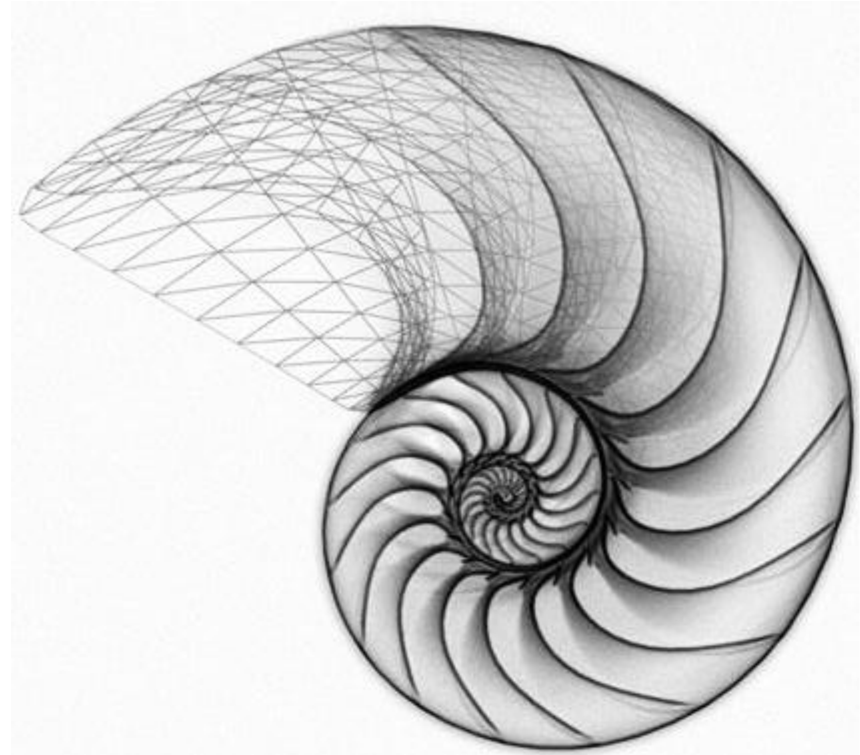


Lecture 12

Binary Search Tree

CSE225: Data Structures and Algorithms

Motivation

Key difference between sorted list and unsorted list.

	Unsorted List	Sorted List
`RetrieveItem`	O(N)	O(logN)
`InsertItem`	O(1)	O(N)

Motivation

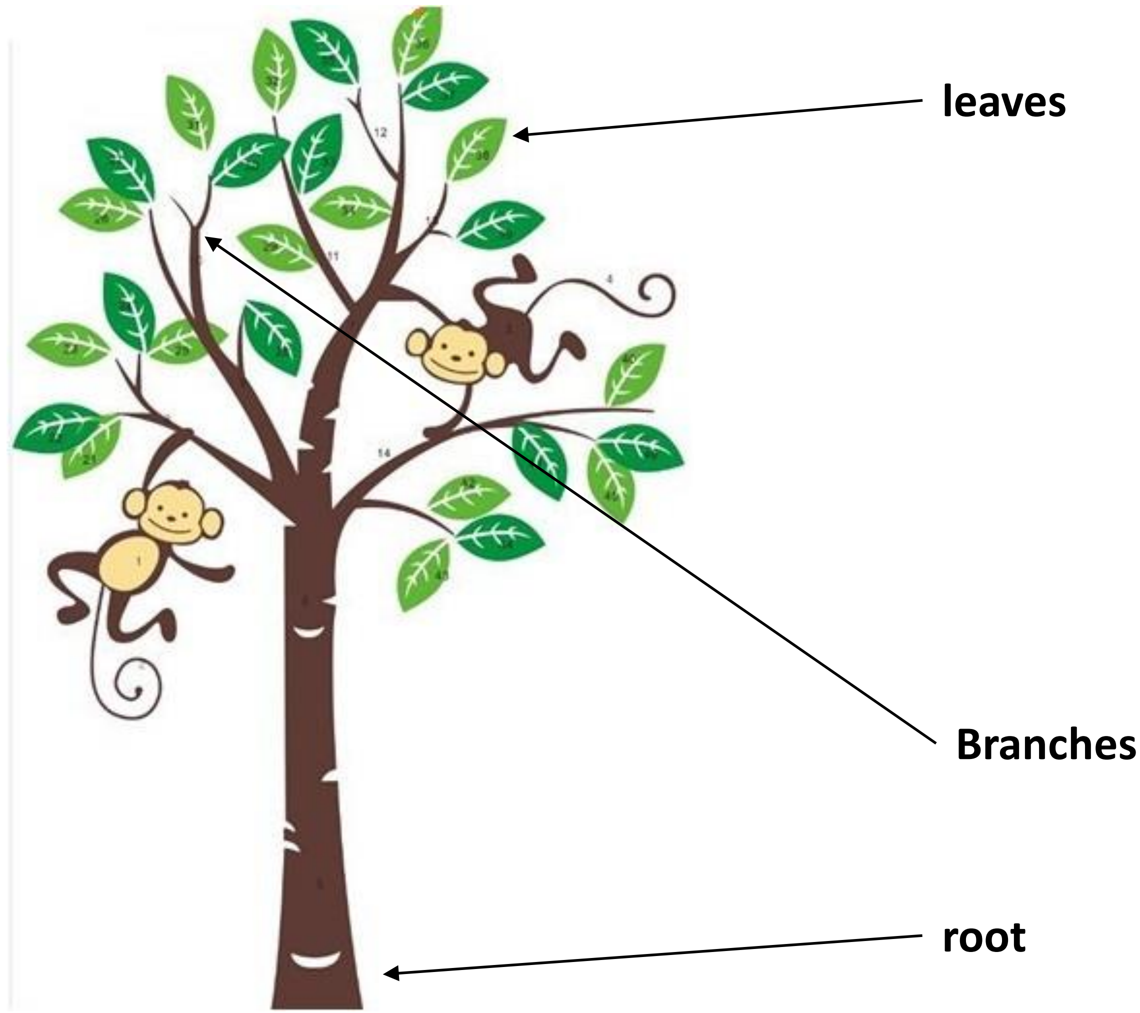
Key difference between sorted list and unsorted list.

	Unsorted List	Sorted List
`RetrieveItem`	O(N)	O(logN)
`InsertItem`	O(1)	O(N)

Can we improve this operation?

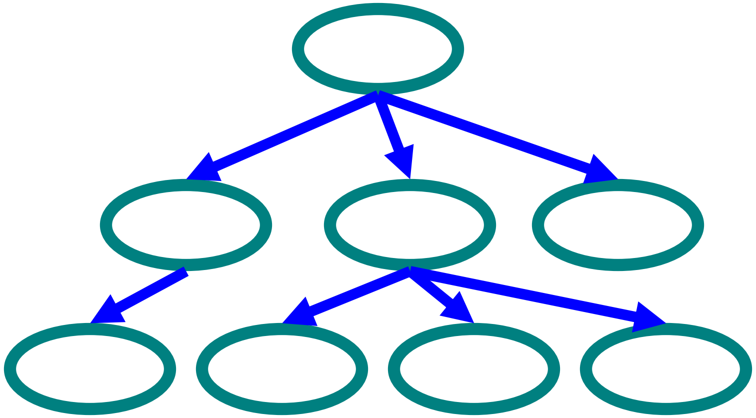
Tree Data Structure

Tree Data Structure

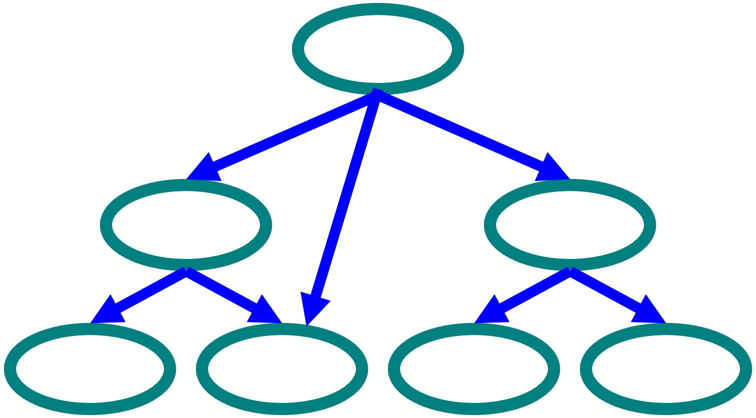


Tree Data Structure

- Tree
  - Collection of nodes linked to each other (similar to linked lists)
  - Each node can have 0 or more children (successor nodes)
  - Each node (except the root) has **exactly one** parent (predecessor node)
    - This means that there is exactly one path to go from the root to any other node



