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

CSC3100 Data Structures
Lecture 14: Binary search tree

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

Outline

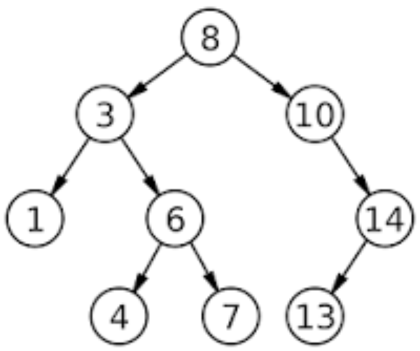
- In this lecture, we will learn
  - Binary search tree (BST)

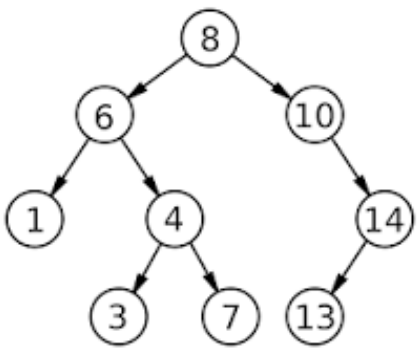
  - Operations on BST
    - Finding successor and predecessor
    - Insert & delete

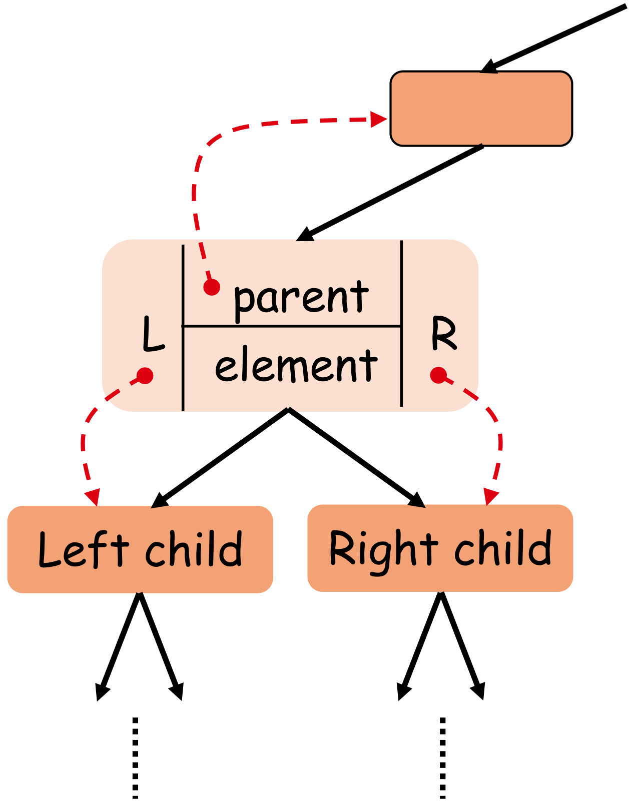
  - Comparison between BST and linked list

Binary Search Tree (BST) property

- BST is a tree such that for each node T,
  - the key values in its left subtree are *smaller* than the key value of T
  - the key values in its right subtree are *larger than* the key value of T







Applications of BST

- Many applications due to its ordered structure
  - Useful for indexing and multi-level indexing
  - Helpful in maintaining a sorted stream of data
  - Helpful to implement various searching algorithms and data structures (e.g., TreeMap, TreeSet, Priority queue)

java.util

Class TreeMap<K,V>

java.lang.Object
    java.util.AbstractMap<K,V>
        java.util.TreeMap<K,V>

java.util

Class TreeSet<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractSet<E>
            java.util.TreeSet<E>

BST ADT

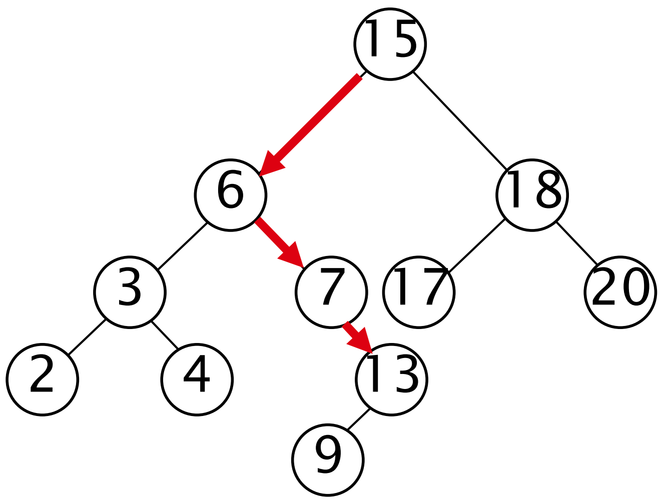
- Support many dynamic set operations
  - searchKey, findMin, findMax, predecessor, successor, insert, delete

- Running time of basic operations on BST
  - On average: $\Theta(\log n)$
    - The expected height of the tree is $\log n$
  - In the worst case: $\Theta(n)$
    - The tree is a linear chain of n nodes

Searching for a Key

- Given a pointer to the root of a tree and a key k:
  - Return a pointer to a node with key k if one exists, otherwise return NIL

- Example

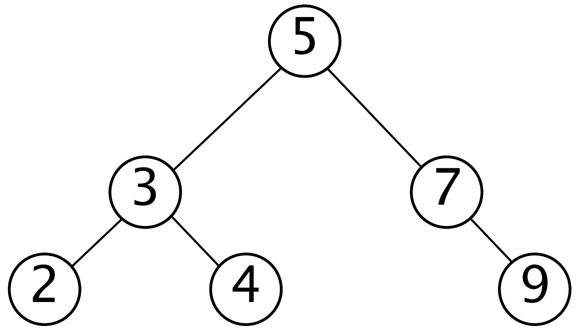


- Search for key 13:
  - $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$

Searching for a Key

find(x, k)

```
if x = NIL or k = key [x]
    then return x
if k < key [x]
    then return find(left [x], k )
    else return find(right [x], k )
```



