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

CSC3100 Data Structures
Lecture 14: Binary search tree

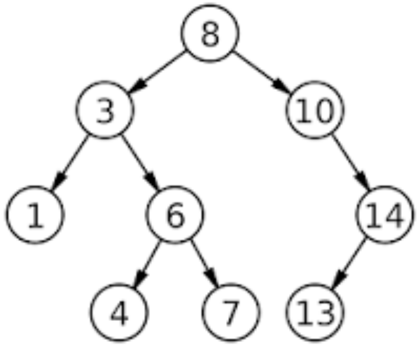
Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

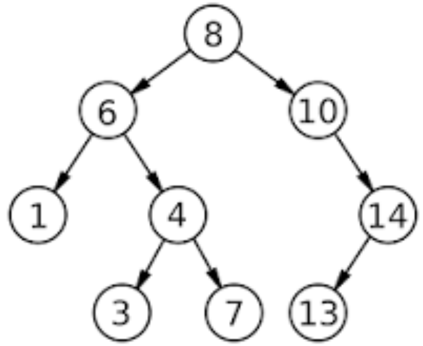
Outline

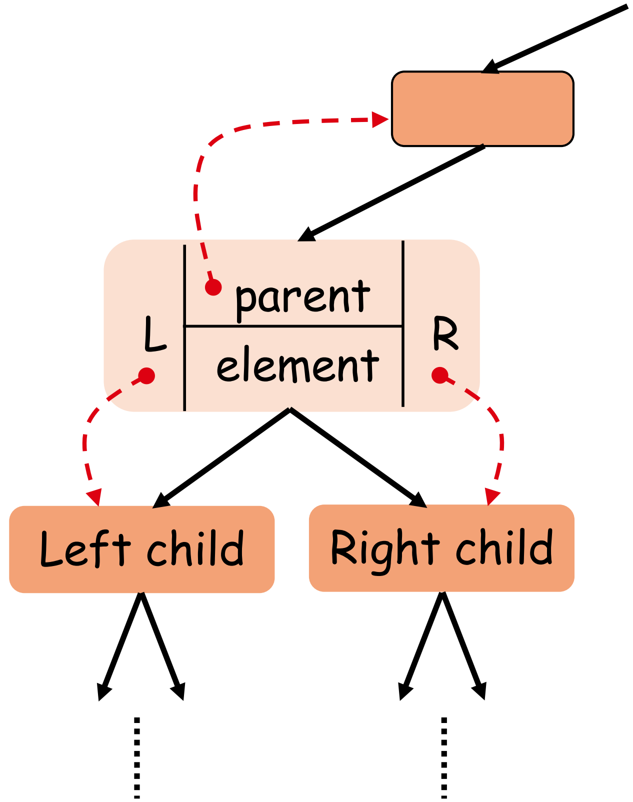
- In this lecture, we will learn
  - Binary search tree (BST)
  - Operations on BST
    - Search a key, find the minimum/maximum, find successor/predecessor
    - Insert, delete
  - Exercises

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>

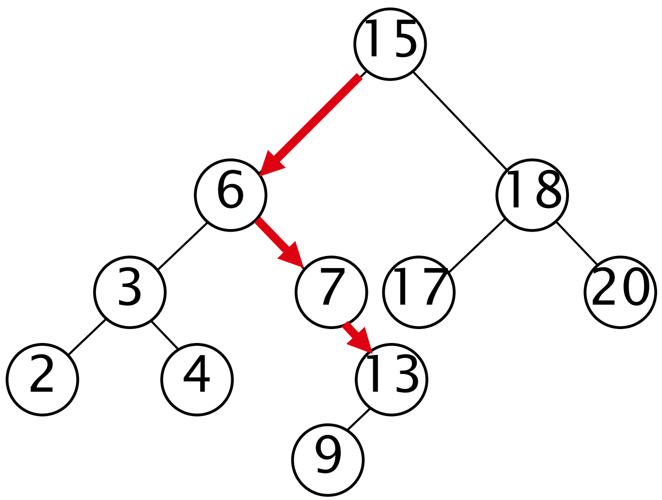
BST ADT

- 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 logn
  - 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

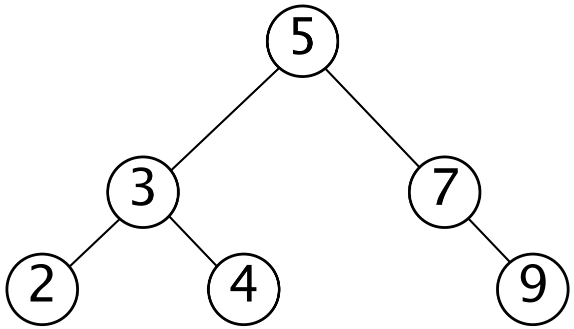


- Search for key 13:
  - $15 \rightarrow 6 \rightarrow 7 \rightarrow 13$

Searching for a key

find(x, k)

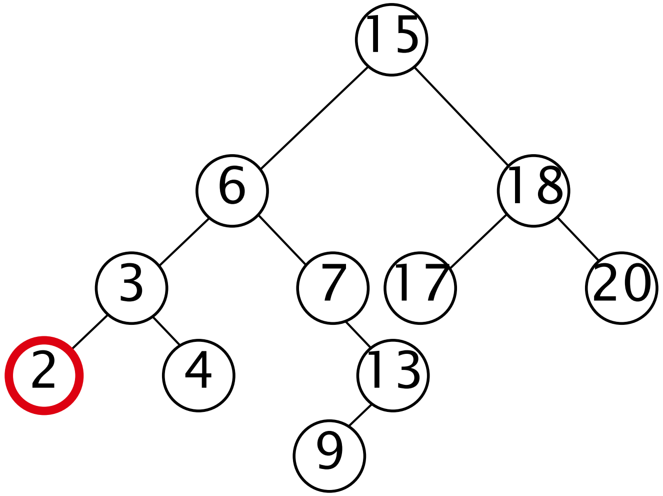
```
if x = NIL or k = key [x]
    return x
if k < key [x]
    return find(left [x], k )
else
    return find(right [x], k )
```