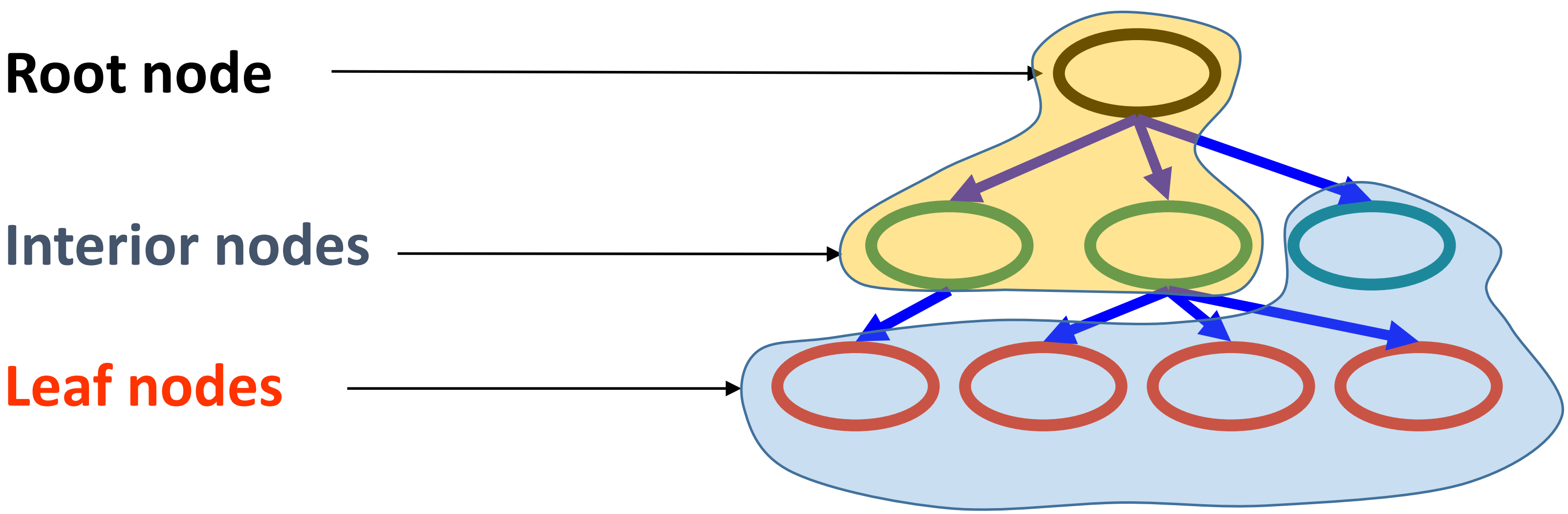
Tree



Not Tree

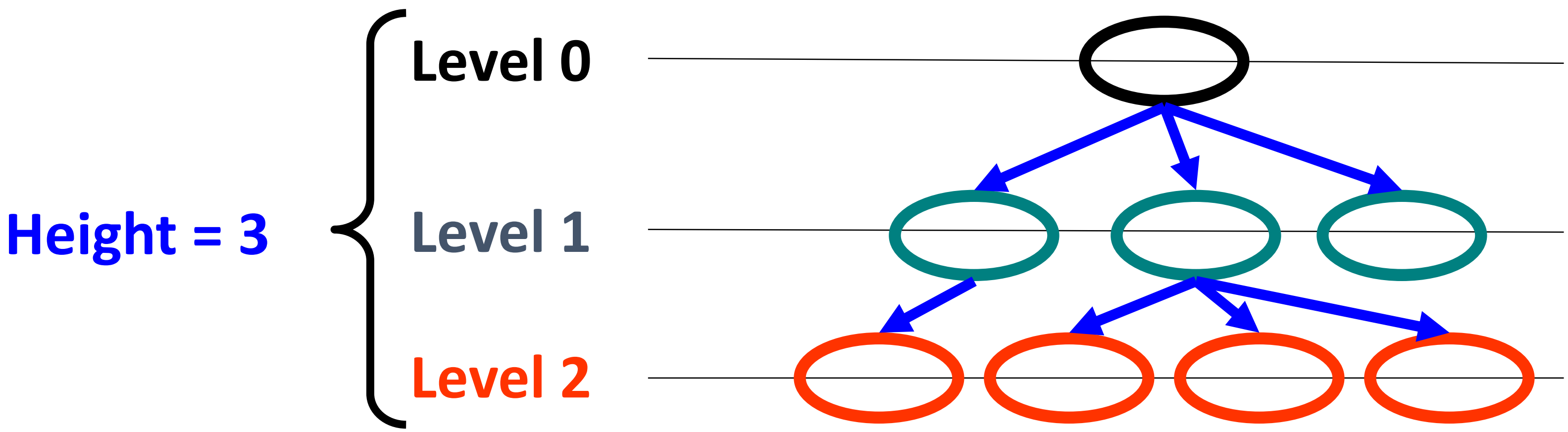
Tree Data Structure

- Terminology
  - Root ⇒ node with no parent
  - Leaf ⇒ node(s) with no child
  - Interior ⇒ non-leaf nodes



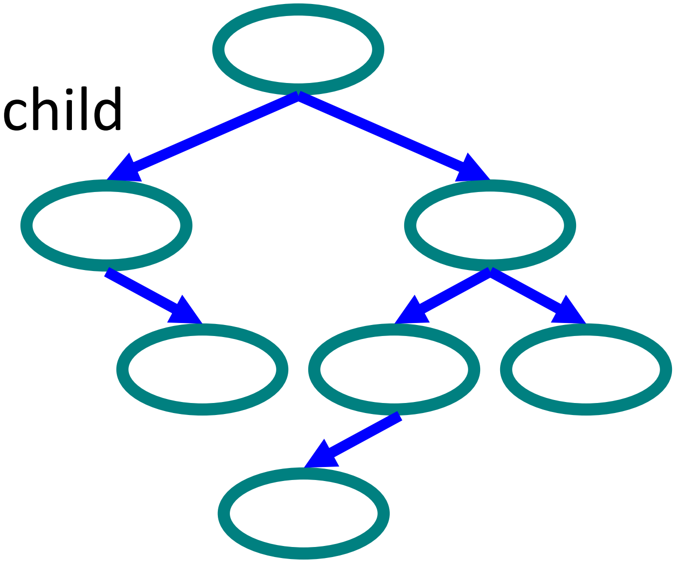
Tree Data Structure

- Terminology
  - Level ⇒ number of ancestors (parent or parent's parent or …) up to the root (distance from the root)
  - Height ⇒ number of levels
    - Some books use height = highest level in the tree



Binary Tree

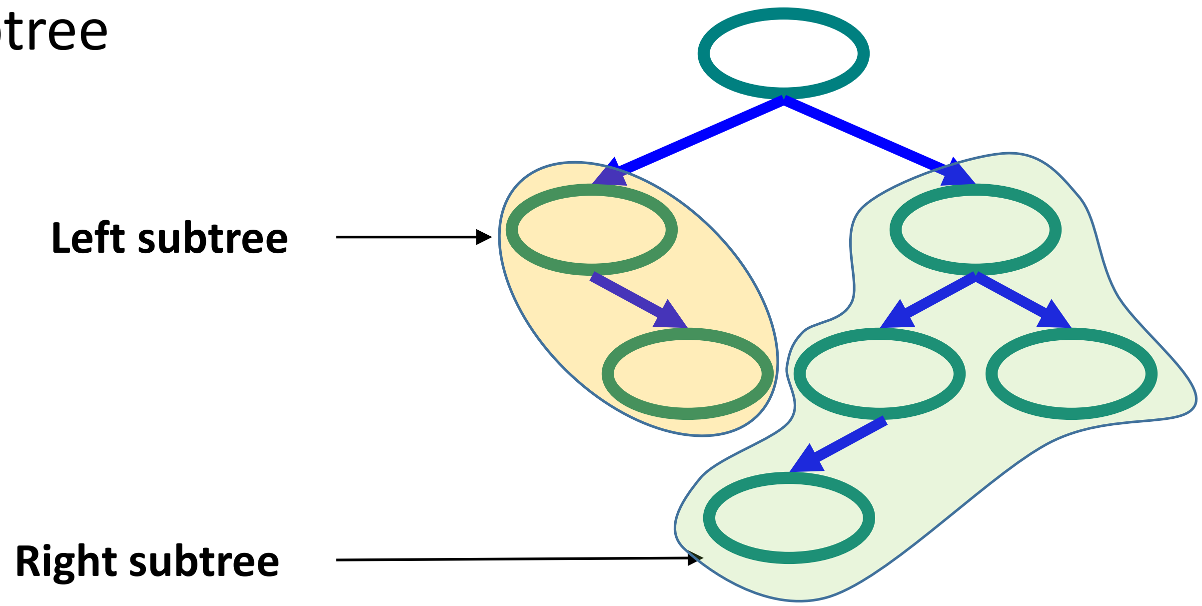
- Characteristics
  - Tree with 0 – 2 children per node
    - No child
    - One left child
    - One right child
    - One left child and one right child



Binary Tree

Binary Tree

- Characteristics
  - Every node is the parent of two smaller trees (subtree)
    - Left subtree
    - Right subtree



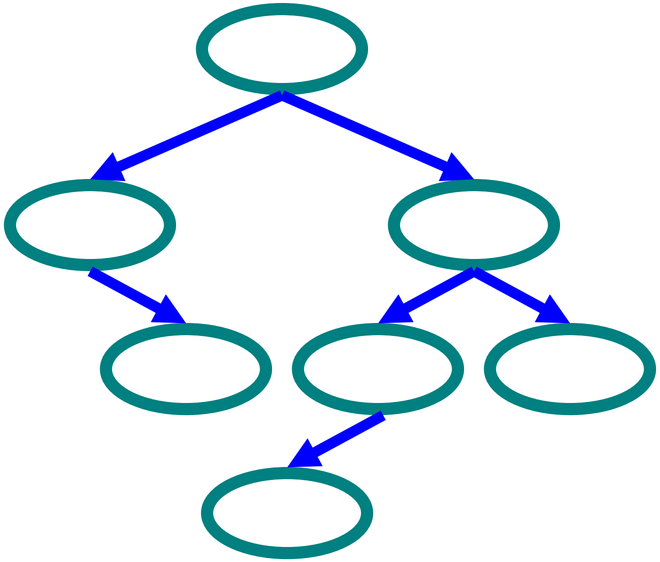
Binary Tree

Binary Tree

- Full tree
  - A binary tree is full if each node is either a leaf or possesses exactly two child nodes.
- Complete tree
  - A binary tree is complete if all levels (except possibly the last) are completely full, and the last level has all its nodes to the left side.