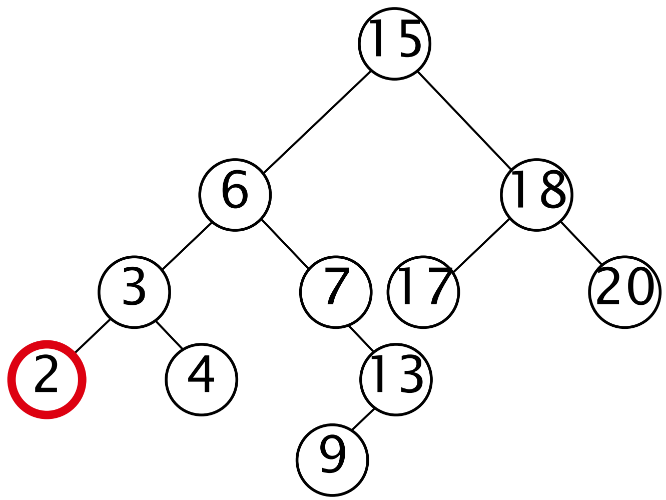
Running Time: O (h),
h is the height of the tree

Finding the Minimum

- Goal: find the minimum value in a BST
  - Following left child pointers from the root, until a NIL is encountered

findMin(x)

1. **while** left [x] $\neq$ NIL
2. **do** x $\leftarrow$ left [x]
3. **return** x



Minimum = 2

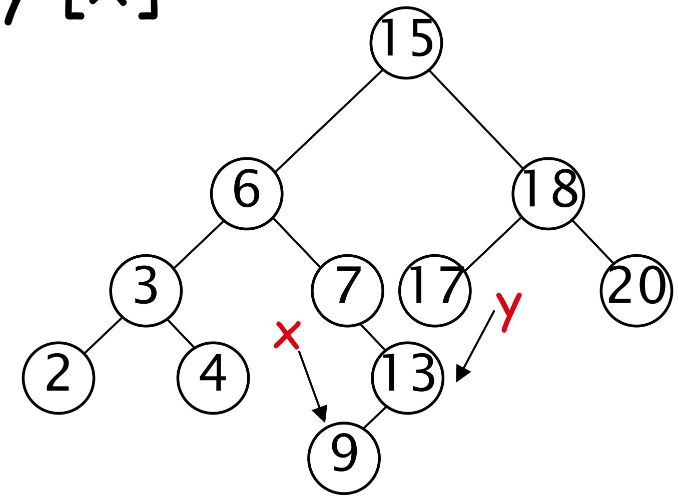
Running time: O(h)

h is the height of tree

Successor

Def: successor (x) = y, such that key [y] is the smallest key > key [x]

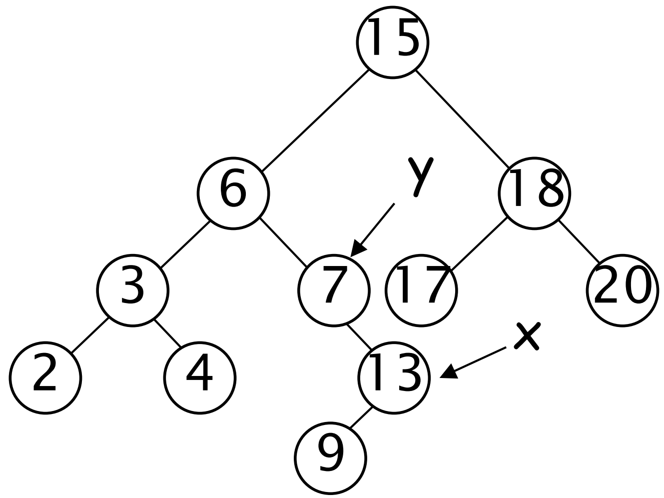
- E.g.: successor (15) = 17
  successor (13) = 15
  successor (9) = 13



- Case 1: right (x) is non empty
  - successor (x) = the minimum in right (x)
- Case 2: right (x) is empty
  - go up the tree until the current node is a left child: successor (x) is the parent of the current node
  - if you cannot go further (and you reached the root): x is the largest element

Successor

```
successor(x)
    if right [x] ≠ NIL
      then return findMin(right [x])
    y ← p[x]
    while y ≠ NIL and x = right [y]
      do x ← y
         y ← p[y]
    return y
```



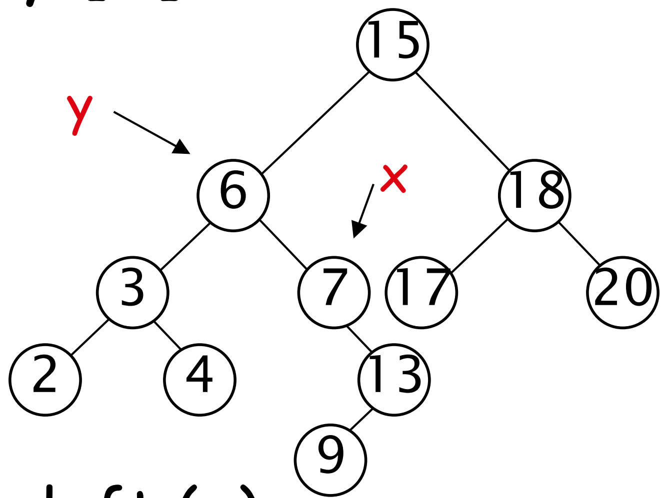
Running time: O (h)

h is the height of the tree

Predecessor

Def: predecessor (x) = y, such that key [y] is the biggest key < key [x]

