Data Structures and Algorithms

Lecture 9: Binary Search Trees

Department of Computer Science & Technology
United International College

Outline

- Trees
  - Basic Concepts
- Binary Trees
  - Tree Traversal
- Binary Search Trees
 - Find
  - Insert
  - Delete

Trees

- A tree is a collection of nodes
  - The collection can be empty
  - (recursive definition) If not empty, a tree consists of a distinguished node r (the root), and zero or more nonempty subtrees T_1, T_2, ..., T_k, each of whose roots are connected by a directed edge from r

Subtree of A
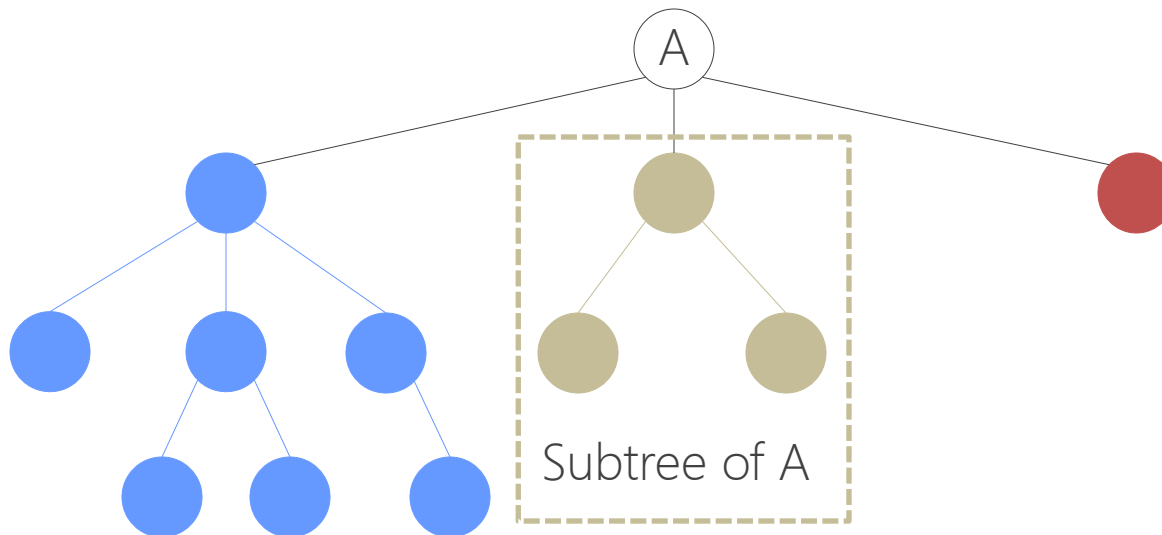
Some Terminologies

- Root and Leaf
- Child and Parent
  - Every node except the root has one parent
  - A node can have an zero or more children
  - A leaf node has no children
- Sibling
  - nodes with same parent