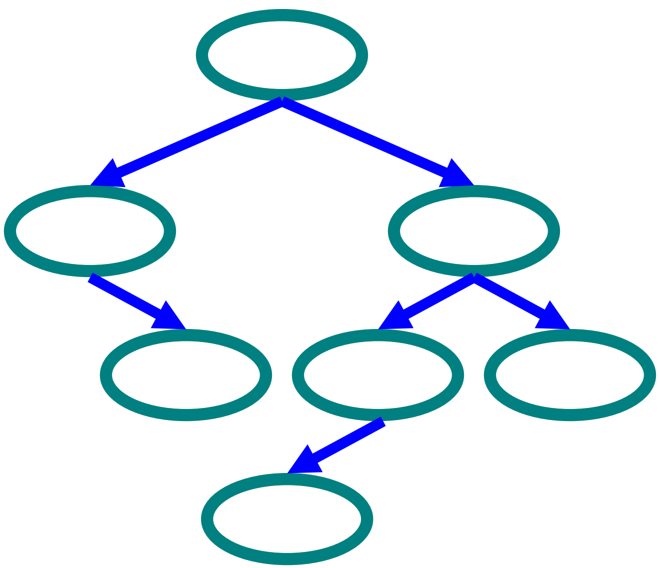
Binary Tree

- Full tree
  - A binary tree is full if each node is either a leaf or possesses exactly two child nodes.
- Complete tree
  - A binary tree is complete if all levels (except possibly the last) are completely full, and the last level has all its nodes to the left side.



Binary Tree

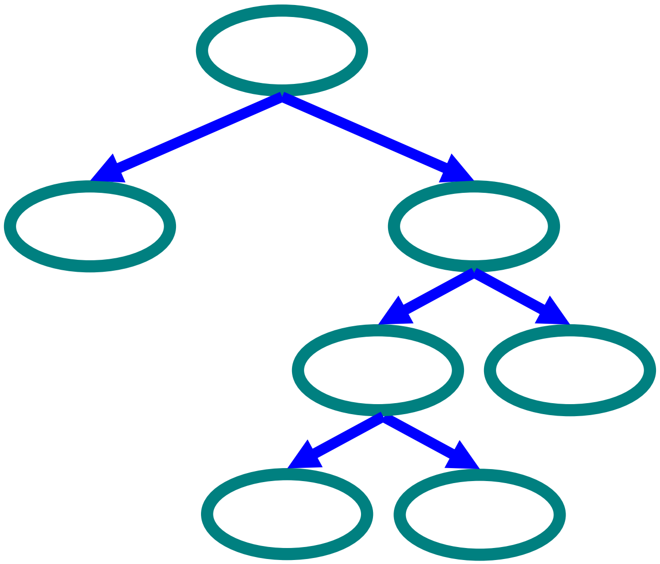
- Full tree
  - A binary tree is full if each node is either a leaf or possesses exactly two child nodes.
- Complete tree
  - A binary tree is complete if all levels (except possibly the last) are completely full, and the last level has all its nodes to the left side.



Neither full nor complete

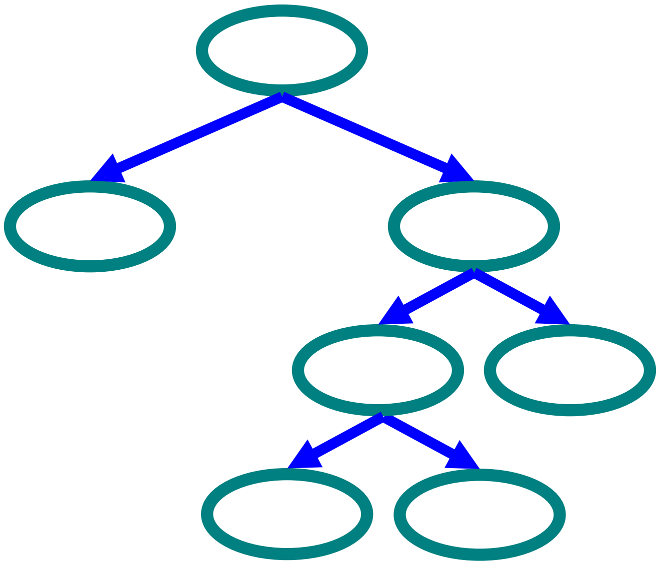
Binary Tree

- Full tree
  - A binary tree is full if each node is either a leaf or possesses exactly two child nodes.
- Complete tree
  - A binary tree is complete if all levels (except possibly the last) are completely full, and the last level has all its nodes to the left side.



Binary Tree

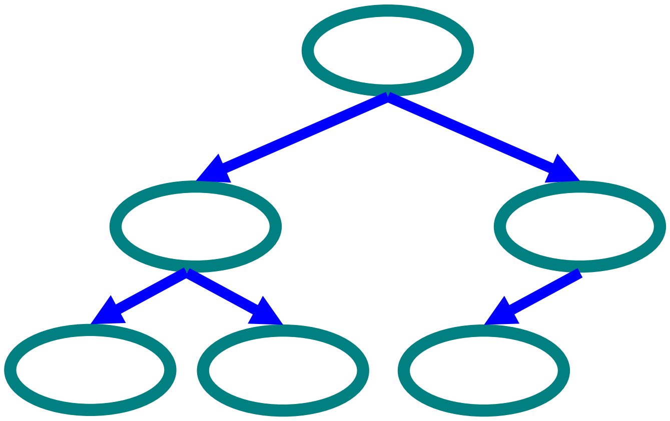
- Full tree
  - A binary tree is full if each node is either a leaf or possesses exactly two child nodes.
- Complete tree
  - A binary tree is complete if all levels (except possibly the last) are completely full, and the last level has all its nodes to the left side.



Full but not complete

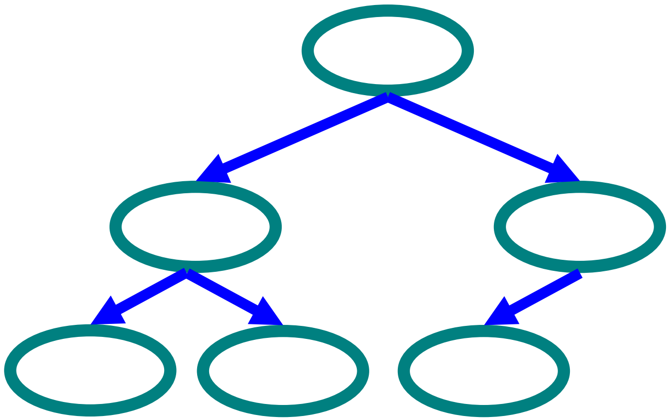
Binary Tree

- Full tree
  - A binary tree is full if each node is either a leaf or possesses exactly two child nodes.
- Complete tree
  - A binary tree is complete if all levels (except possibly the last) are completely full, and the last level has all its nodes to the left side.



Binary Tree

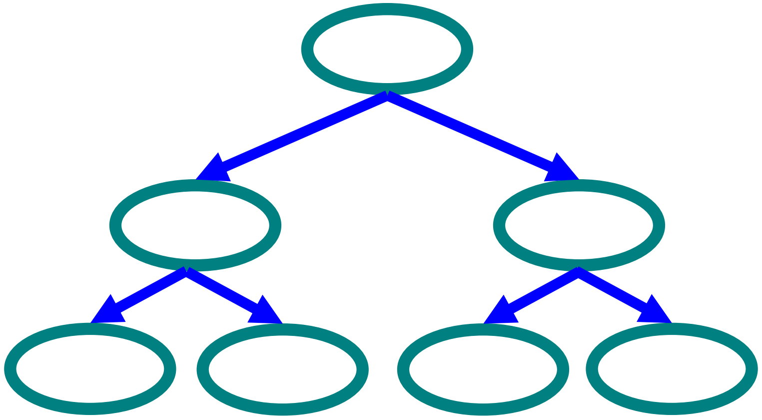
- Full tree
  - A binary tree is full if each node is either a leaf or possesses exactly two child nodes.
- Complete tree
  - A binary tree is complete if all levels (except possibly the last) are completely full, and the last level has all its nodes to the left side.