- E.g.: predecessor (15) = 13
  predecessor (9) = 7
  predecessor (7) = 6



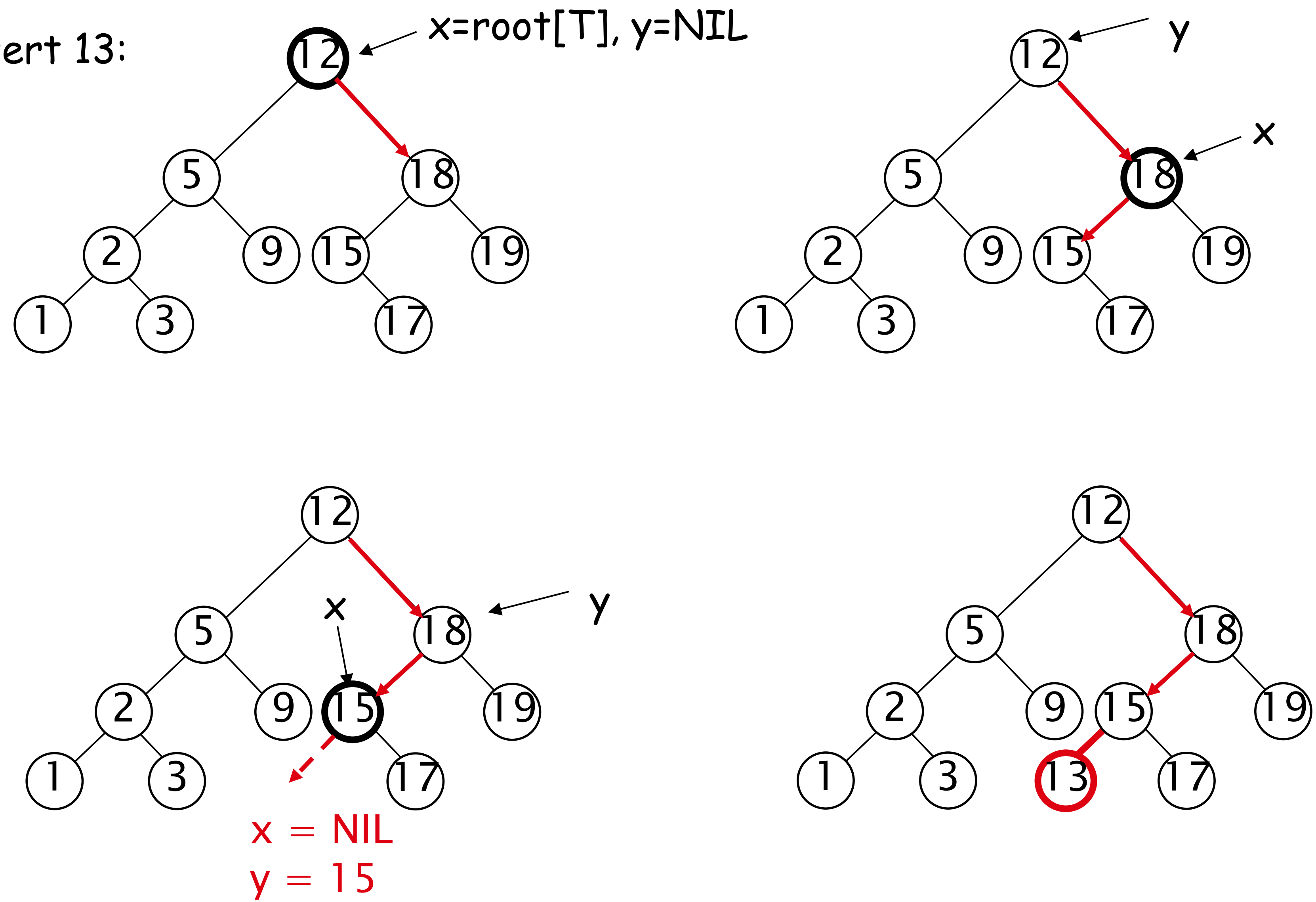
- Case 1: left (x) is non empty
  - predecessor (x) = the maximum in left (x)
- Case 2: left (x) is empty
  - go up the tree until the current node is a right child: predecessor (x) is the parent of the current node
  - if you cannot go further (and you reached the root): x is the smallest element

Insertion

- 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")