Siblings

Siblings
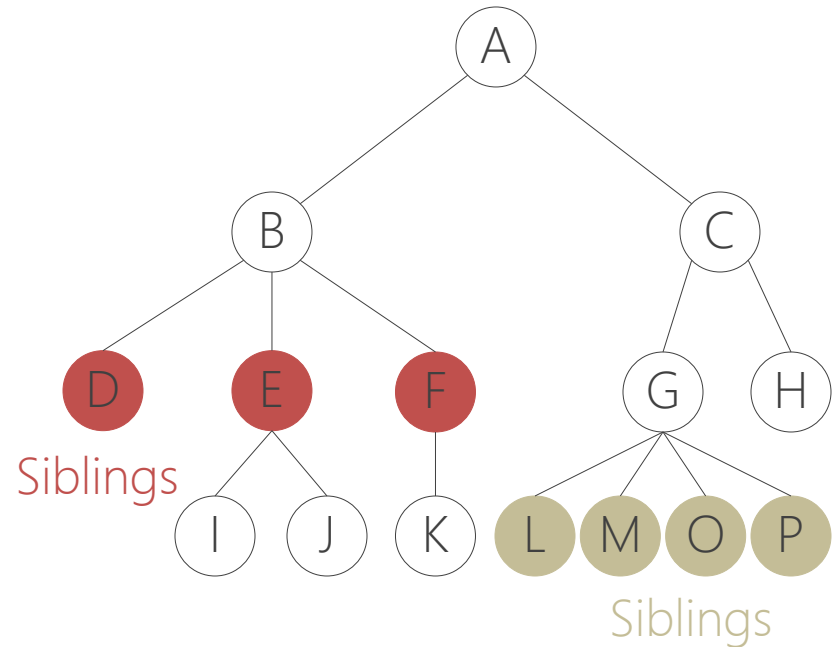
We are Family

More Terminologies

- Path
  - a sequence of edges
- Length of a path
- Depth of a node
  - length of the unique path to the root
- Height of a node
  - length of the longest path to a leaf
- Tree height
  - the height of the root
  - the depth of the deepest leaf
- Ancestor and descendant
  - If there is a path from n1 to n2
  - n1 is an ancestor of n2, n2 is a descendant of n1
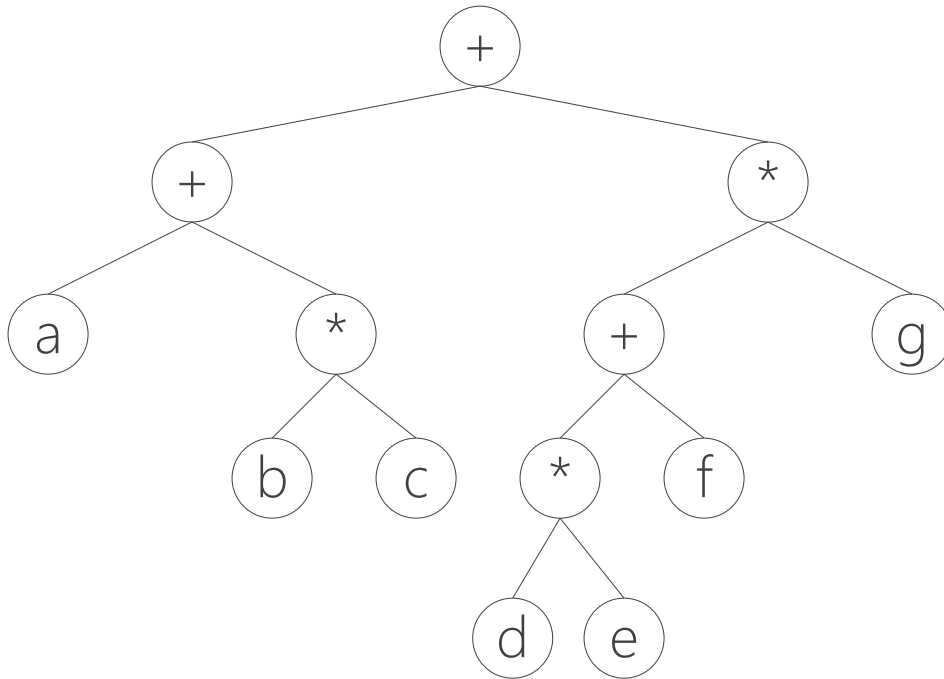 - Proper ancestor and proper descendant
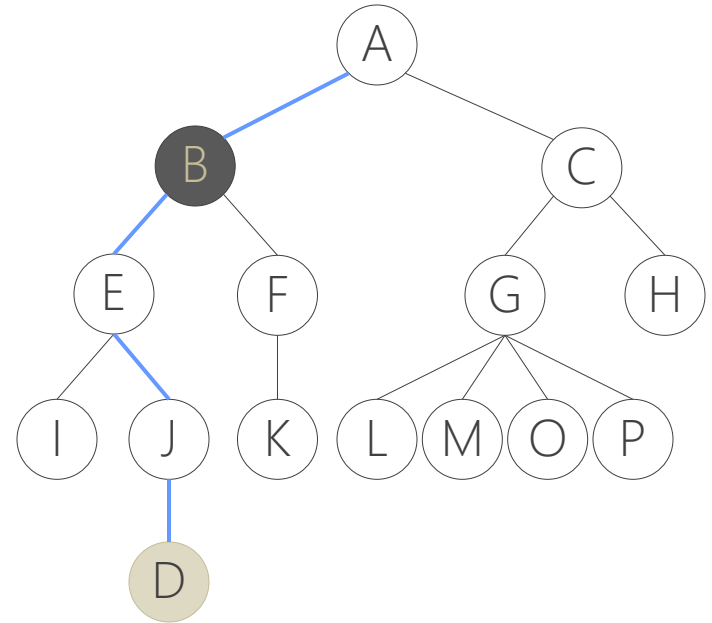
Length of the blue path = 4
Depth(B) = 1
Height(B) = 3
B is D's Ancestor
D is B's Descendant
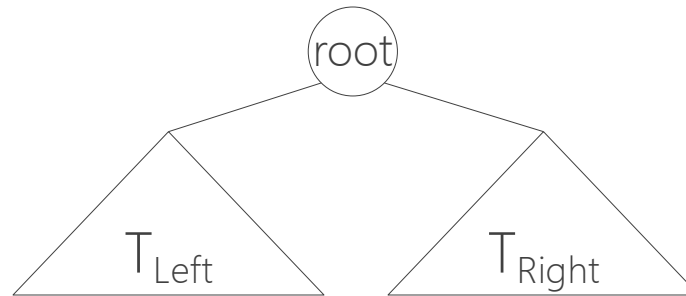
Example: UNIX Directory
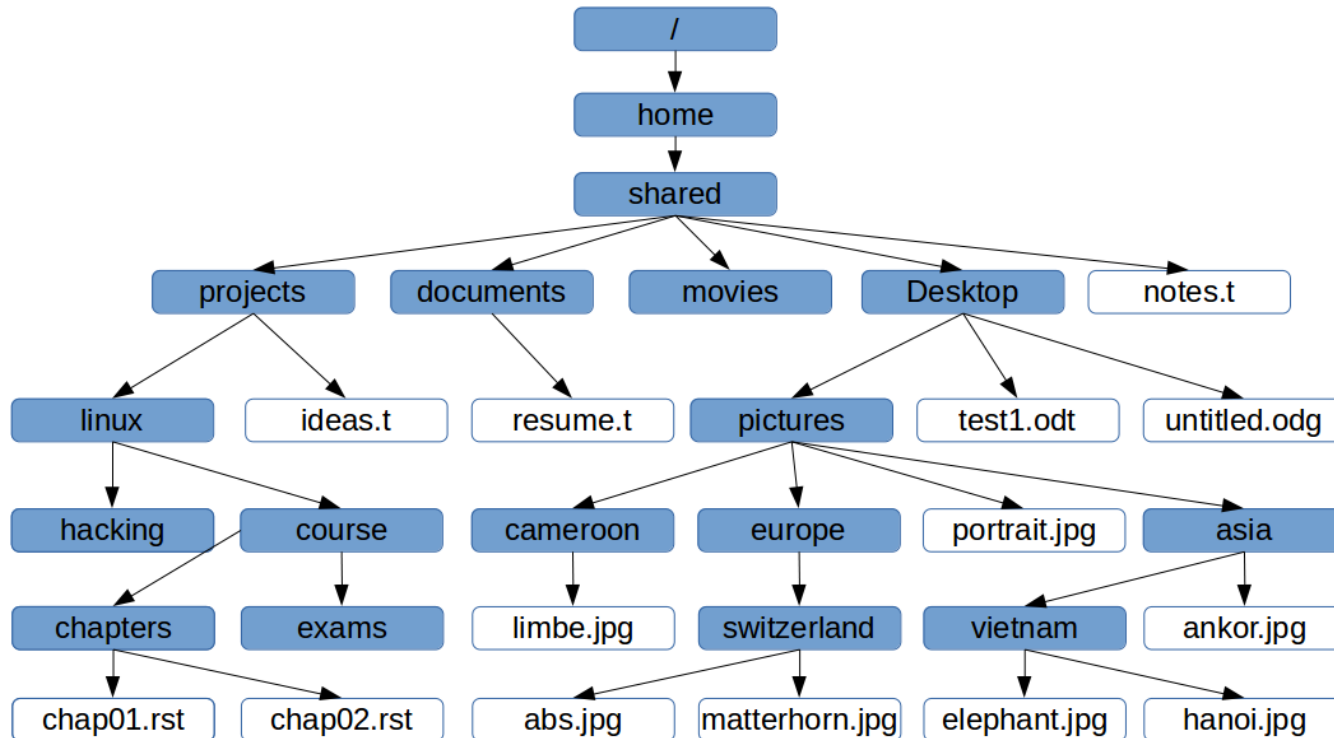
Example: Expression Trees