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

Finding the minimum

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



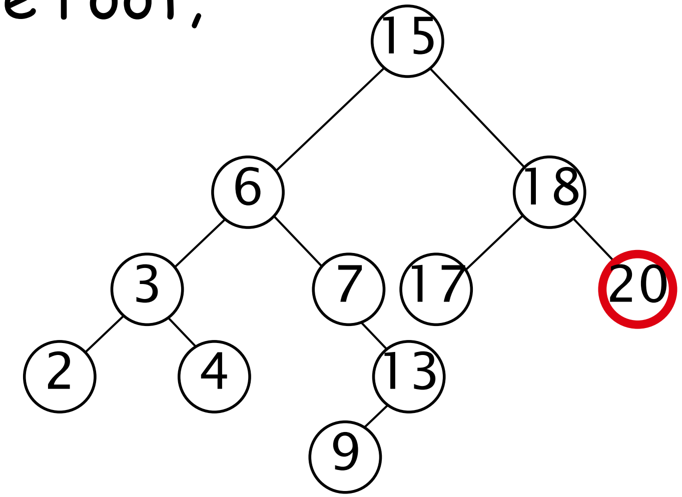
Minimum = 2

```
findMin(x)
    while left [x] ≠ NIL
        do x ← left [x]
    return x
```

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

Finding the maximum

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



Maximum = 20

findMax(*x*)

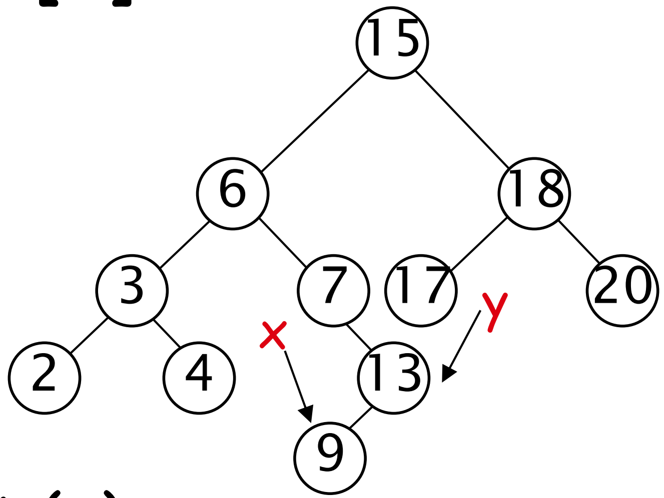
```
while right [x] ≠ NIL
        do x ← right [x]
return x
```

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

Successor

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

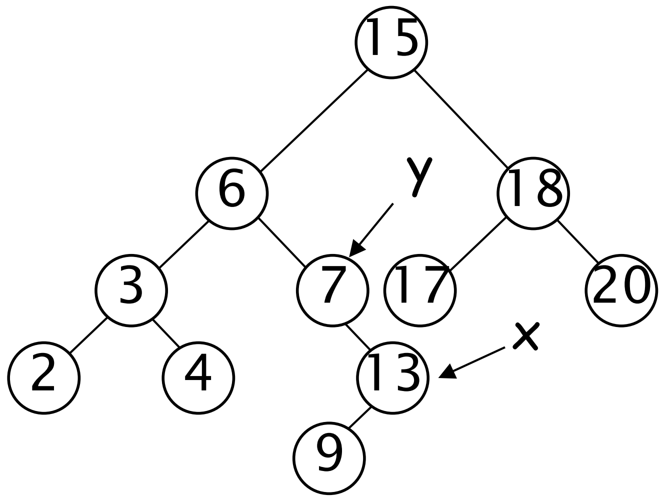
- E.g.: successor (15) = 17
  successor (13) = 15
  successor (9) = 13



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

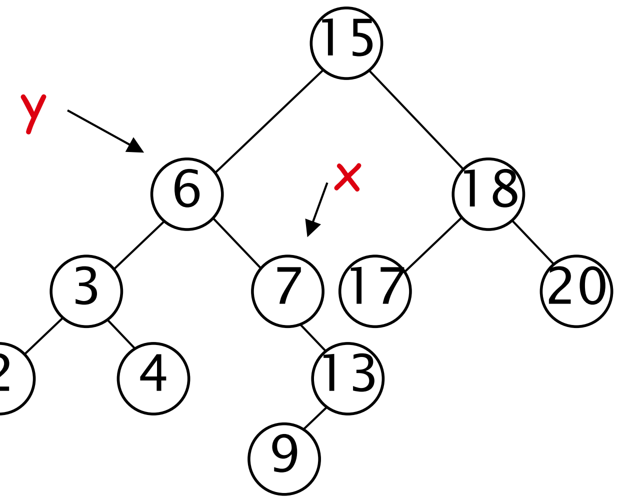


Running time: O(h), where h is the height of tree

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

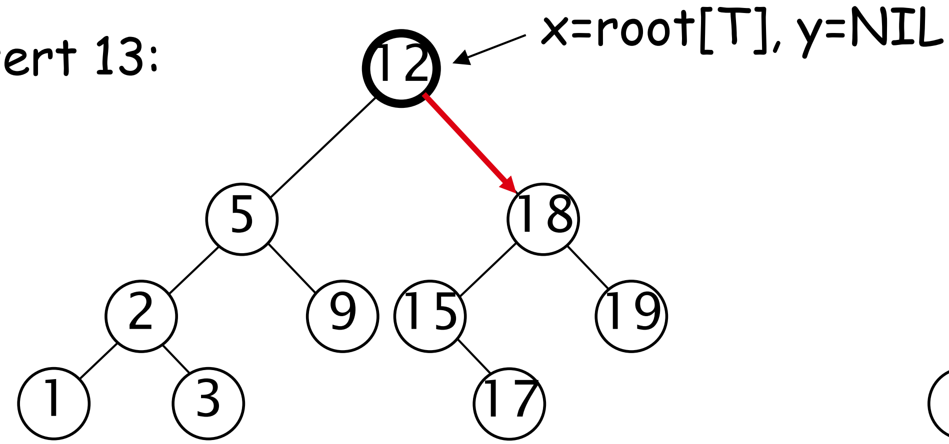
- Goal: Insert value v into a binary search tree