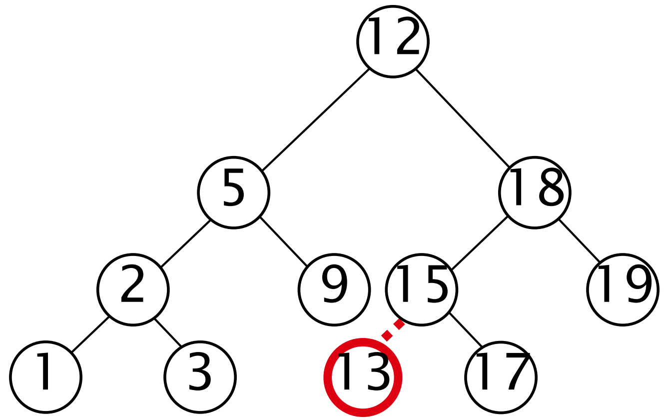
Example

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                    ▷Tree T was empty
   else if key [z] < key [y]
            then left [y] ← z
            else right [y] ← z
```



Best-case and worst-case time complexities?

Running time: O(h)

Exercise 1

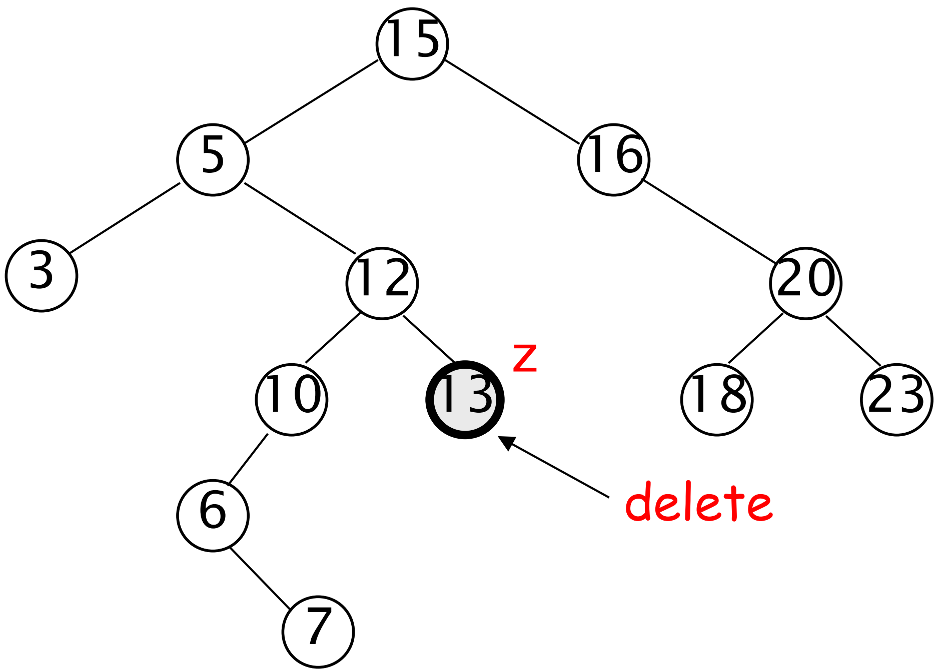
- Build a binary search tree for the following sequence

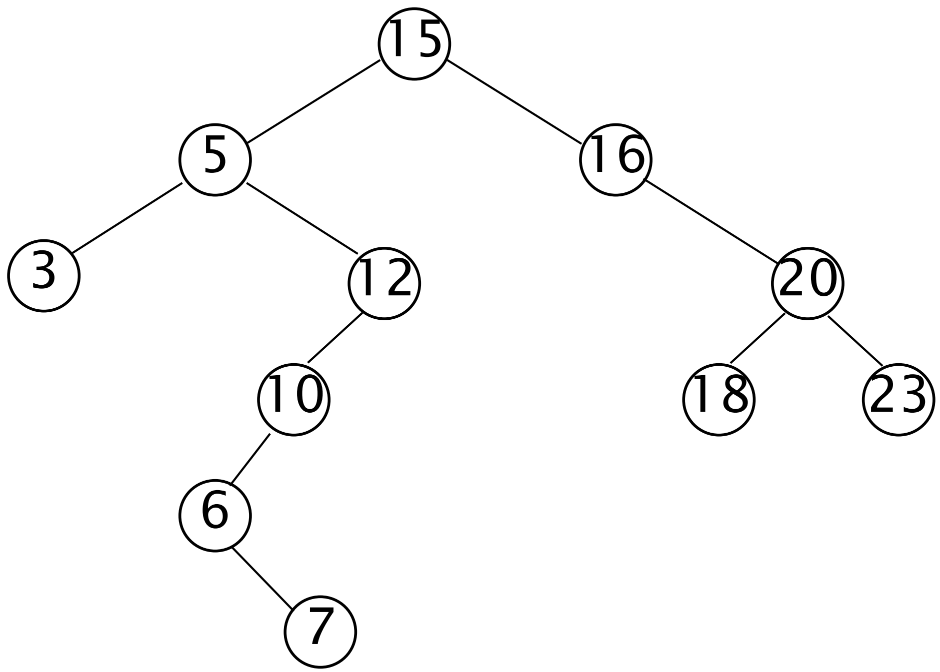
    15, 6, 18, 3, 7, 17, 20, 2, 4

Deletion

- Goal: Delete a given node z from a binary search tree

