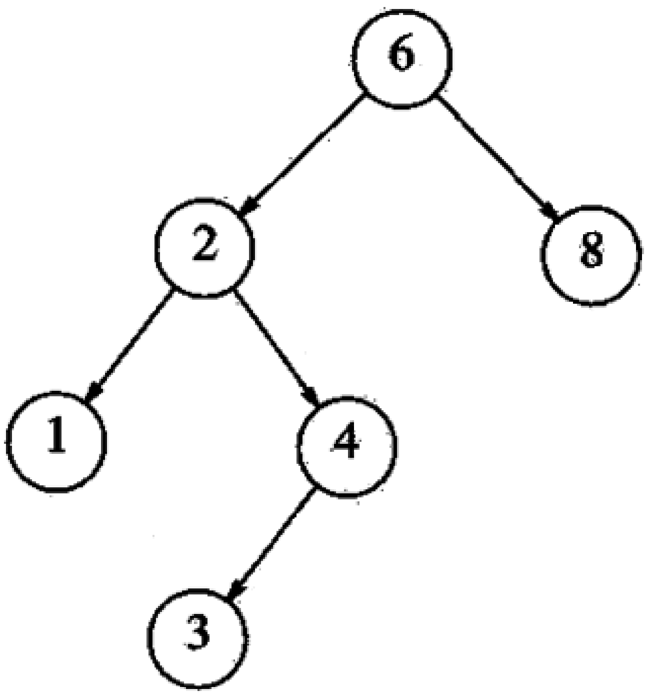
Binary Heap

- (B) Heap Order Property
  - The value at any node should be smaller than (or equal to) all of its descendants (guarantee that the node with the minimum value is at the root)

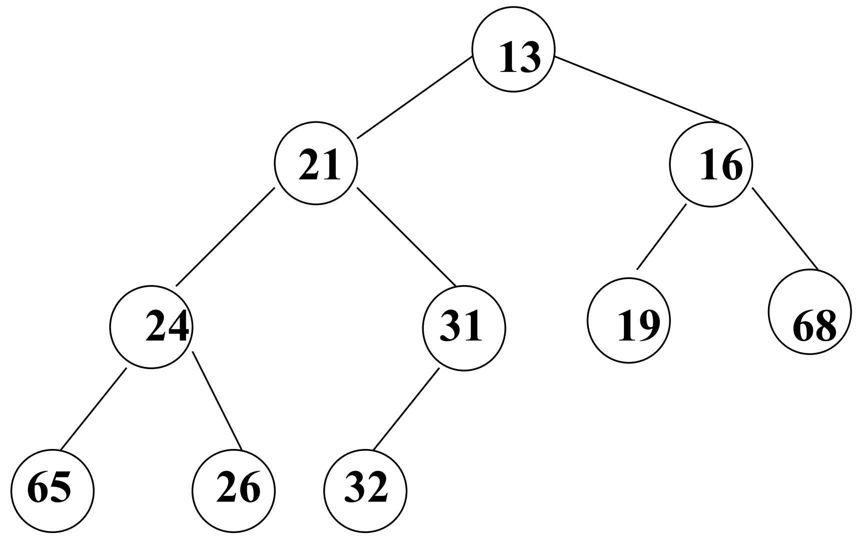




BST vs heap



A Binary Search Tree



A Heap

Note the difference in node ordering!!

Binary Heap

- Class skeleton for **Elements**

```
class ElementType {
    int priority;
    String data;
    public ElementType(int priority, String data) {
        this.priority = priority;
        this.data = data;
    }
    public boolean isHigherPriorityThan(ElementType e) {
        return priority < e.priority;
    }
}
```

Binary Heap

- Definition and constructor of Priority Queue

```
public class BinaryHeap {
        private int currentSize;      // Number of elements in heap
        private ElementType[] arr; // The heap array

        public BinaryHeap (int capacity) {
                currentSize = 0;
                arr = new ElementType[capacity + 1];
        }
}
```