Not full but complete

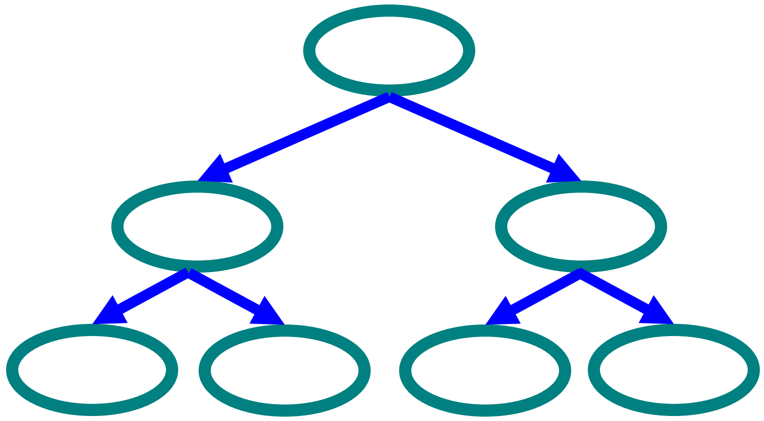
Binary Tree

- Full tree
  - A binary tree is full if each node is either a leaf or possesses exactly two child nodes.
- Complete tree
  - A binary tree is complete if all levels (except possibly the last) are completely full, and the last level has all its nodes to the left side.



Binary Tree

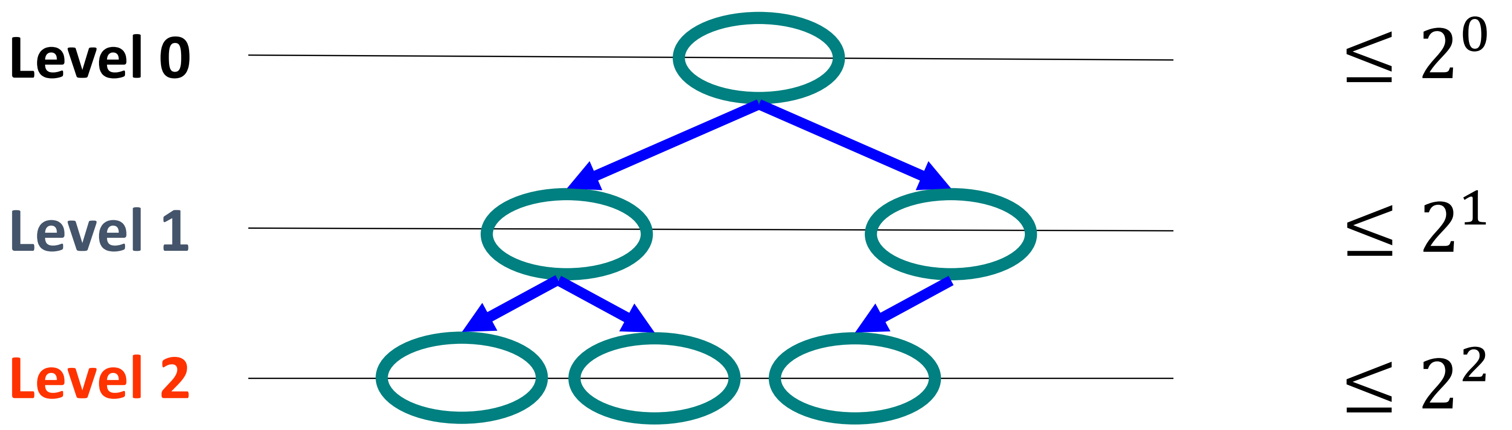
- Full tree
  - A binary tree is full if each node is either a leaf or possesses exactly two child nodes.
- Complete tree
  - A binary tree is complete if all levels (except possibly the last) are completely full, and the last level has all its nodes to the left side.



Full and complete

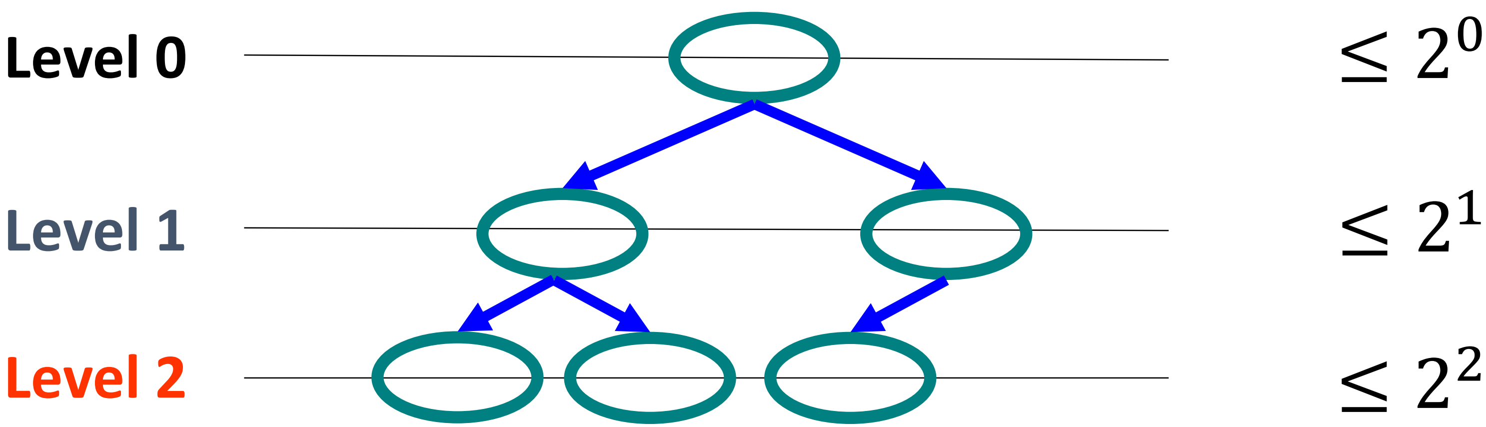
Binary Tree

- How many nodes can there be in one level of a binary tree?
  - **Maximum number** of nodes in level i = $2^i, i = 0, 1, 2 \ldots$



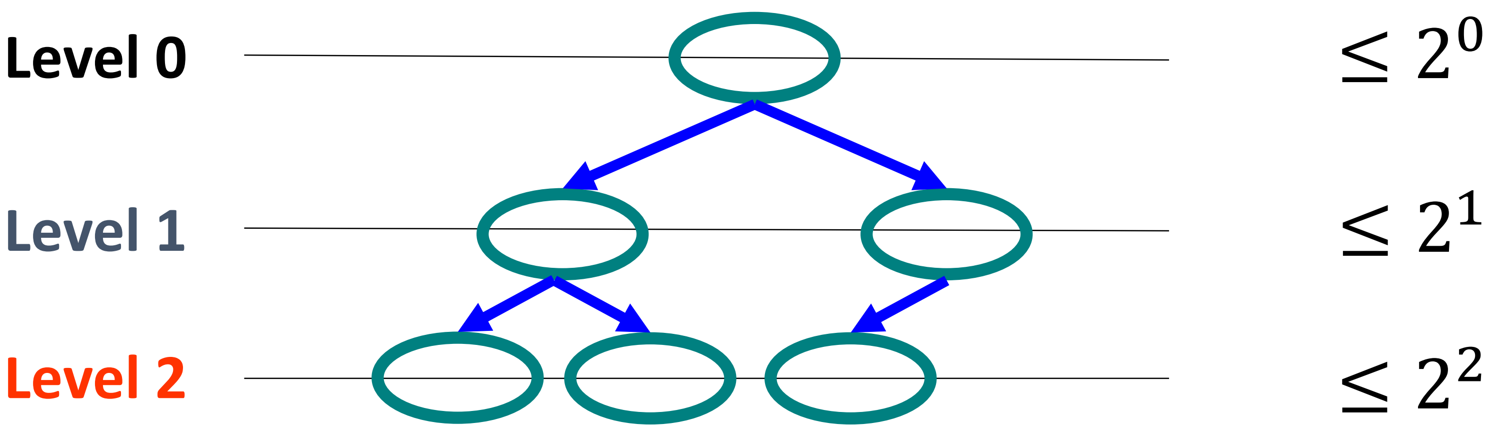
Binary Tree

- How many nodes can there be in in a binary tree with height h?
  - We have the levels $0, 1, 2\ \ldots (h-1)$
  - **Maximum number of nodes** $N = 2^0 + 2^1 + 2^2 + \cdots + 2^{h-1} = \frac{2^{(h-1)+1}-1}{2-1} = 2^h - 1$



Binary Tree

- What is the height of a binary tree with N nodes?
  - We have seen that $N = 2^h - 1$
  - So, $h = \log_2(N + 1)$
  - **Minimum height** is $\log_2(N + 1)$ or simply $\log N$
  - **Maximum height** is N