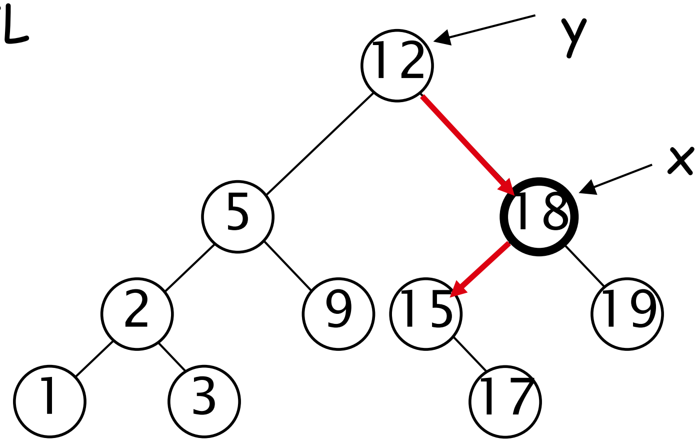
- Find the position and insert as a leaf:
  - If key [x] < v move to the right child of x,
    else move to the left child of x
  - When x is NIL, we found the correct position
  - If v < key [y] insert the new node as y's left child
    else insert it as y's right child

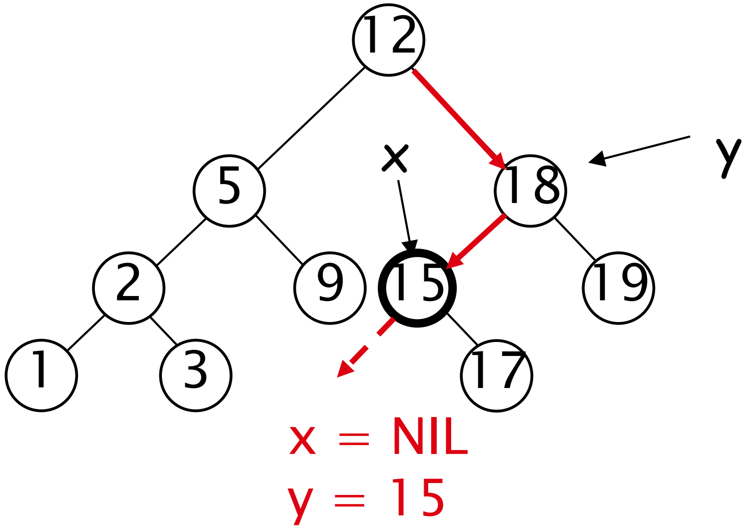
  - Beginning at the root, go down the tree and maintain:
    - Pointer x: traces the downward path (current node)
    - Pointer y: parent of x ("trailing pointer")

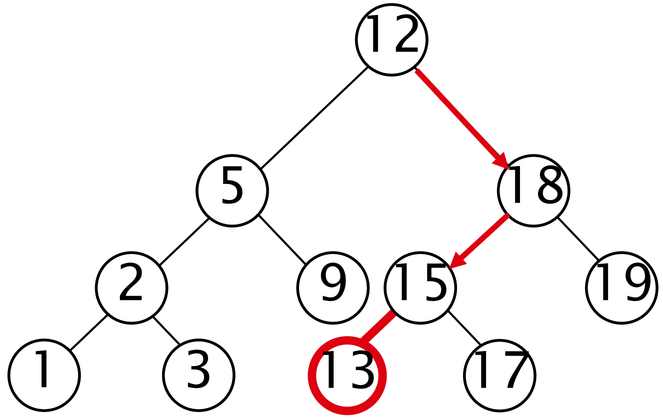
Example

Insert 13:



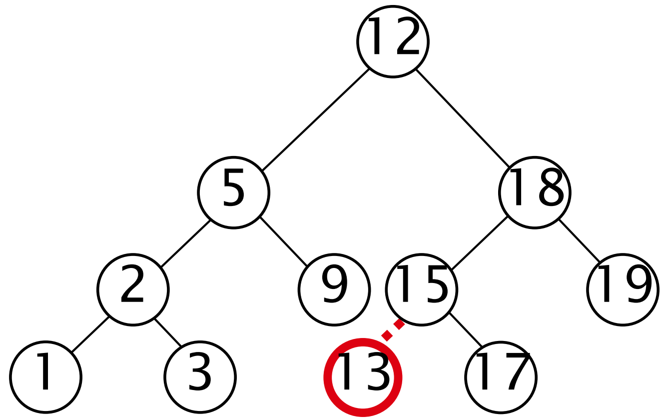






Insert algorithm

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



Best-case and worst-case time complexities?