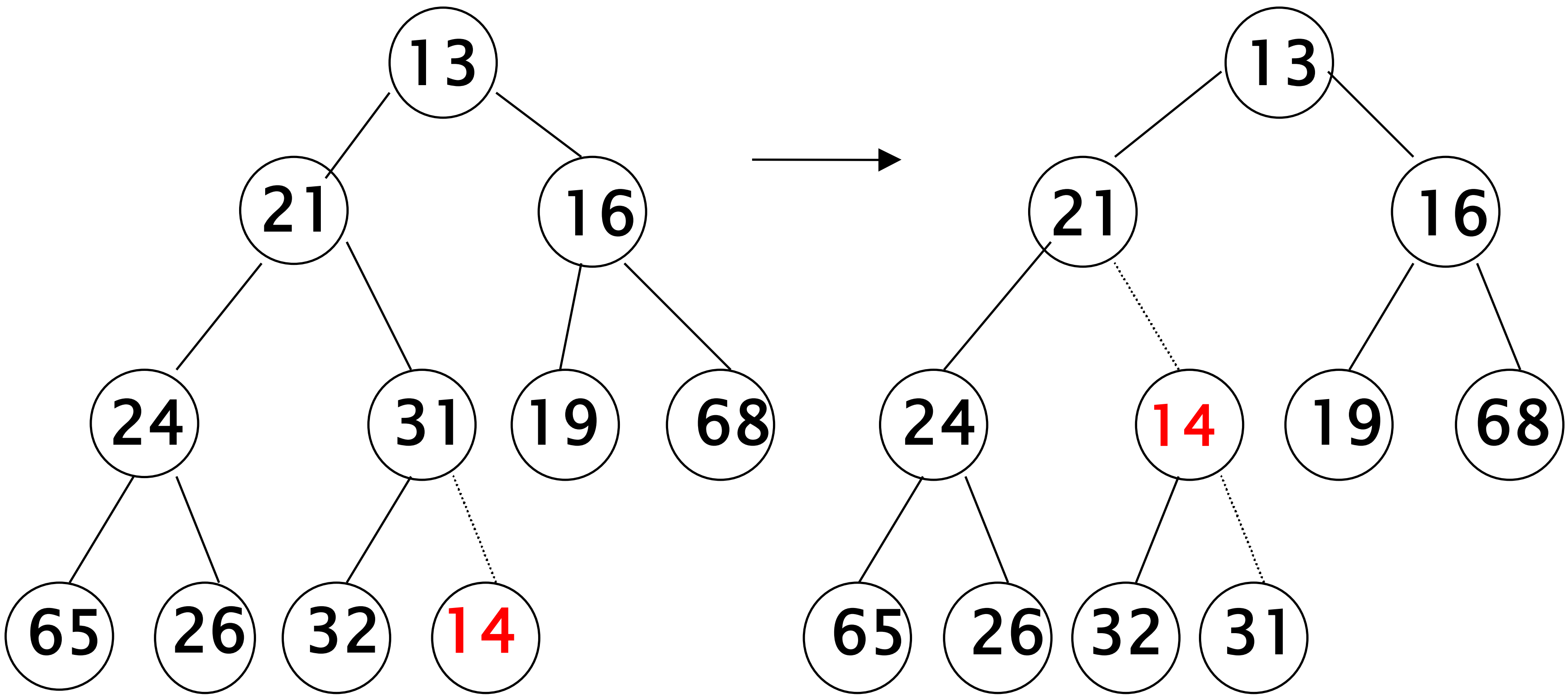
Binary Heap

- Some functions
  - void destroy ( );
  - void makeEmpty ( );
  - void insert (ElementType X);
  - ElementType deleteMin ( );
  - ElementType findMin ( );
  - boolean isEmpty ( );
  - boolean isFull ( );