Expression tree for: (a + b*c) + (d*e + f) * g

- Leaves are operands (constants or variables)
- The internal nodes contain operators
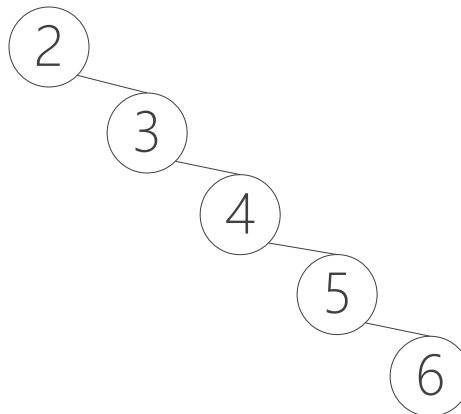- Will not be a binary tree if some operators are not binary

Binary Trees

- A tree in which no node can have more than two children

Generic binary tree

- The depth of an "average" binary tree is considerably smaller than N, even though in the worst case, the depth can be as large as N – 1.


Worst-case binary tree

Binary Tree Traversal

- Three strategies for tree nodes enumeration
- Pre-order traversal
  - Recursive algorithm
  - First visit the root, then the left subtree, then the right
- In-order traversal
  - Recursive algorithm
  - First visit the left subtree, then the root, then the right subtree
- Post-order traversal
  - Recursive algorithm
  - First visit the left subtree, then the right, then the root