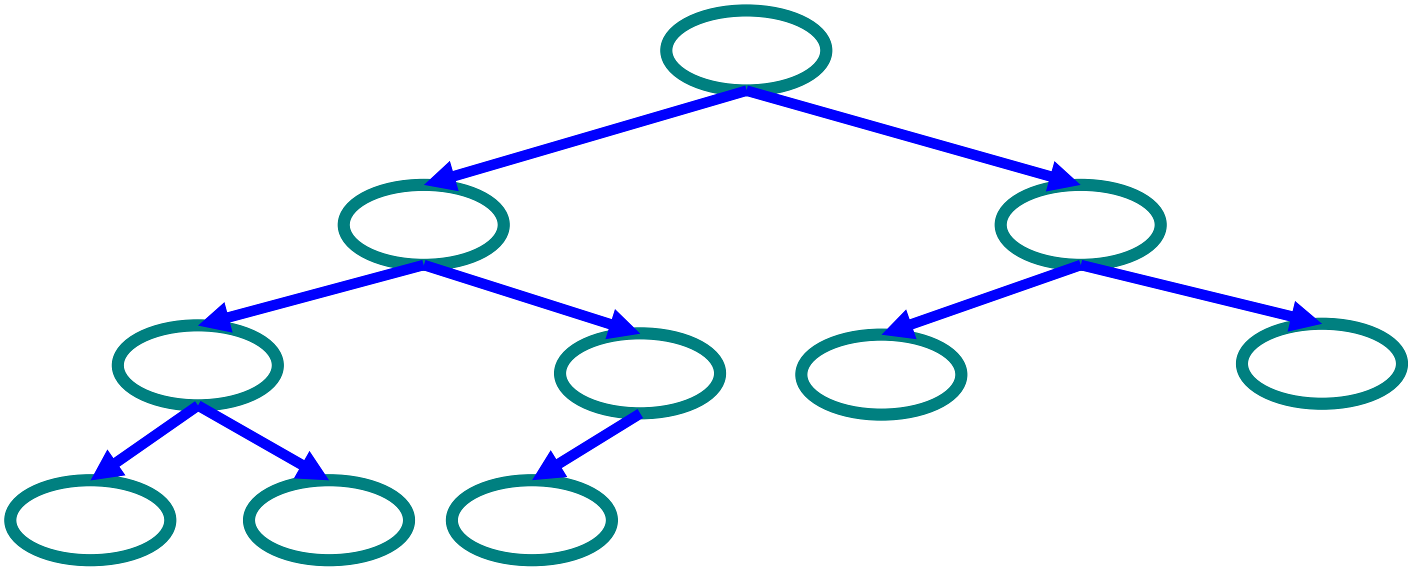


Binary Tree

- What is the height of a binary tree with N nodes?
  - For example, if we have 10 nodes
    - Minimum height = $\log 10 = 3.31{\sim}4$
    - Maximum height = 10

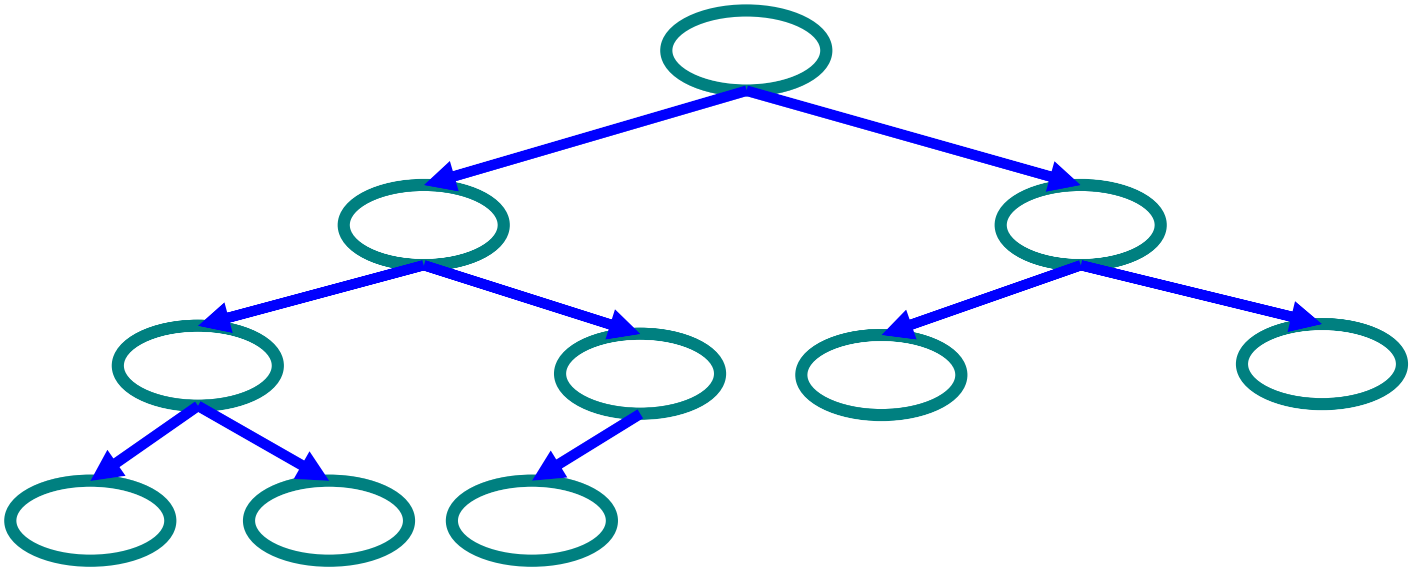
Binary Tree

- What is the height of a binary tree with N nodes?
  - For example, if we have 10 nodes
    - **Minimum height =** $\mathbf{\log 10 = 3.31 \sim 4}$
    - Maximum height = 10



Binary Tree

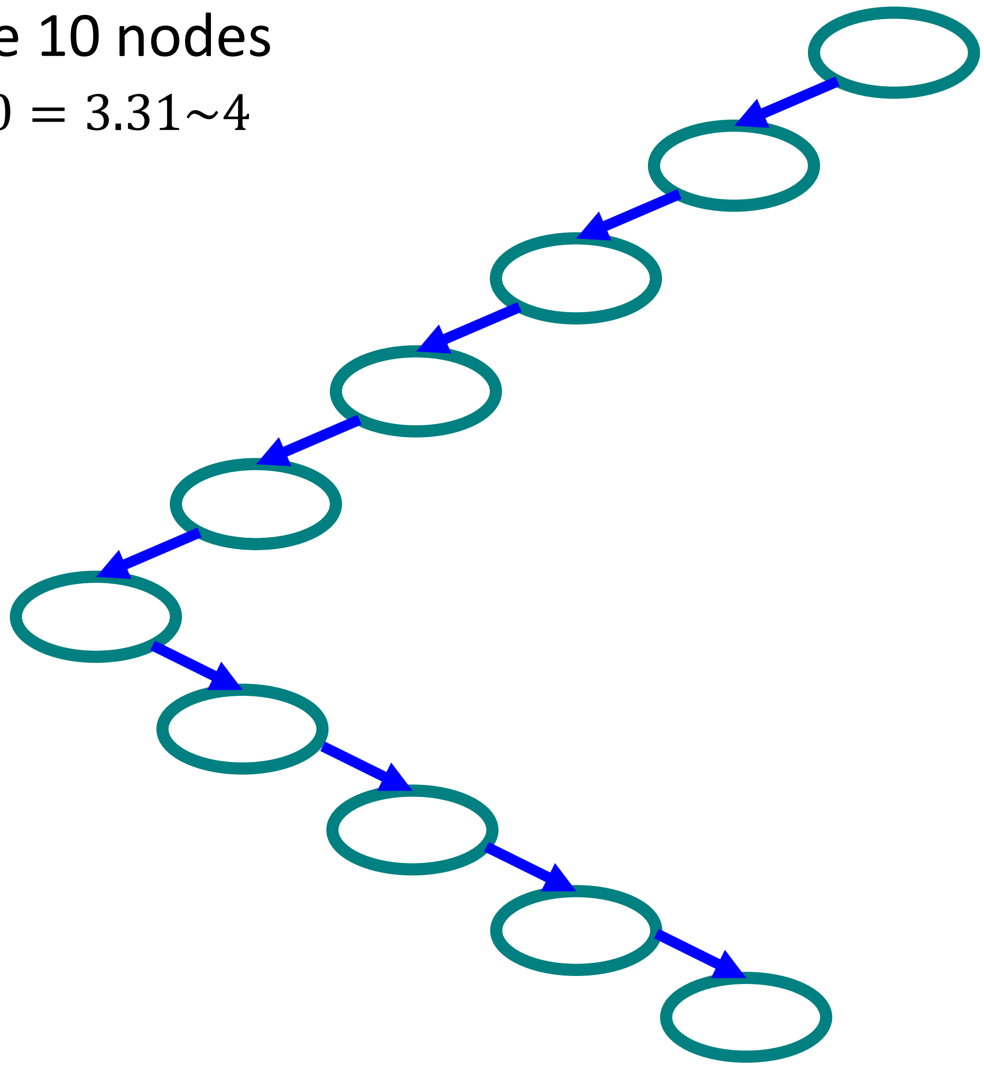
- What is the height of a binary tree with N nodes?
  - For example, if we have 10 nodes
    - **Minimum height =** $\mathbf{\log 10 = 3.31 \sim 4}$
    - Maximum height = 10



When tree is complete
(best case)