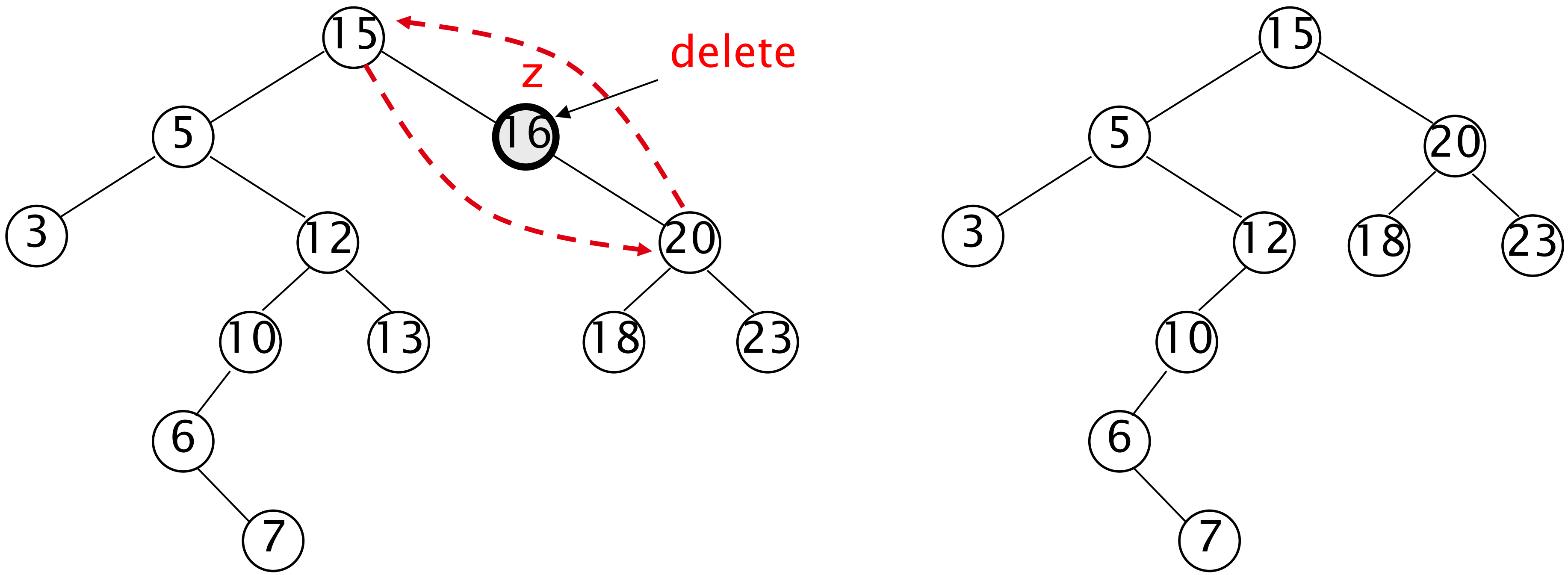
- Idea:
  - **Case 1:** z has no children
    - Delete z by making the parent of z point to NIL





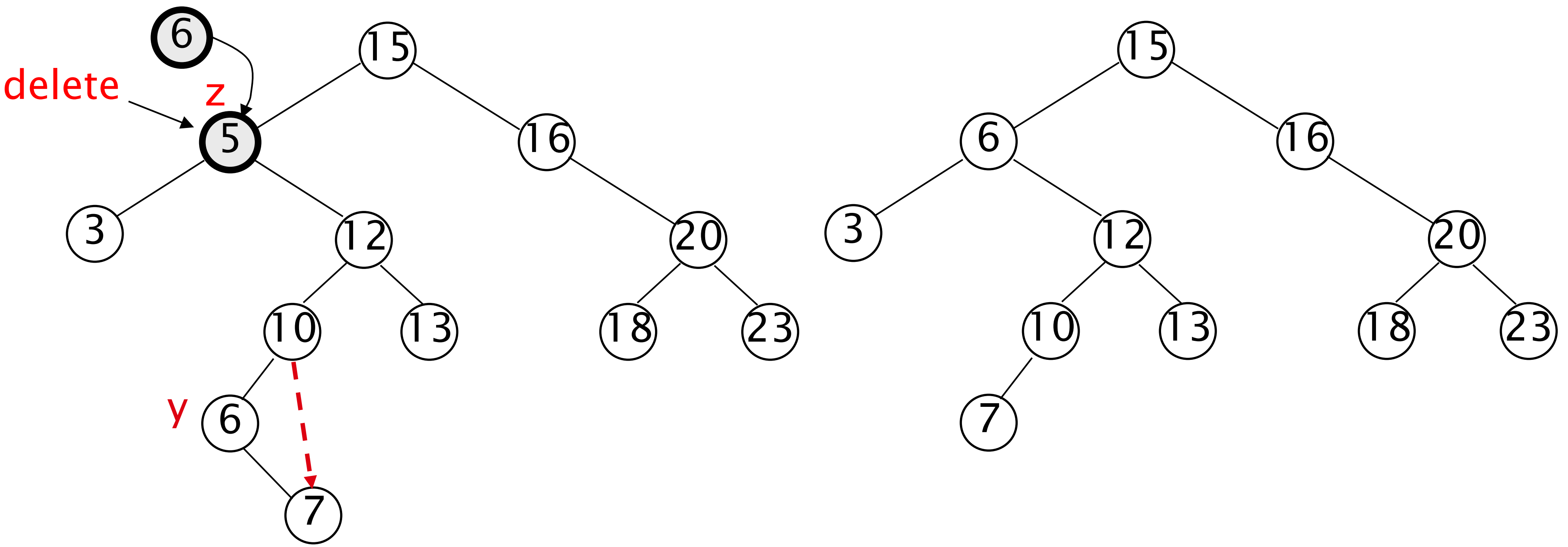
Deletion

- **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



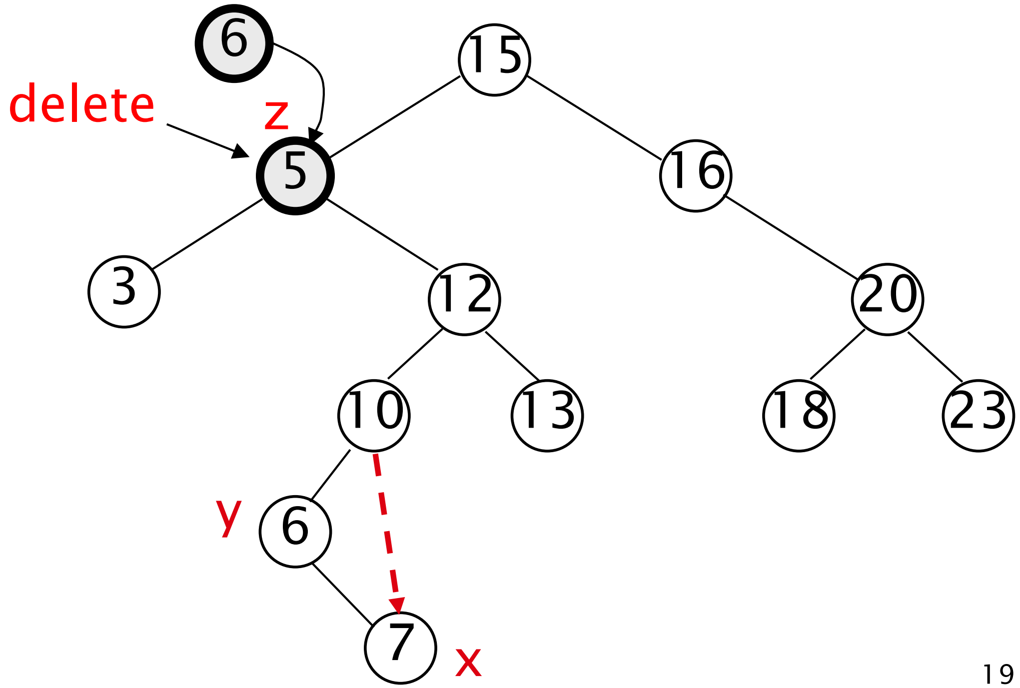
Deletion

- **Case 3:** z has two children
  - Find z's *successor y* (the leftmost node in z's right subtree)
  - y has either no child 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



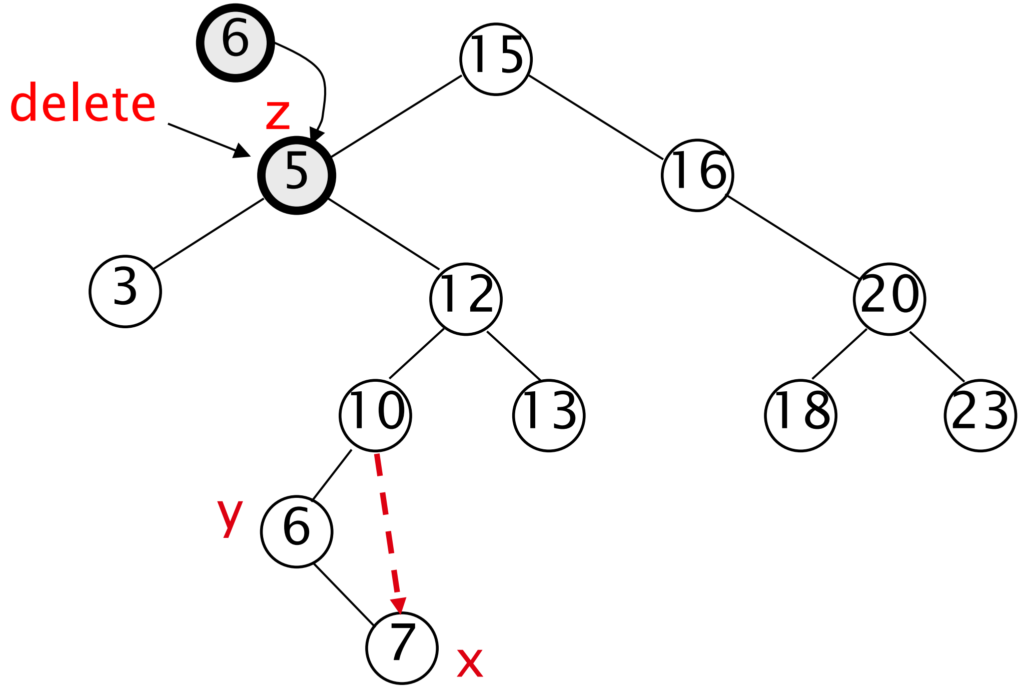
Deletion algorithm