Pre-order Traversal

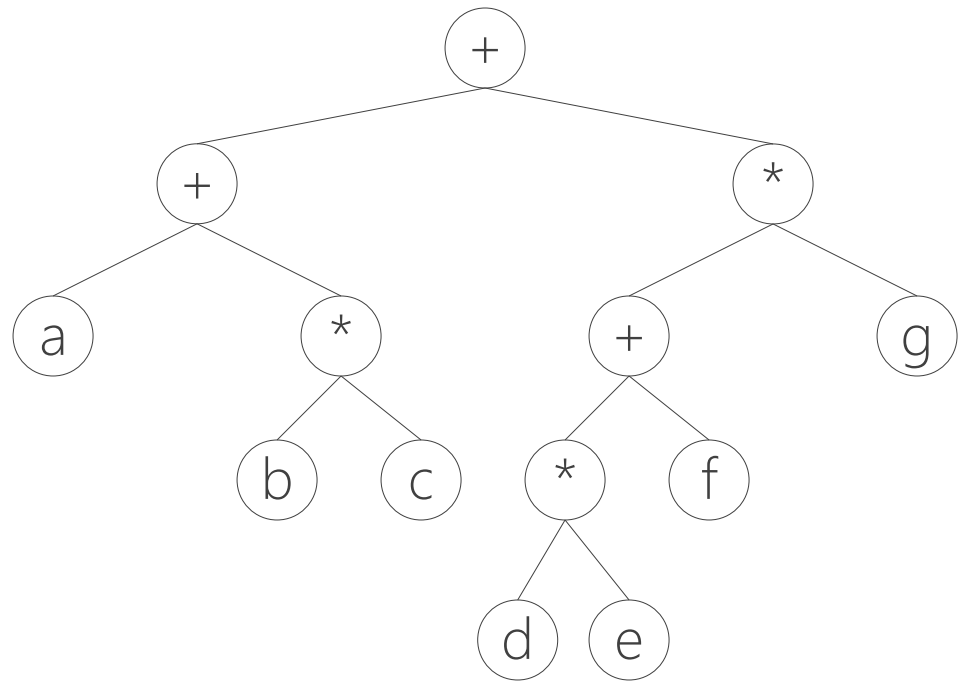- node, left, right
- prefix expression

  ++a*bc*+*defg
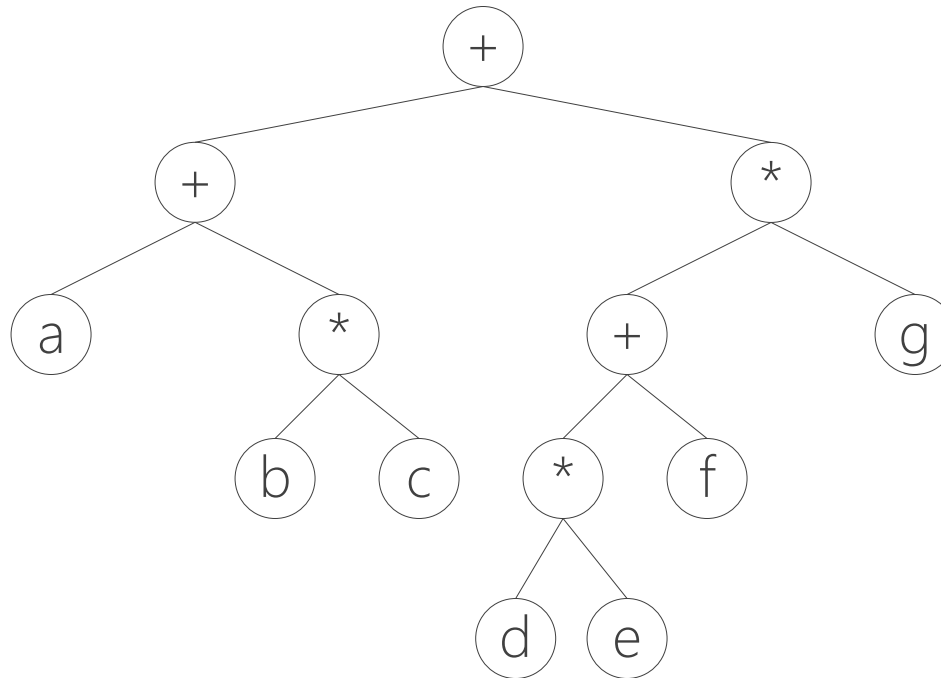
Expression tree for: (a + b*c) + (d*e + f) * g

Post-order Traversal

- left, right, node
- postfix expression

  abc*+de*f+g*+

In-order Traversal

- left, node, right
- infix expression

  a+b*c+d*e+f*g

Expression tree for: (a + b*c) + (d*e + f) * g

Pseudo Code for Pre-order, In-order and Post-order

PREORDER(root)

1. IF root = Null
2. return
3. PRINT(root)
4. PREORDER(LEFT(root))
5. PREORDER(RIGHT(root))

INORDER(root)

1. IF root = Null
2. return
3. INORDER(LEFT(root))
4. PRINT(root)
5. INORDER(RIGHT(root))

POSTORDER(root)

1. IF root = Null
2. return
3. POSTORDER(LEFT(root))
4. POSTORDER(RIGHT(root))
5. PRINT(root)

Node Struct of Binary Tree

- Possible operations on the Binary Tree ADT
  - Parent, left, right, sibling, root, etc
- Implementation
 - Because a binary tree has at most two children, we can keep direct pointers to them

Node
- data: Object - left: Node - right: Node
+ Node() + ...

A binary tree which offers directed search

BINARY SEARCH TREES

Binary Search Trees (BST)