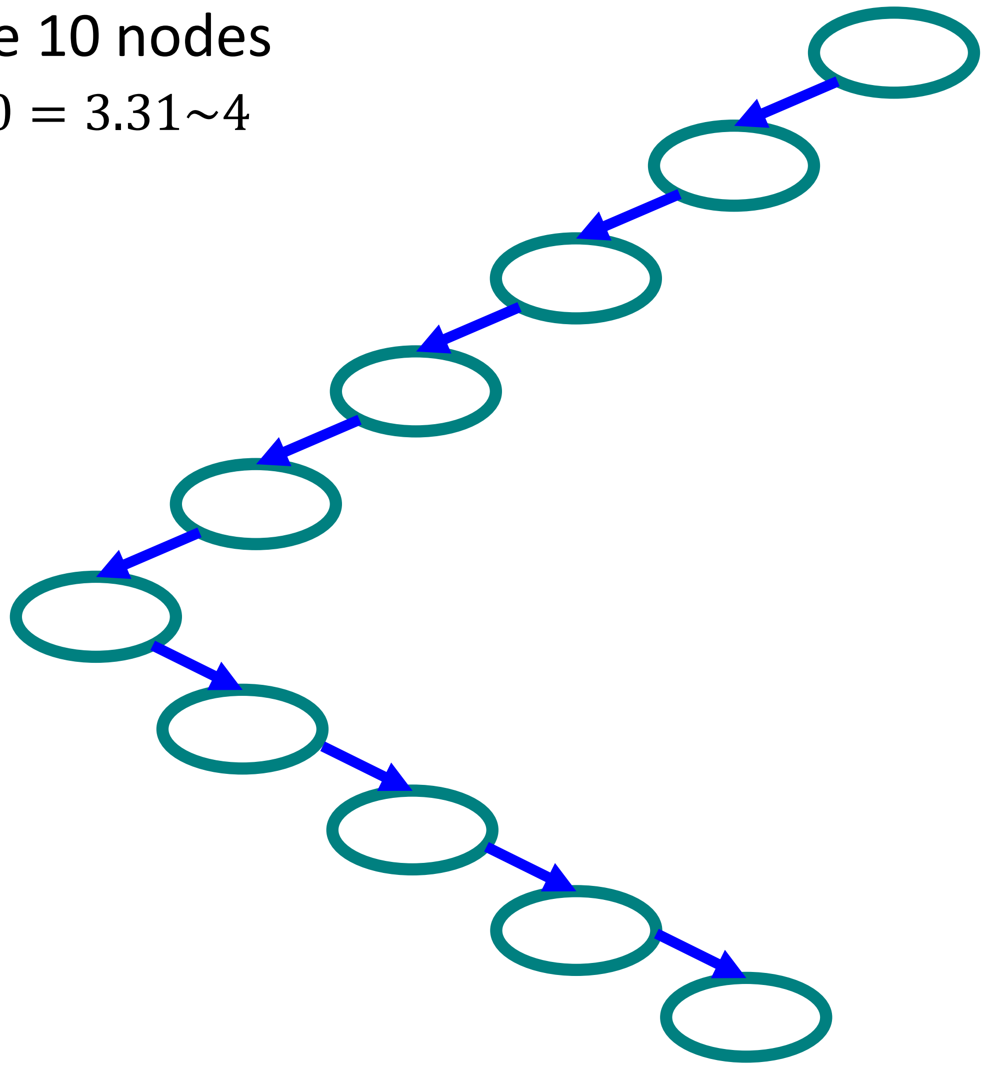
Binary Tree

- What is the height of a binary tree with N nodes?
  - For example, if we have 10 nodes
    - Minimum height = $\log 10 = 3.31 \sim 4$
    - **Maximum height = $\mathbf{10}$**



Binary Tree

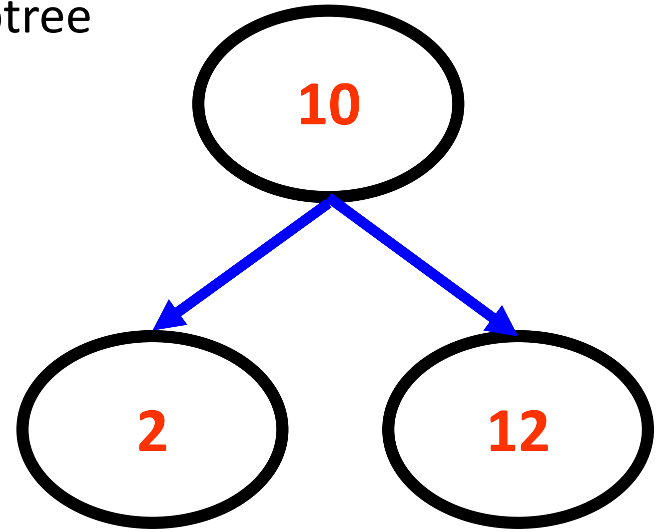
- What is the height of a binary tree with N nodes?
  - For example, if we have 10 nodes
    - Minimum height = $\log 10 = 3.31{\sim}4$
    - **Maximum height =** $\mathbf{10}$



When each node has one child (worst case)

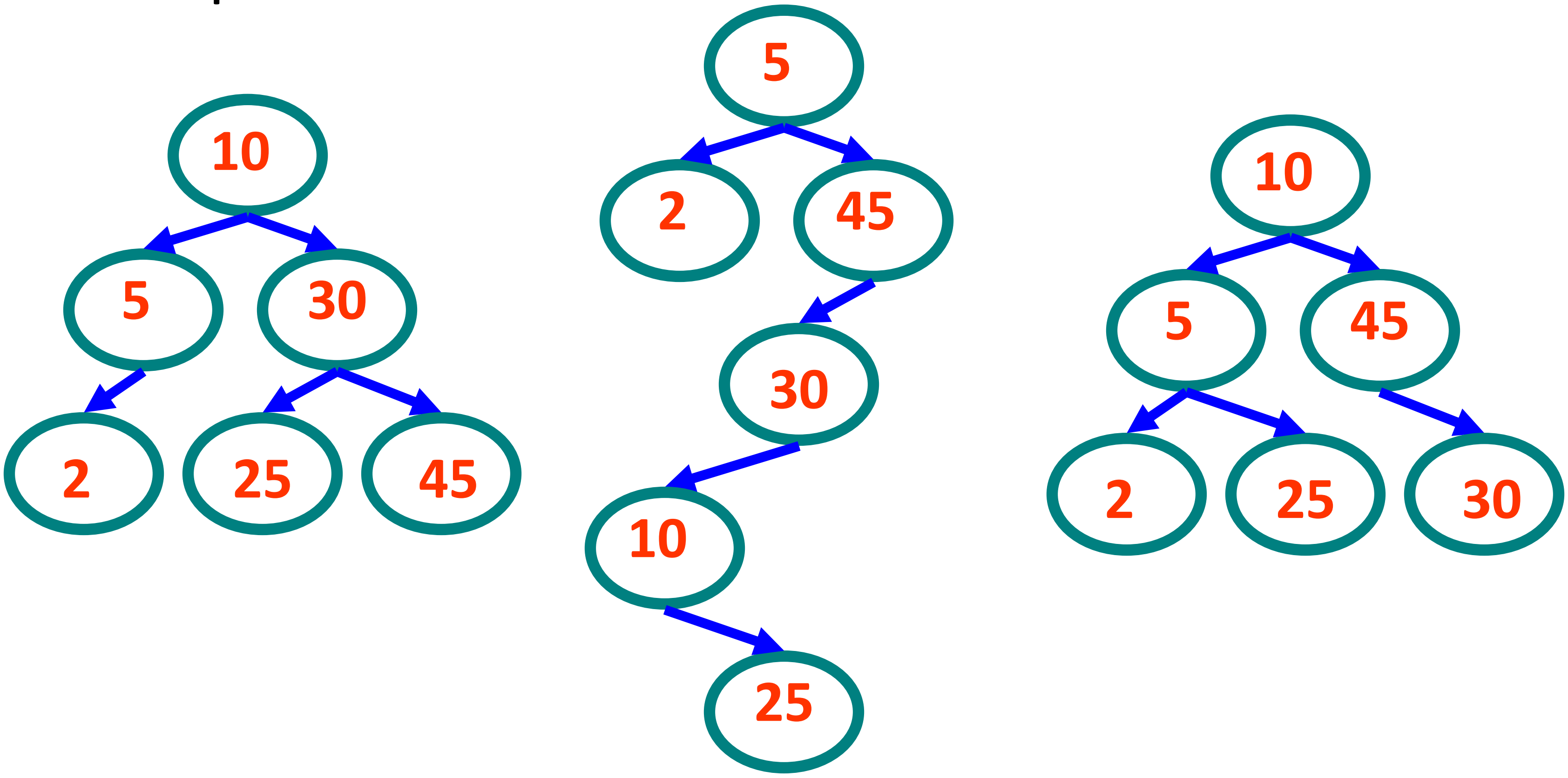
Binary Search Trees

- Key property
  - Value at any node
    - Larger than the value of left child
      - Which means that it is larger than all values in left subtree
    - Smaller or equal to the value of right child
      - Which means that it is smaller than all values in right subtree