```
if left[z] = NIL or right[z] = NIL
   then y ← z                           //z has one child
   else y ← TREE-SUCCESSOR(z)           //z has 2 children
if left[y] ≠ NIL
   then x ← left[y]
   else x ← right[y]
if x ≠ NIL
   then p[x] ← p[y]
```



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
```



Best-case and worst-case time complexities?

Running time: O(h)

Summary

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

Binary search trees vs linear lists

Operation	BST	Array-based List	Linked List
Constructor	O(1)	O(1)	O(1)
IsFull	O(1)	O(1)	O(1)
IsEmpty	O(1)	O(1)	O(1)
RetrieveItem	O(logN)*	O(logN)	O(N)
InsertItem	O(logN)*	O(N)	O(N)
DeleteItem	O(logN)*	O(N)	O(N)

*assuming h=O(logN)

The issues in BST

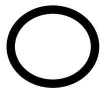
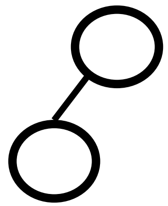
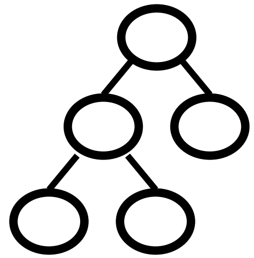
- After a series of DELETION, the above algorithm favors making the left sub-trees deeper than the right

- 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
  - AVL tree
  - Red-Black tree: height $O(\log n)$

Exercise 1: 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

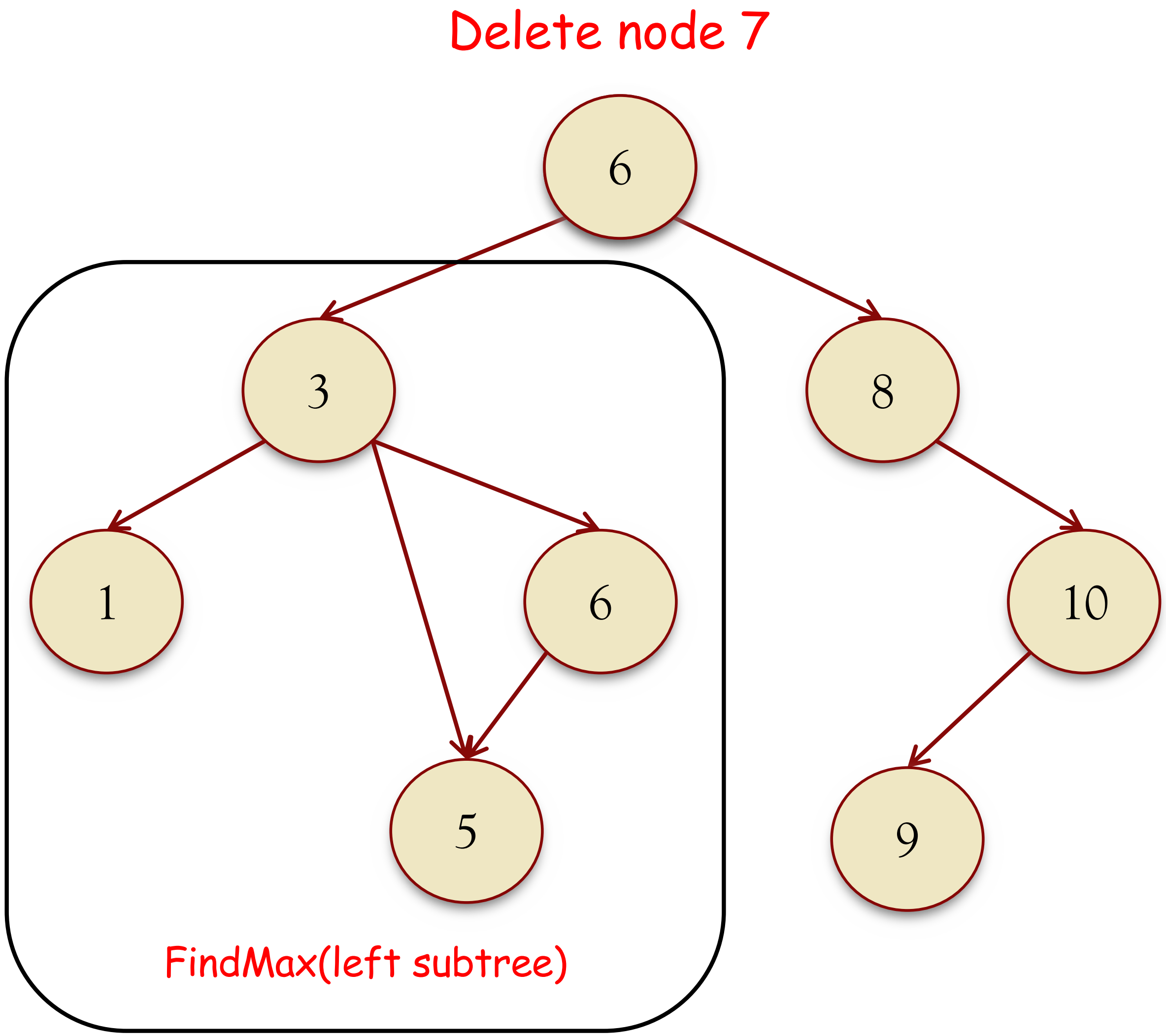
```
//To count the number of leaf nodes

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);
}
```