- Binary search tree property:
  - For every node X:
    - All the keys in its left subtree are smaller than the key value in X
    - All the keys in its right subtree are larger than the key value in X
- Pre-assumption
  - Objects are stored in tree nodes
    - Book information: ISBN, Title, author, abstract, price, …
  - The keys of the objects are used for search and comparison
    - ISBN
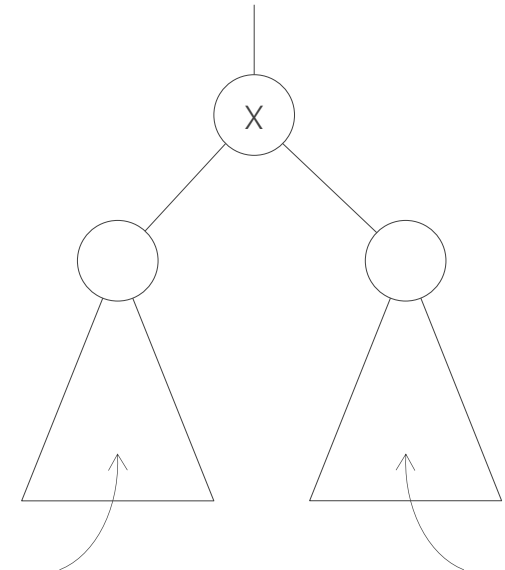
For any node y in this subtree
$key(y) < key(x)$

For any node z in this subtree
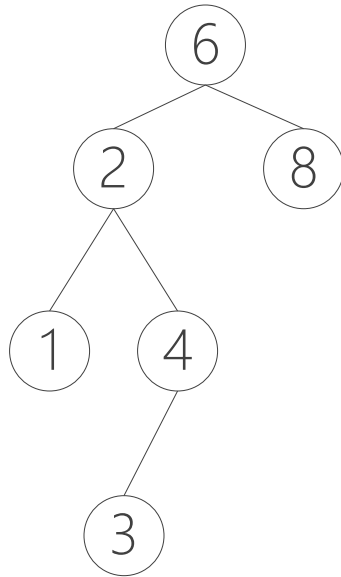$key(z) > key(x)$

No Duplicates!

For easy demonstration, we store just

integer keys in the tree nodes, but

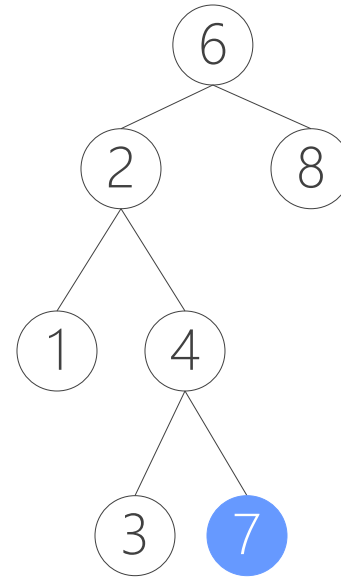be noted that in practice, it is usually

OBJECTS WITH KEYS

that are stored and are later inserted, deleted and searched.
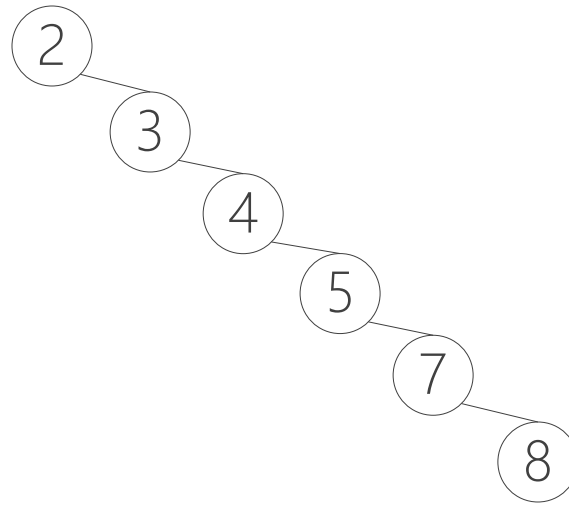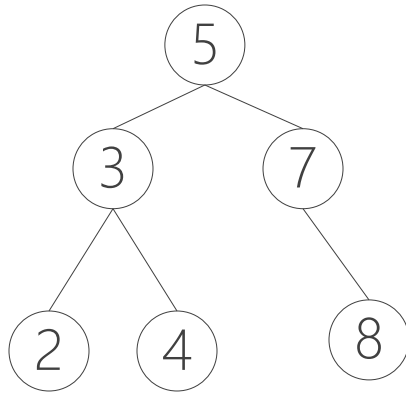
Binary Search Tree Example
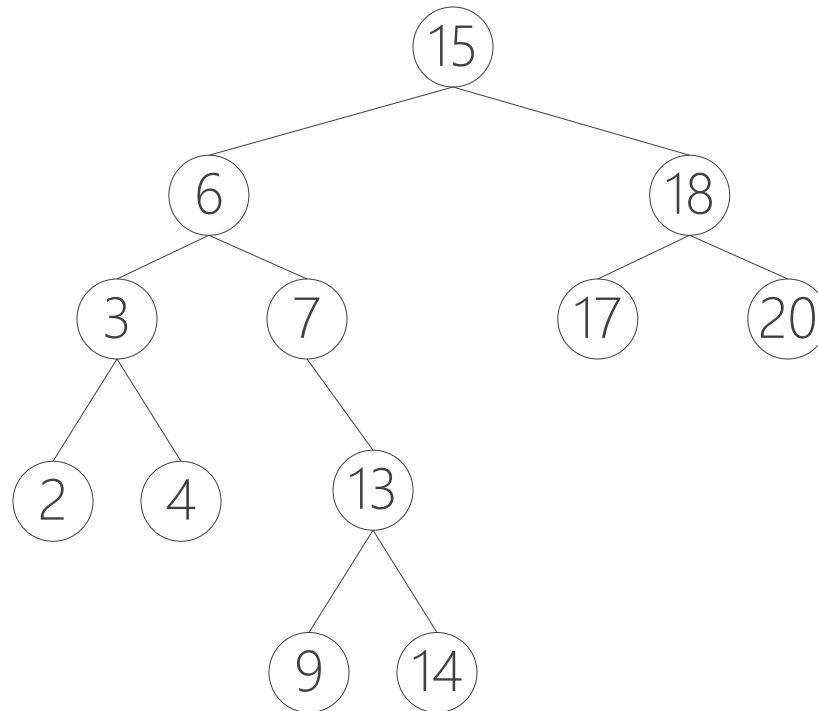