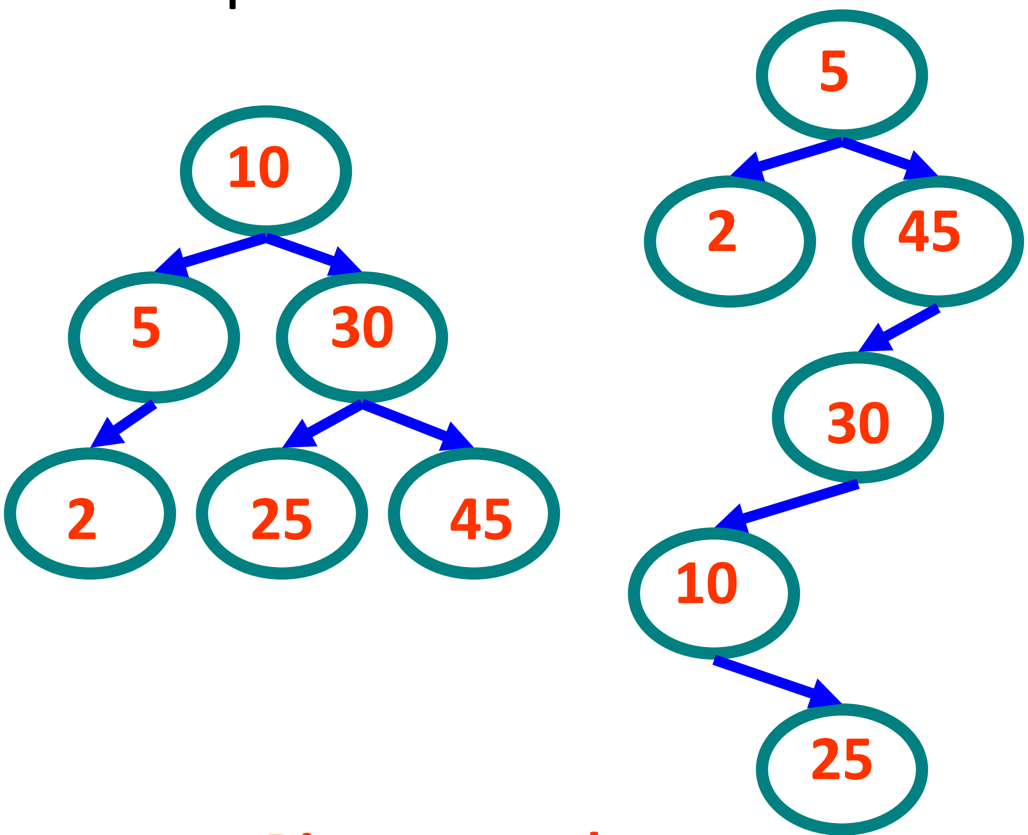
Binary Search Trees

- Examples

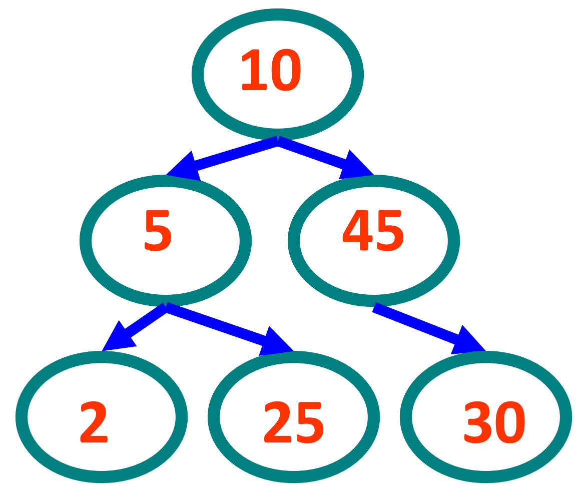


Binary Search Trees

- Examples



Binary search trees



Not a binary search tree

Binary Search Tree Specification

Structure:	The placement of each element in the binary tree must satisfy the binary search property: The value of the key of an element is greater than the value of the key of any element in its left subtree, and less than the value of the key of any element in its right subtree.
Operations:	
MakeEmpty	
Function	Initializes tree to empty state.
Postcondition	Tree exists and is empty.
Boolean IsEmpty	
Function	Determines whether tree is empty.
Postcondition	Returns true if tree is empty and false otherwise.
Boolean IsFull	
Function	Determines whether tree is full.
Postcondition	Returns true if tree is full and false otherwise.

Binary Search Tree Specification

int LengthIs	
Function	Determines the number of elements in tree.
Postcondition	Returns the number of elements in tree.
RetrieveItem(ItemType& item, Boolean& found)	
Function	Retrieves item whose key matches item's key (if present).
Precondition	Key member of item is initialized.
Postcondition	If there is an element someItem whose key matches item's key, then found = true and item is a copy of someItem; otherwise, found = false and item is unchanged**. Tree is unchanged**.
InsertItem(ItemType item)	
Function	Adds item to tree.
Precondition	Tree is not full. item is not in tree.
Postcondition	item is in tree. **Binary search property is maintained**.

Binary Search Tree Specification

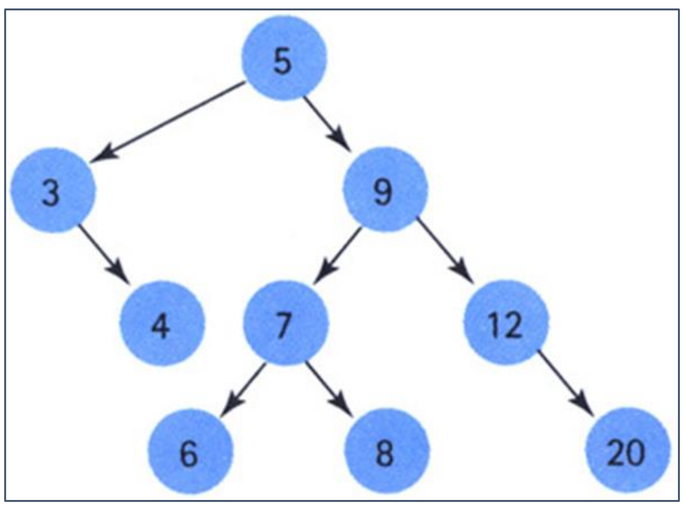
DeleteItem(ItemType item)	
Function	Deletes the element whose key matches item's key.
Precondition	Key member of item is initialized. One and only one element in tree has a key matching item's key.
Postcondition	No element in tree has a key matching item's key.
Print()	
Function	Prints the values in the tree in ascending key order.
Postcondition	Items in the tree are printed in ascending key order.

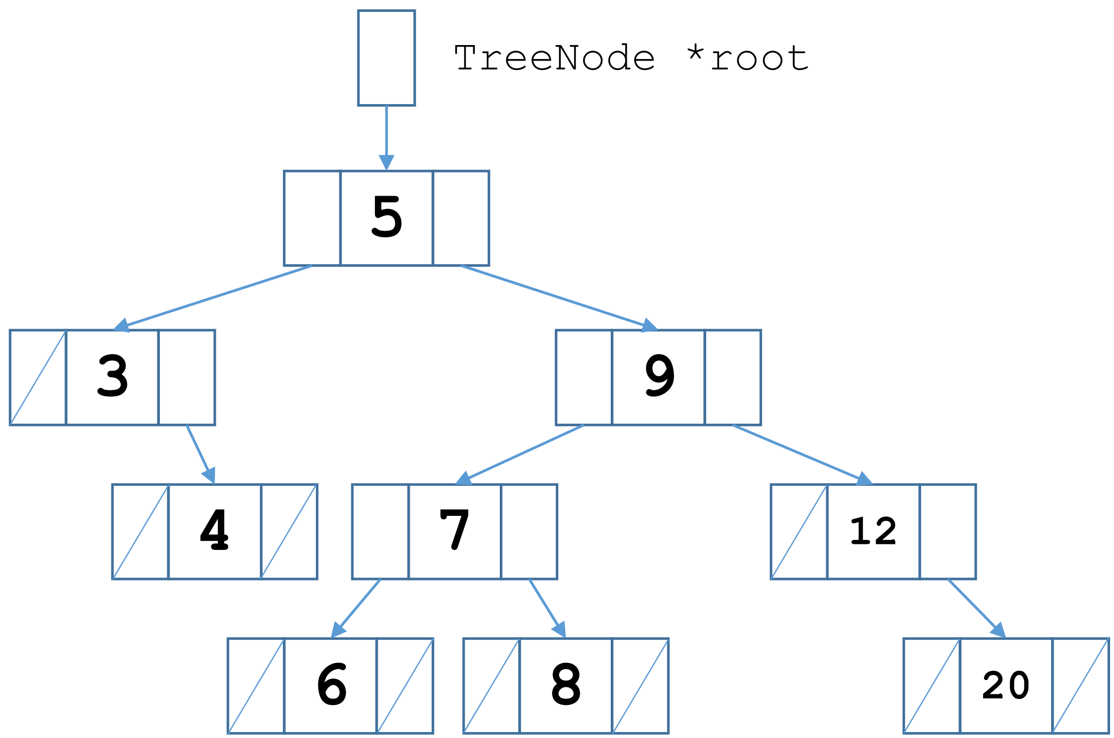
Implementing the Nodes in Binary Search Tree


```
struct TreeNode
{
  ItemType info;
  TreeNode *left;
  TreeNode *right;
};
```

Implementing the Nodes in Binary Search Tree


```
struct TreeNode
{
  ItemType info;
  TreeNode *left;
  TreeNode *right;
};
```





binarysearchtree.h

```
#ifndef BINARYSEARCHTREE_H_INCLUDED
#define BINARYSEARCHTREE_H_INCLUDED

struct TreeNode{ItemType info;TreeNode *left, *right;};

enum OrderType {PRE_ORDER, IN_ORDER, POST_ORDER};

class TreeType
{
public:
  TreeType();
  ~TreeType();
  void MakeEmpty();
  bool IsEmpty();
 // bool IsFull();
  int LengthIs();
  void RetrieveItem(ItemType& item, bool& found);
  void InsertItem(ItemType item);
  void DeleteItem(ItemType item);
  void ResetTree(OrderType order);
  void GetNextItem(ItemType& item, OrderType order, bool& finished);
  void Print();
private:
  TreeNode* root;
};
#endif // BINARYSEARCHTREE_H_INCLUDED
```

binarysearchtree.cpp

```
#include "binarysearchtree.h"

TreeType::TreeType()
{
  root = NULL;
}

bool TreeType::IsEmpty()
{
  return root == NULL;
}
```

```
bool TreeType::IsFull()
{
  TreeNode* location;
  try
  {
    location = new TreeNode;
    delete location;
    return false;
  }
  catch(std::bad_alloc exception)
  {
    return true;
  }
}
```

binarysearchtree.cpp

```
#include "binarysearchtree.h"
#include <new>

TreeType::TreeType()
{
  root = NULL;
}
```