Running time: $O(h)$

Exercise

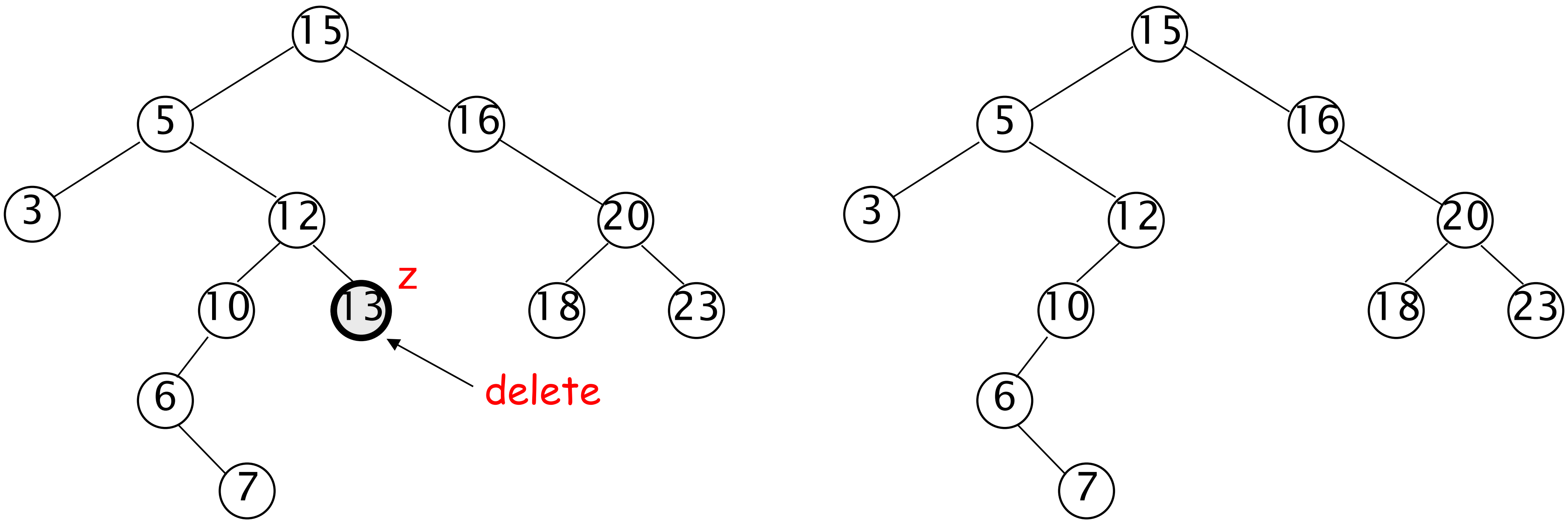
- Build a binary search tree for the following sequence

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

Deletion

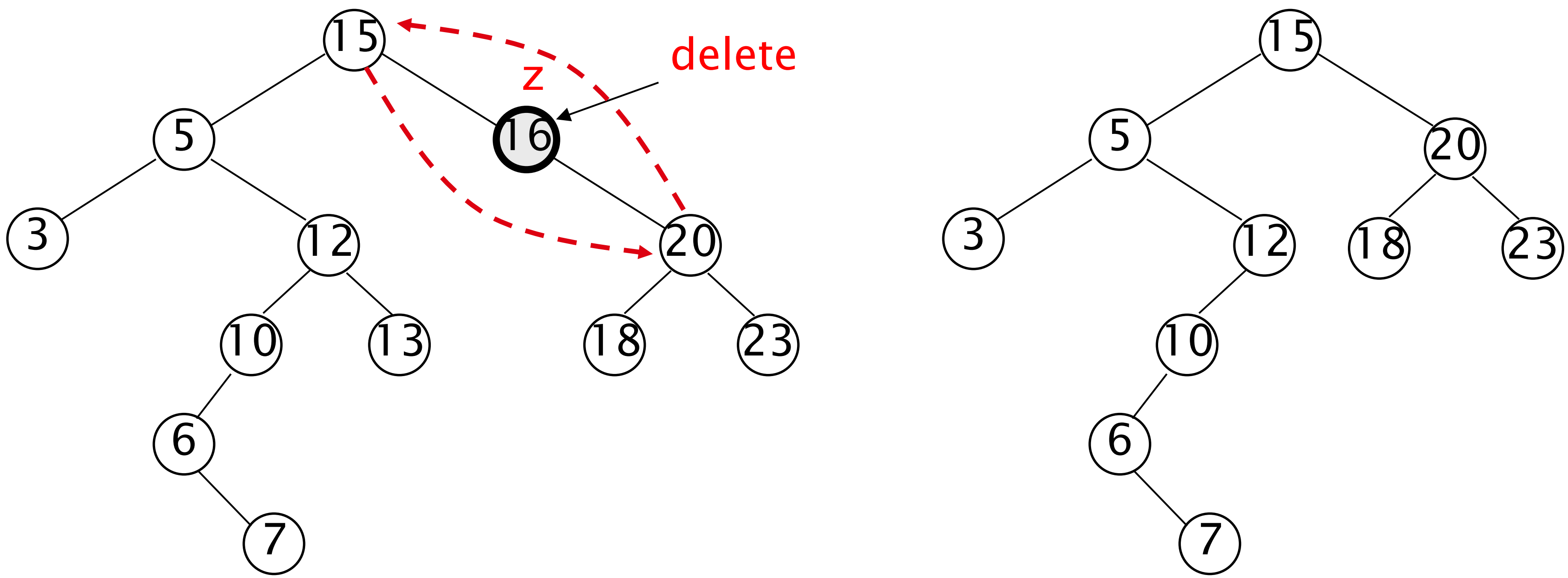
- Goal: Delete a given node z from a binary search tree