A binary search tree

Not a binary search tree
WHY?

Binary Search Trees

The same set of keys may have different BSTs

- The order of node insertion affects the shape of the tree
- Maximum depth of a node is n-1

# In-order Traversal of BST

2, 3, 4, 6, 7, 9, 13, 14, 15, 17, 18, 20
A sorted list!

Operations on Binary Search Trees

- **SEARCH**
- **FINDMIN / FINDMAX**
- **INSERT**
- **DELETE**

Searching BST

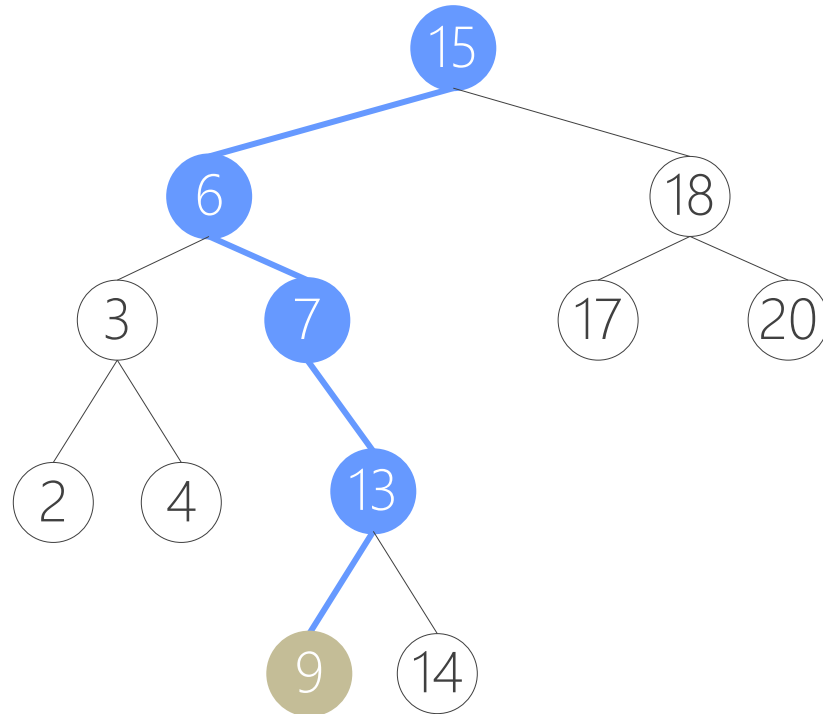- The current root is 15
  - If we are searching for 15, then we are done.
  - If we are searching for a key < 15, then we should search in the left subtree.
  - If we are searching for a key > 15, then we should search in the right subtree.

```
         ( 15 )
          root
         /    \
        /      \
       /        \
      /          \
   /__\        /__\
   <15          >15
```
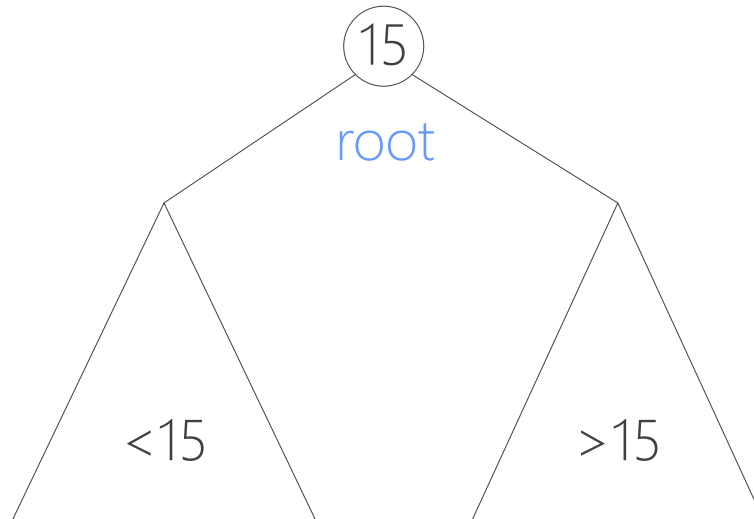
Directed Search

Search for 9:

1. Compare 9:15, go left
2. Compare 9:6, go right
3. Compare 9:7, go right
4. Compare 9:13, go left
5. Compare 9:9, found it!