Exercise 2: delete the node with two children

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

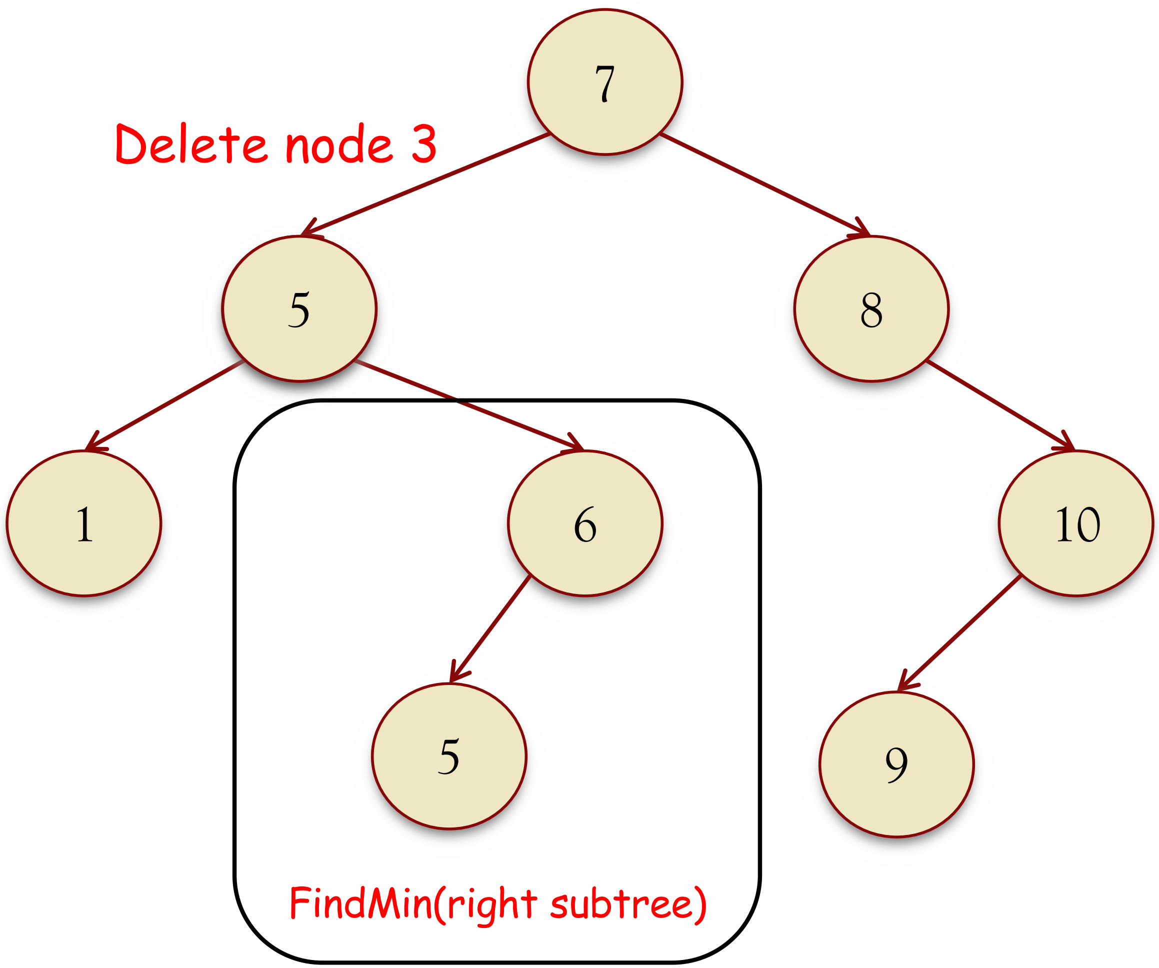
1. Locate the node
2. Find the rightmost node in its left subtree
3. Or find the leftmost node in its right subtree
4. Use the key of the node to replace its key
5. Delete the node



Exercise 3: delete the node with two children

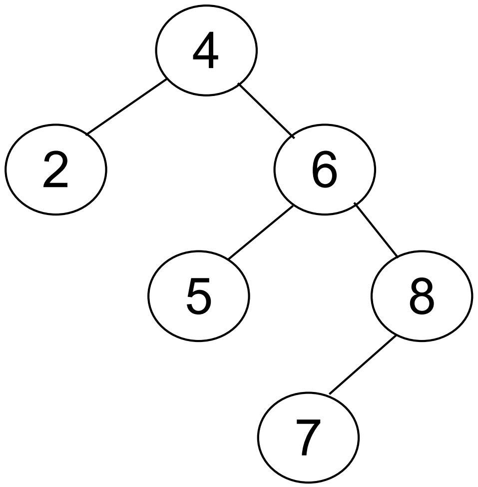
A bit complicated if we want to delete a NON-LEAF NODE with TWO children

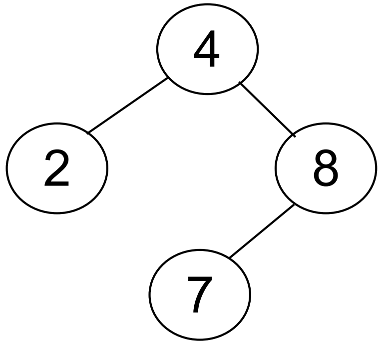
1. Locate the node
2. Find the rightmost node in its left subtree
3. Or find the leftmost node in its right subtree
4. Use the key of the node to replace its key
5. Delete the node

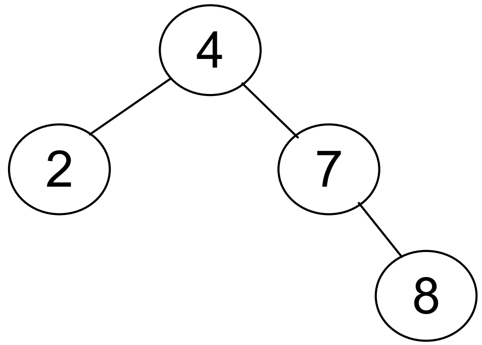


Exercise 4

- 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)?