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



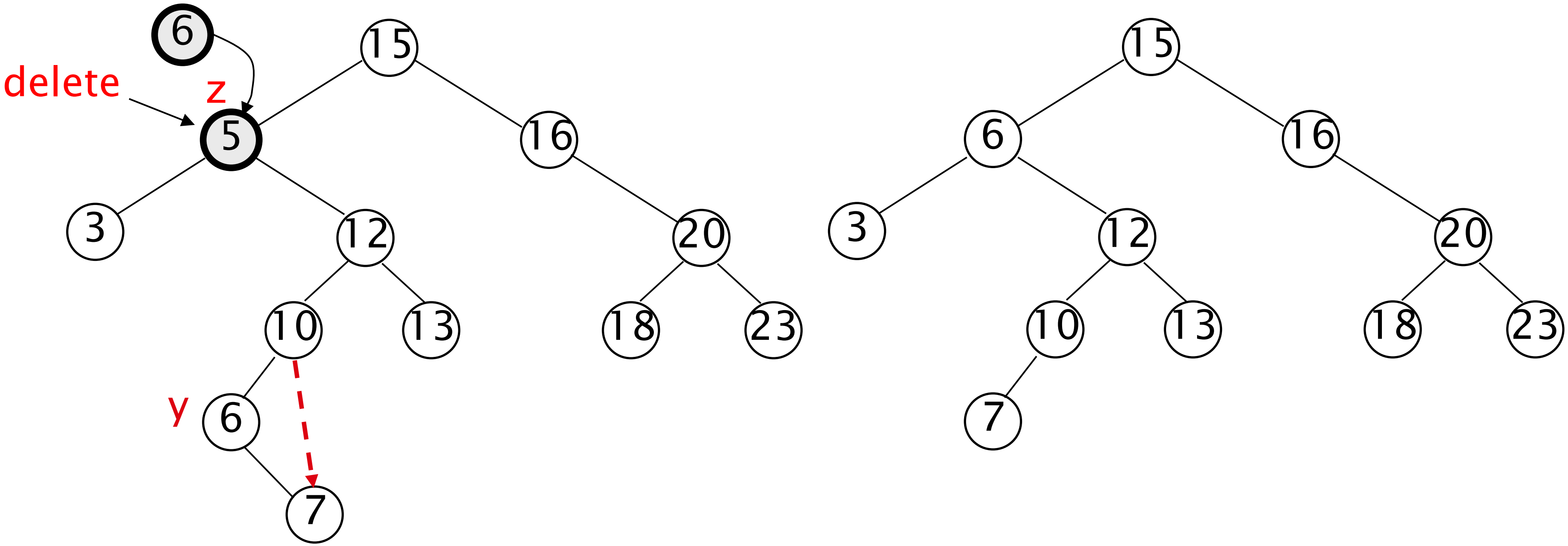
Deletion

- **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



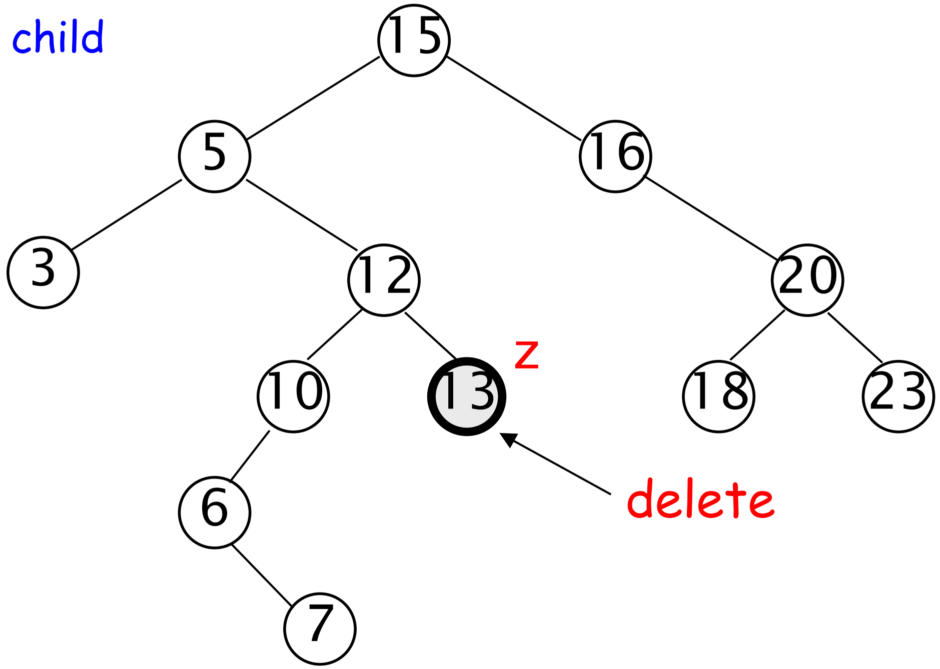
Deletion

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



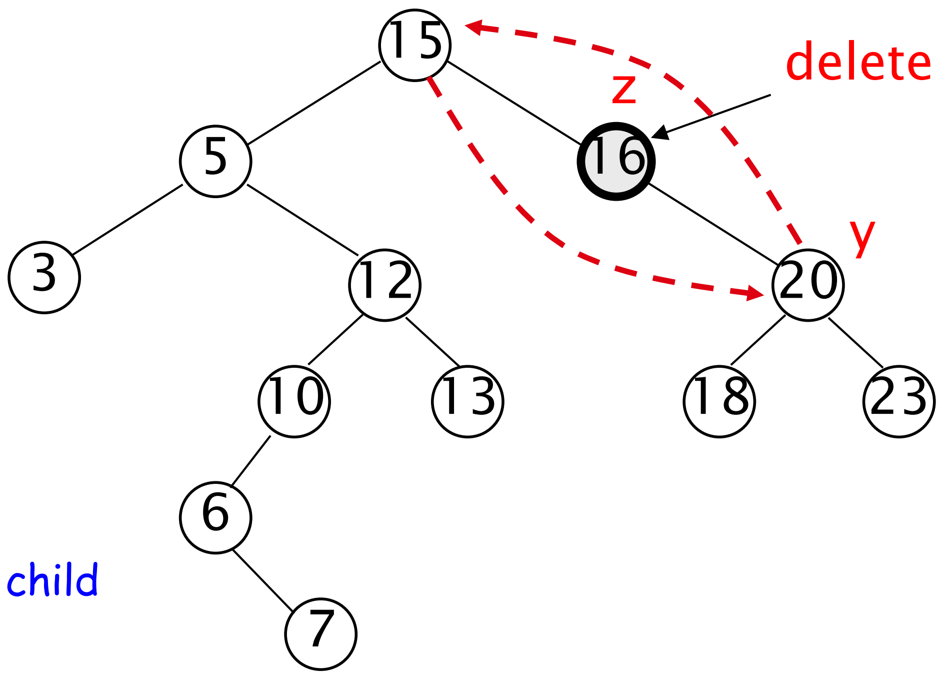
Deletion algorithm

```
if left[z] = NIL and right[z] = NIL    //z has no child
    if p[z] = NIL then root[T] = NIL
    if z = left[p[z]]
        left[p[z]] = NIL
    else
        right[p[z]] = NIL
```