Pseudo Code for Search

- FIND(root, x)
  - Searches the subtree rooted at root
  - Returns a pointer to the node whose key is x
  - Returns Null if no such node exists
- Time complexity: O(tree height)

FIND(root, x)
1. IF root=Null
2.   return Null
3. IF root.key=x
4.   return root
5. IF root.key>x
6.   return FIND(root.left, x)
7. return FIND(root.right, x)

findMin / findMax

- Goal: return the node containing the smallest (largest) key in the tree
- Algorithm: Start at the root and go left (right) as long as there is a left (right) child. The stopping point is the smallest (largest) element
- Time complexity: O(tree height)

FIND-MIN(root)
1. IF root=Null
2.   return Null
3. IF root.left=Null
4.   return root
5. return FIND-MIN(root.left)

FIND-MAX(root)
1. IF root=Null
2.   return Null
3. IF root.right=Null
4.   return root
5. return FIND-MAX(root.right)

Insertion

- Insert(root, x)
  - Proceed down the tree as you would with a find
  - If x is found, do nothing (reject duplicates)
  - Otherwise, insert x at the last spot on the path traversed
- Time complexity = O(tree height)
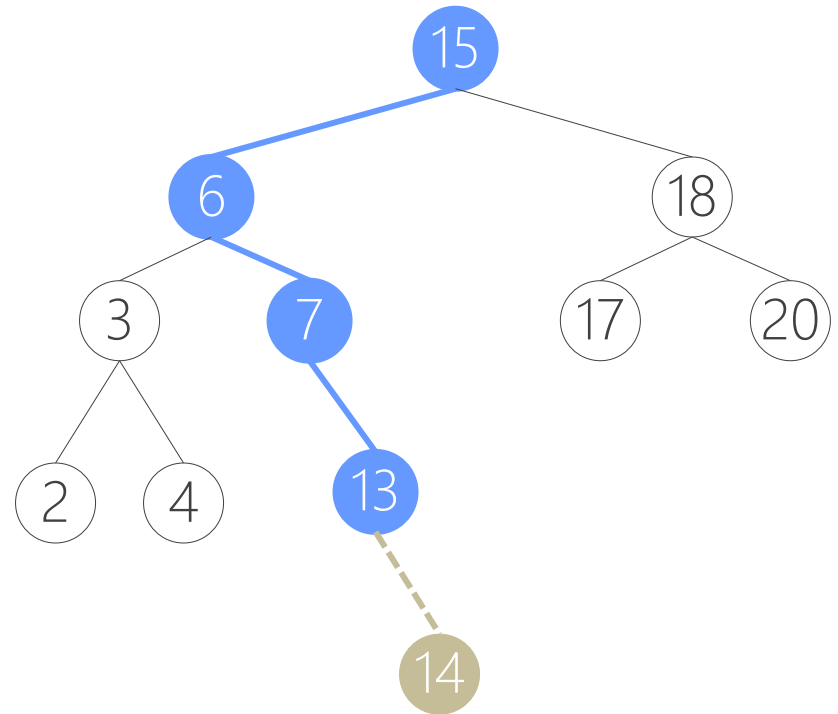
Insert(root, 14)

Pseudo Code for Insertion

INSERT(root, x)

// returns the (updated) root of the subtree

1.   IF root=Null
2.   return CREATE-NODE(x)
3.   IF x<root.key
4.   root.left = INSERT(root.left, x)
5.   ELSE IF x>root.key
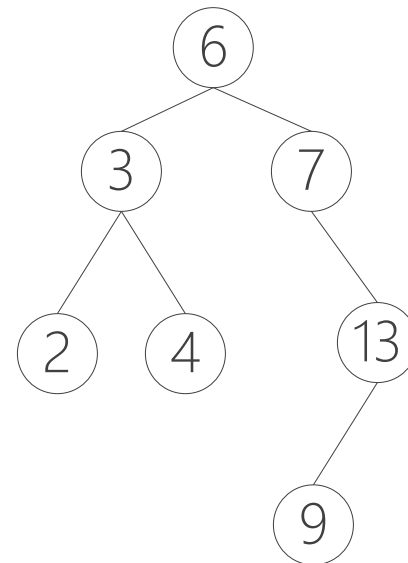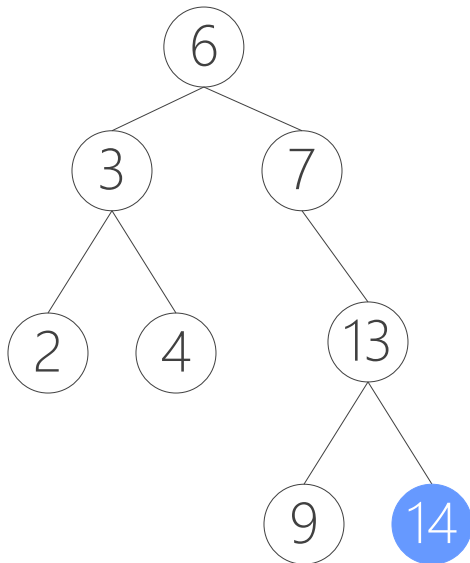6.   root.right = INSERT(root.right, x)
7.   return root