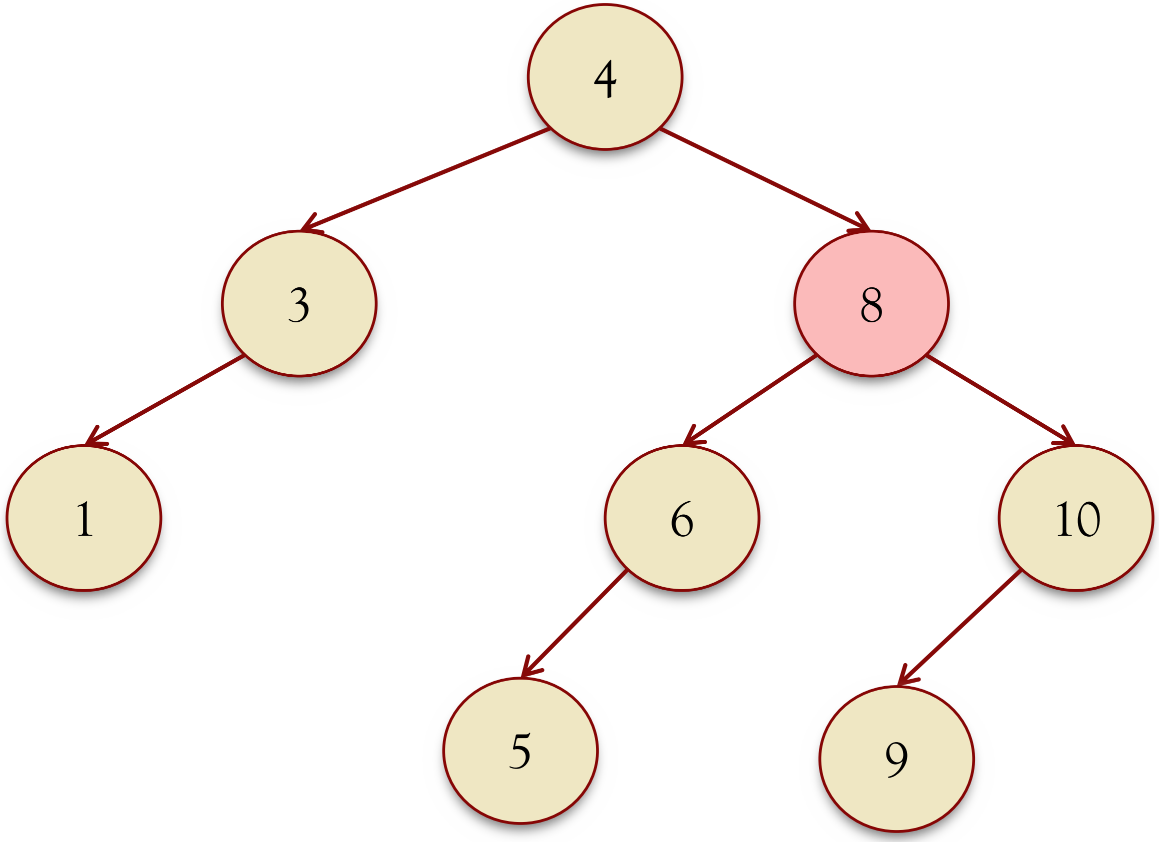


Delete 5 and then 6 or delete 6 and then 5

Exercise 5

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

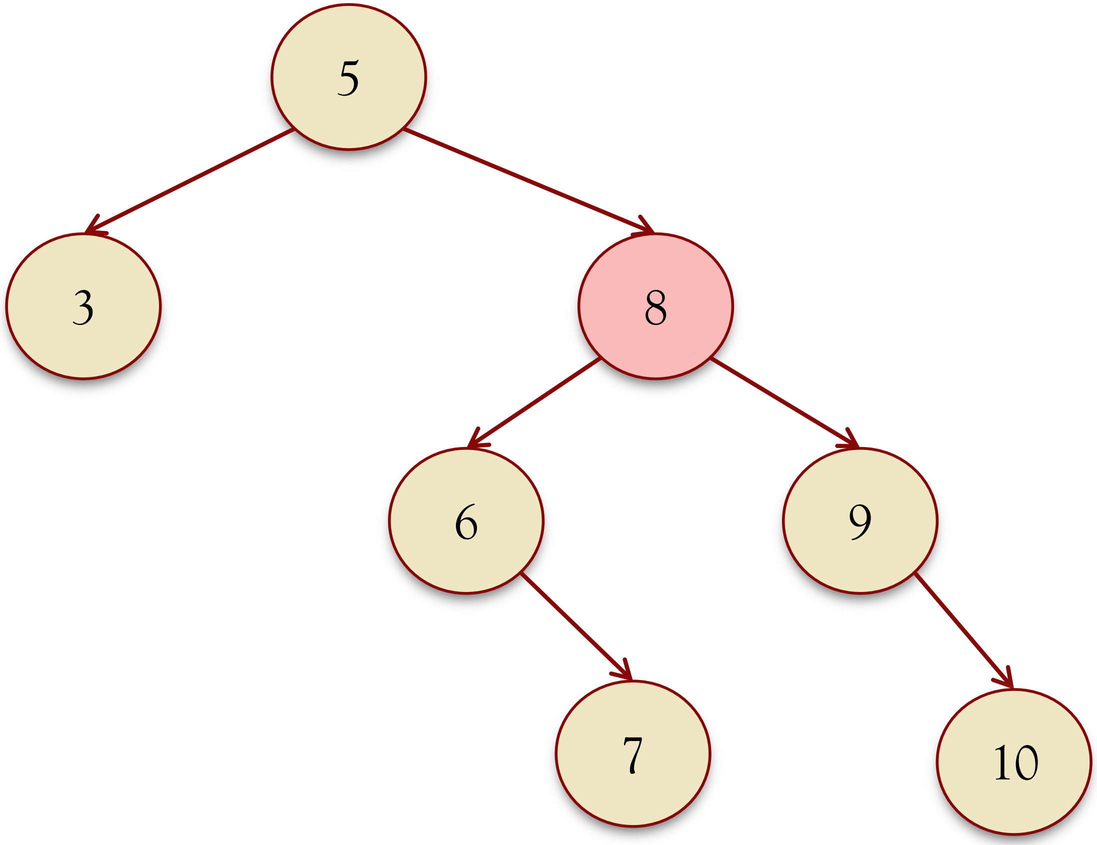
1. Locate the node
2. Find the leftmost node in its right subtree
3. Or find the rightmost node in its left subtree
4. Use the key of the node to replace its key
5. Delete the node



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



Recommended reading

- Reading this week
  - Chapter 12, textbook

- Next lecture
  - Heap: chapter 12, textbook