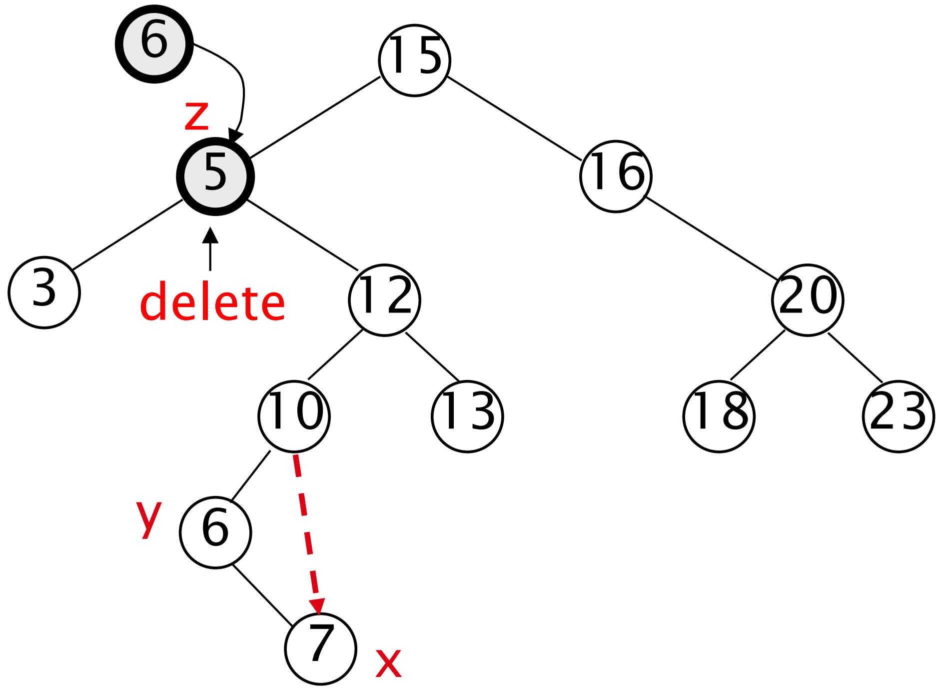
Deletion algorithm

```
if left[z] = NIL and right[z] ≠ NIL    //z has one right child
    y = right[z]
    if p[z] = NIL
        root[T] = y
    else
        p[y] = p[z]
        if z = left[p[z]]
            left[p[z]] = y
        else
            right[p[z]] = y
if left[z] ≠ NIL and right[z] = NIL    //z has one left child
    y = left[z]
    if p[z] = NIL
        root[T] = y
    else
        p[y] = p[z]
        if z = left[p[z]]
            left[p[z]] = y
        else
            right[p[z]] = y
```



Deletion algorithm

```
if left[z] ≠ NIL and right[z] ≠ NIL //z has two children
    y ← TREE-SUCCESSOR(z)        //left-most node in right tree
    if p[y] = z
        right[z] = right[y]
        if right[y] ≠ NIL
            p[right[y]] = z
    else
        if right[y] = NIL
            left[p[y]] ← NIL
        else
            x ← right[y]
            p[x] ← p[y]
            left[p[y]] ← x
    key[z] ← key[y] //copy y's data into z
```



Best/worst-case time complexities?

Summary

- Operations on binary search trees:
  - Search O(h)
  - Predecessor O(h)
  - Successor O(h)
  - FindMin O(h)
  - FindMax 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	Sorted-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

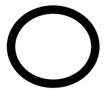
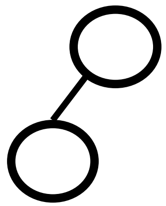
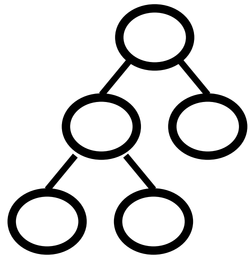
- After a series of delete operations, the above algorithm favors making the left sub-trees deeper than the right