Deletion

- When we delete a node, we need to consider how we take care of the children of the deleted node.
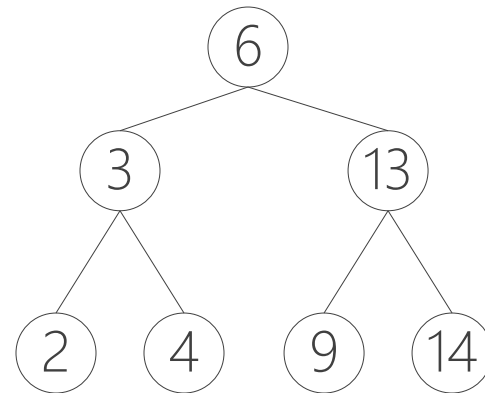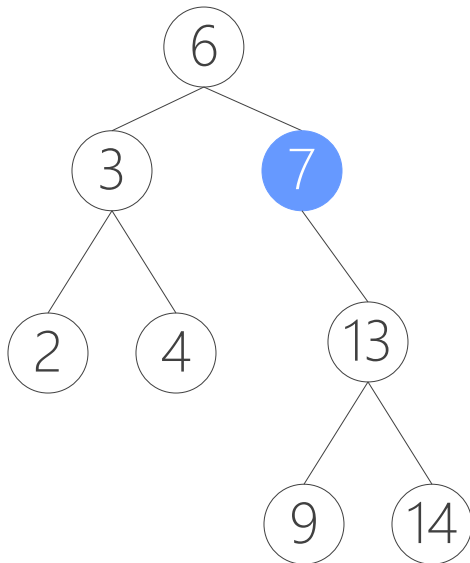- This has to be done such that the property of the search tree is maintained.

Three Delete Cases

- Case 1: the node is a leaf
 - Delete it immediately

Three Delete Cases

- Case 2: the node has one child
  - Adjust a pointer from the parent to bypass that node

Three Delete Cases

- Case 3: the node has two children
  - Replace that node with the minimum node in the right subtree
    - Or replace it with the maximum node in the left subtree
  - This invokes removal of the minimum node
    - It's case 1 or 2. WHY?
- Time complexity = O(tree height)
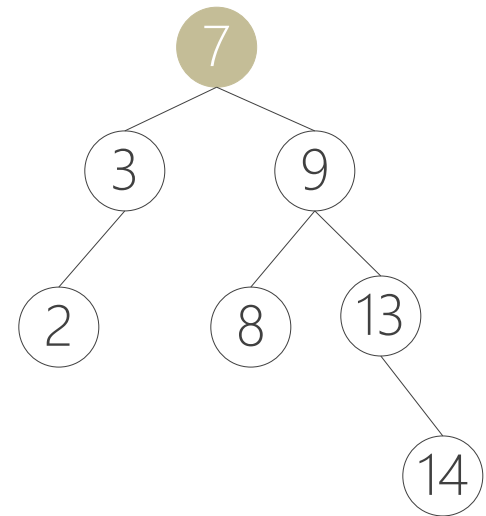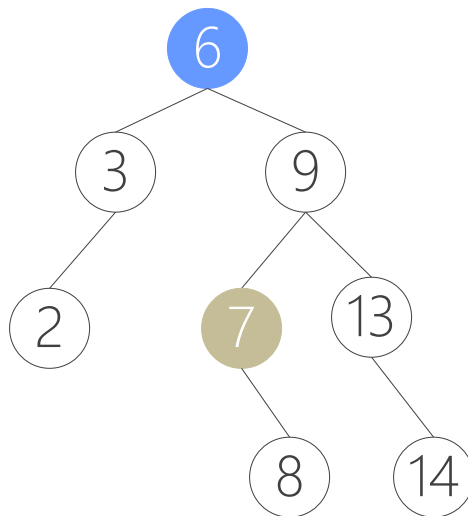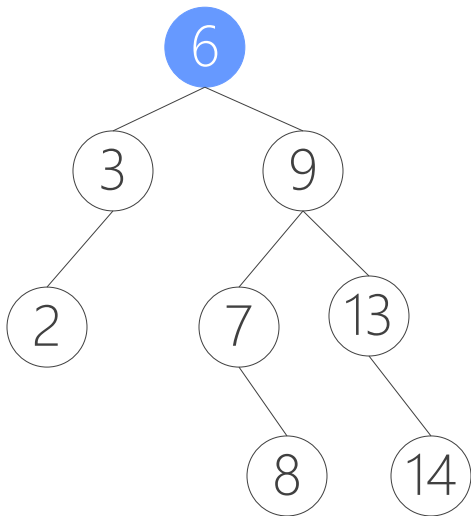
Pseudo Code?

The cost of search, insert and delete are all bounded by the **TREE HEIGHT** which is $O(n)$ in the worst case. And O(n) is

NOT FAST ENOUGH!

Task

- Given Node.java, BinaryTreePrinter.java and the skeleton of BST.java, complete BST.java
  - Node.java: implements the class for a BST node
  - BinaryTreePrinter.java : implements a method to print out a binary tree
  - BST.java: implements the methods defined in the BST data structure
    - Read the skeleton and complete all the methods
    - This is the only file that you are going to modify
    - You may add auxiliary functions if there is a need
    - a main function is provided for testing purpose
- Submit BST.java to iSpace.