Binary Heap - Insert

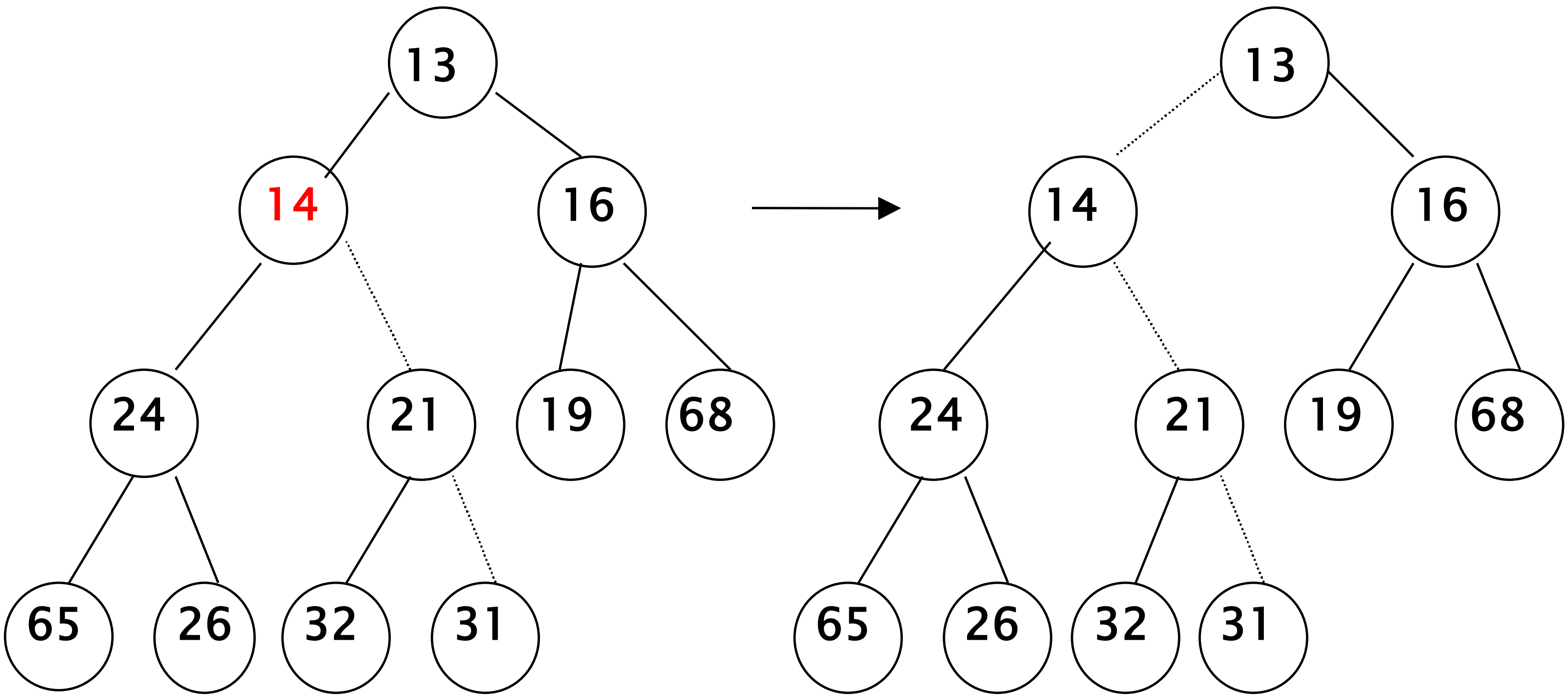
Attempt to insert 14:

(1) creating the hole, and (2) bubbling the hole up



Binary Heap - Insert

The remaining two steps to insert 14 in previous heap



Binary Heap - Insert

- To insert an element X,
  - Create a hole in the next available location
  - If X can be placed in the hole without violating heap order, insertion is complete
  - Otherwise slide the element that is in the hole's parent node into the hole, i.e., bubbling the hole up towards the root
  - Continue this process until X can be placed in the hole (a *percolating up* process)

Binary Heap - Insert

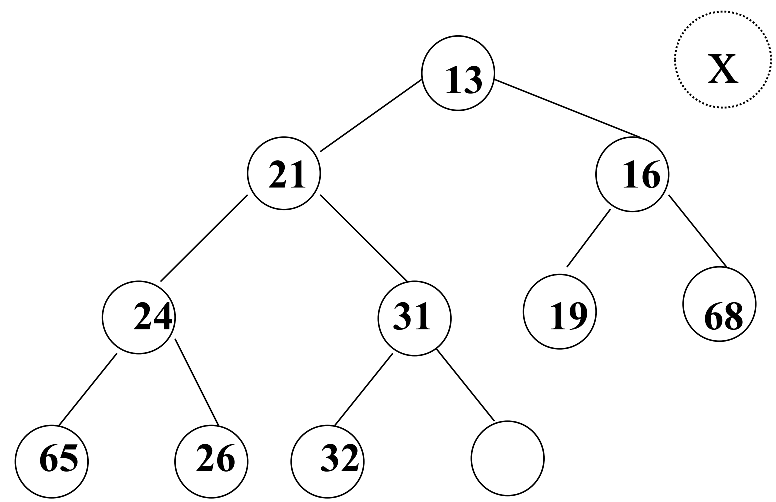
Attention!

Worst case running time is $O(\log N)$ - the new element is percolating up all the way to the root

Question: when does this happen?

Binary Heap - Insert

```
public void insert(ElementType x) throws Exception {
        if (isFull())
                throw new Exception("Overflow");

        // Percolate up
        int hole = ++currentSize;
        while(hole > 1 && x.isHigherPriorityThan(arr[hole/2])) {
                arr[hole] = array[hole / 2];
                hole /= 2;
        }
        arr[hole] = x;
}
```



Binary Heap - Insert (another implementation)

```
public void insert(ElementType x) throws Exception {
        if (isFull())
                throw new Exception("Overflow");

        // Percolate up
        int hole = ++currentSize;
        for(arr[0] = x; x.isHigherPriorityThan(arr[hole/2]));hole /= 2)
                arr[hole] = array[hole / 2];
        arr[hole] = x;
}
```