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

Exercise 1: count leaves

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

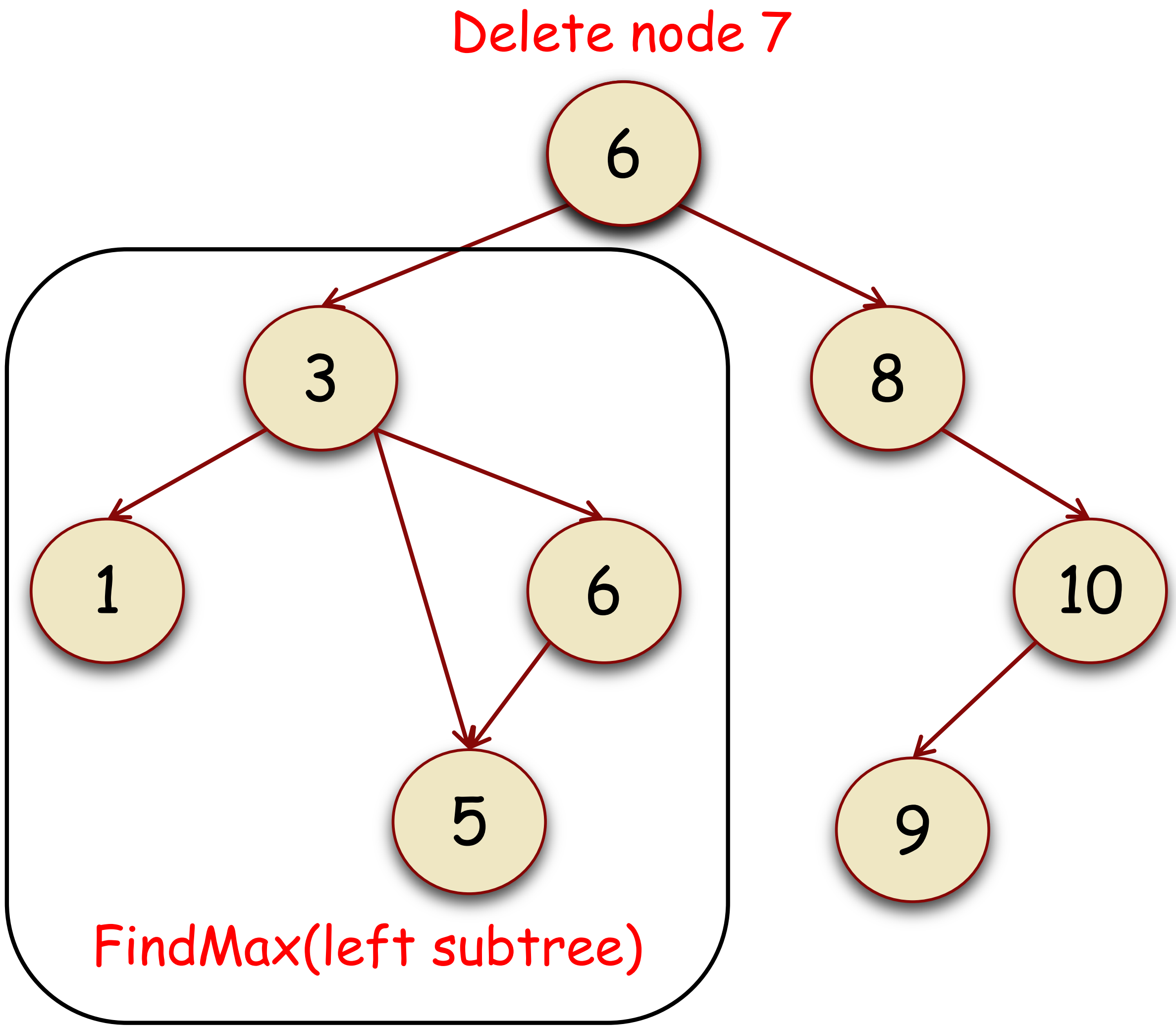
```
//To count the number of leaf nodes

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

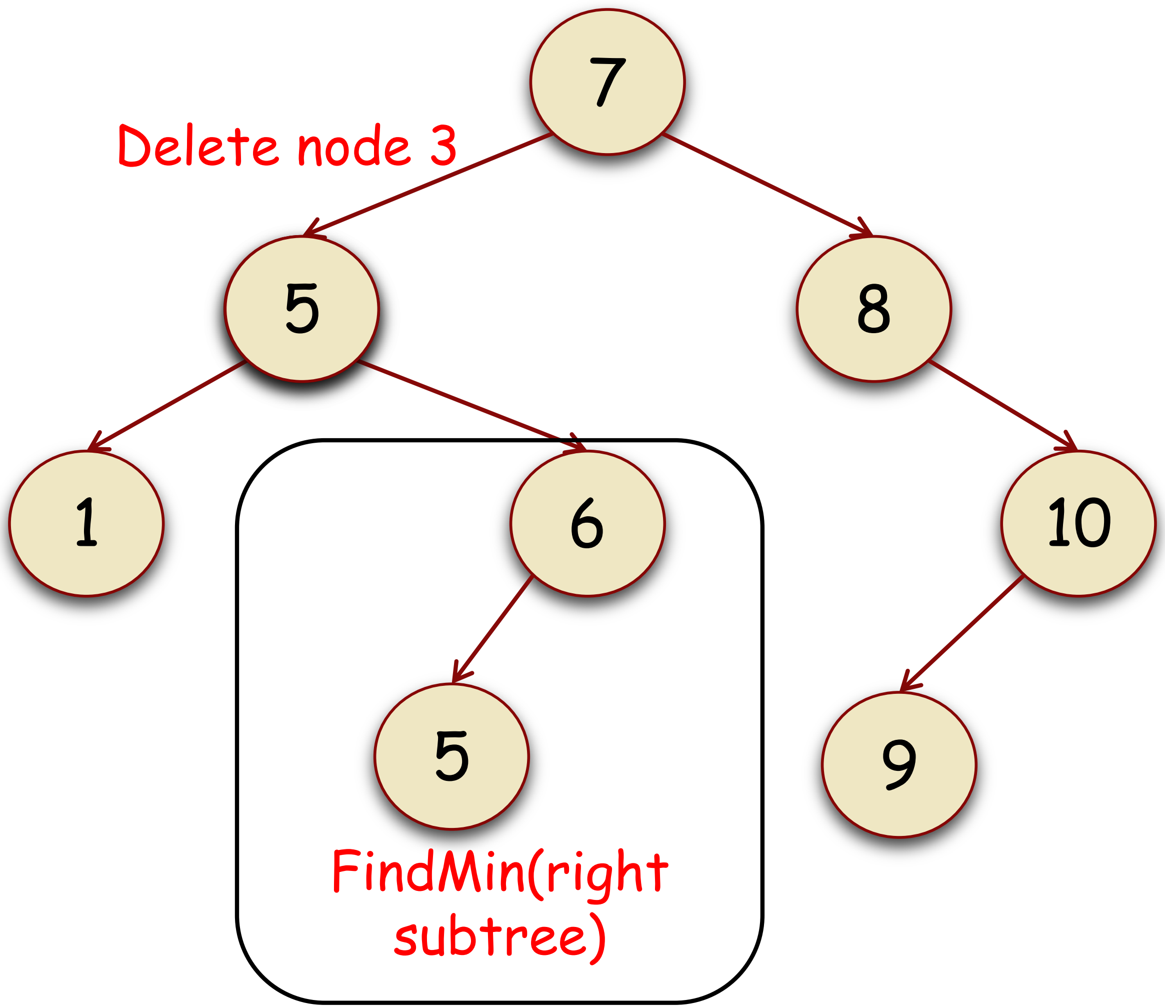
1. Locate the node
2. Find the rightmost node in its left subtree
3. Or find the leftmost node in its right subtree
4. Use the key of the node to replace its key
5. Delete the node



Exercise 2: delete the node with two children

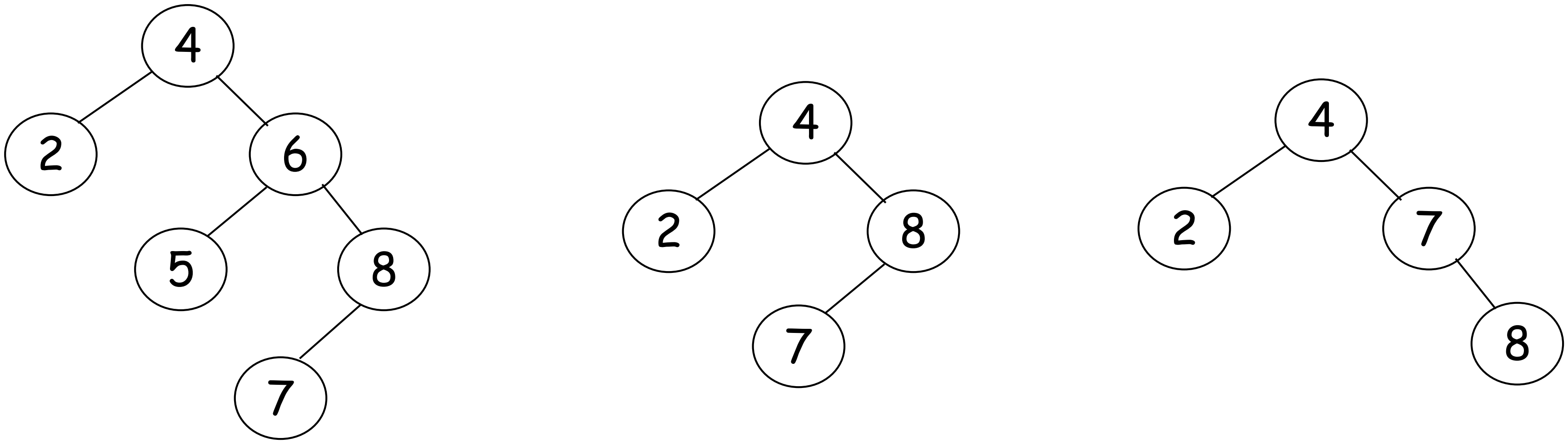
A bit complicated if we want to delete a NON-LEAF NODE with TWO children

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: operation commutative

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



Case 1: Delete 5 and then 6
Case 2: Delete 6 and then 5

Exercise 4: sorting with BST

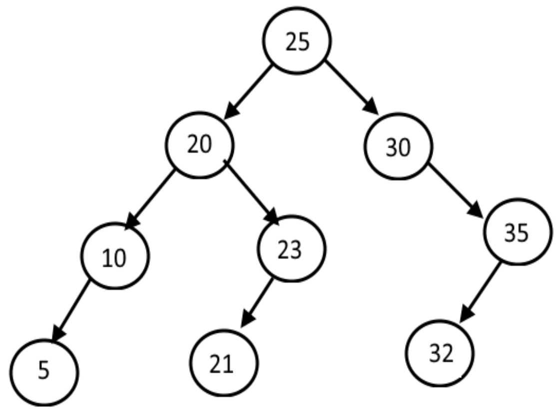
- How to sort an array of keys by building and traversing a BST?

```
Sort (A)
    for i = 1 to n
        insert(A[i])
inorder-tree-walk(root)
```

  - What are the worst case and best case running times?
  - In practice, how would this compare to other sorting algorithms?

Exercise 5: lowest common ancestor

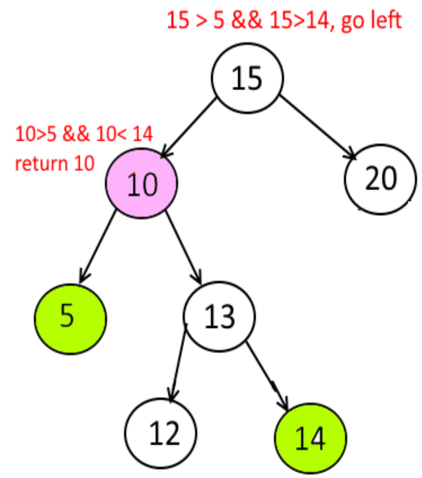
- Lowest common ancestor (LCA):
  - The LCA of two nodes n1 and n2 is a node X such that node X will be the lowest node who has both n1 and n2 as its descendants
  - Given a BST and two nodes n1 and n2, how to find their LCA?



Lowest Ancestor Ancestor (5, 21) = 20
Lowest Ancestor Ancestor (10, 30) = 25
Lowest Ancestor Ancestor (5, 32) = 25
Lowest Ancestor Ancestor (10, 23) = 20

Approach:
1) Start will the root
2) If root>n1 and root>n2 then lowest common ancestor will be in left subtree
3) If root<n1 and root<n2 then lowest common ancestor will be in right subtree
4) If Step 2 and Step 3 is false then we are at the root which is LCA, return it

n1 = 5, n2 = 14



Lowest Common Ancestor of Nodes 5 and 14 is : 10

Recommended reading

- Reading this week
  - Chapter 12, textbook
- Next lecture
  - AVL-tree: Chapter 12, textbook