O(1)

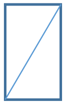
```
bool TreeType::IsEmpty()
{
  return root == NULL;
}
```

O(1)

```
bool TreeType::IsFull()
{
  TreeNode* location;
  try
  {
    location = new TreeNode;
    delete location;
    return false;
  }
  catch(std::bad_alloc exception)
  {
    return true;
  }
}
```

O(1)

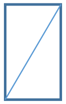
Function `InsertItem`



`TreeNode *root`

Function `InsertItem`

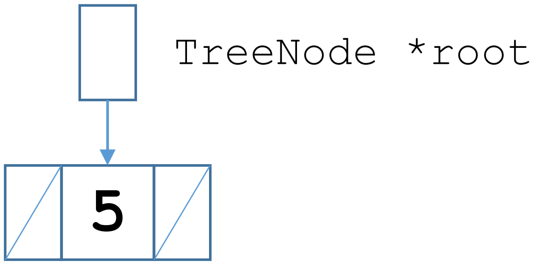
Insert 5



`TreeNode *root`

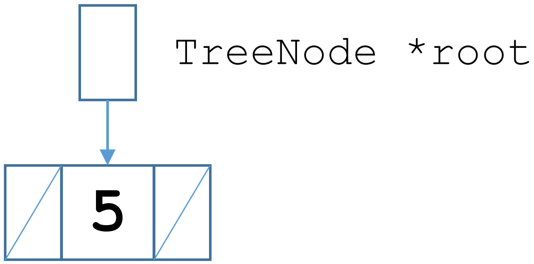
Function `InsertItem`

Insert 5



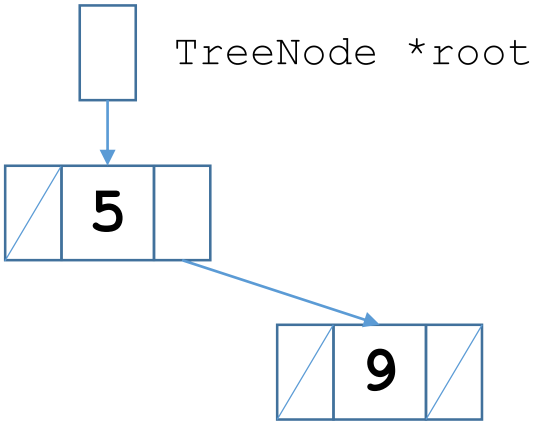
Function `InsertItem`

Insert 9



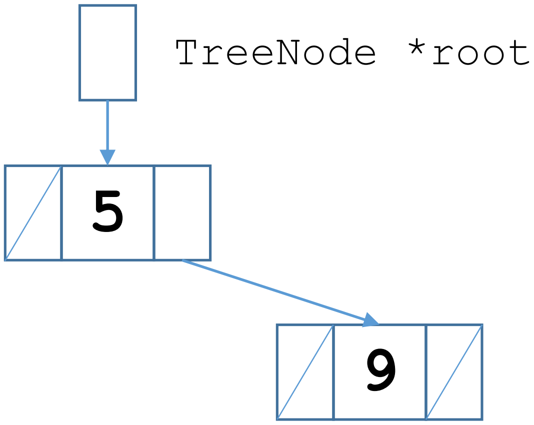
Function `InsertItem`

Insert 9



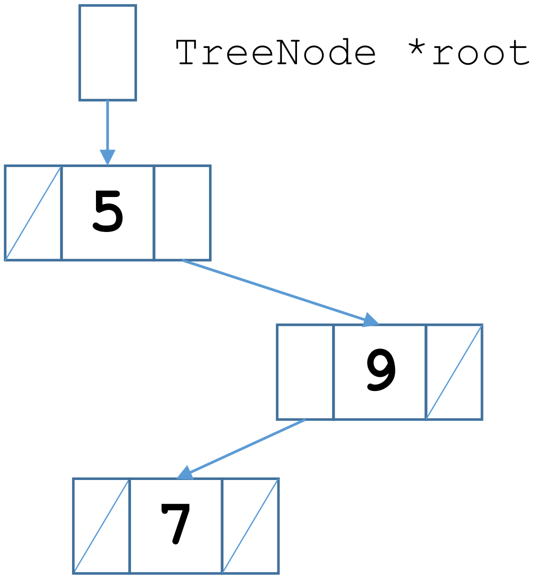
Function `InsertItem`

Insert 7



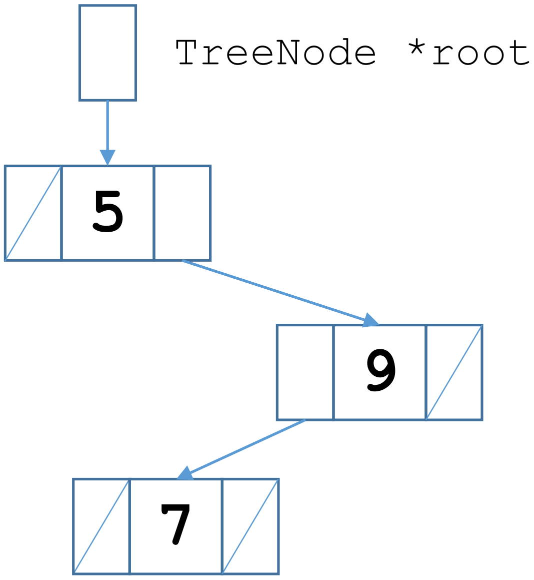
Function `InsertItem`

Insert 7



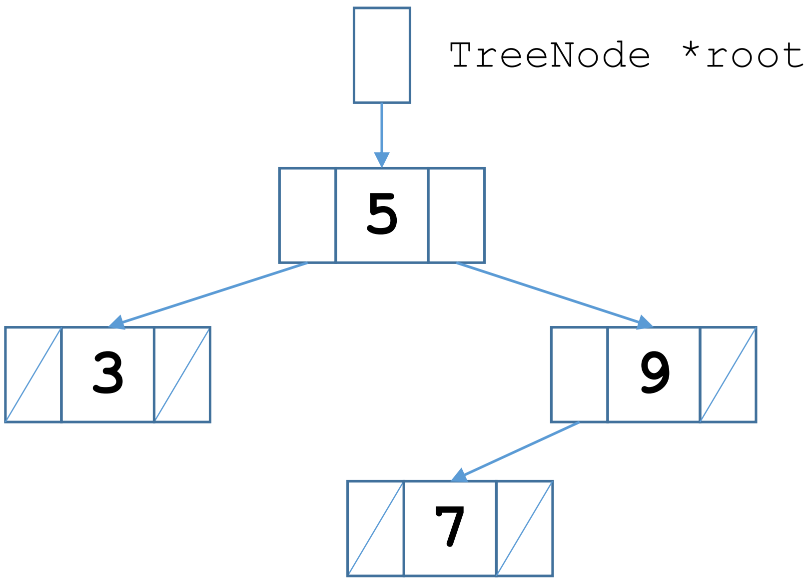
Function `InsertItem`

Insert 3



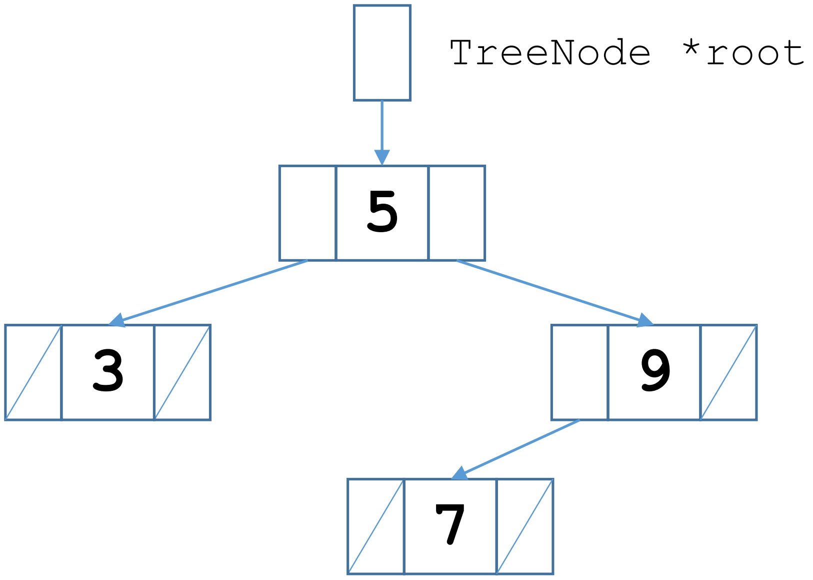
Function `InsertItem`

Insert 3



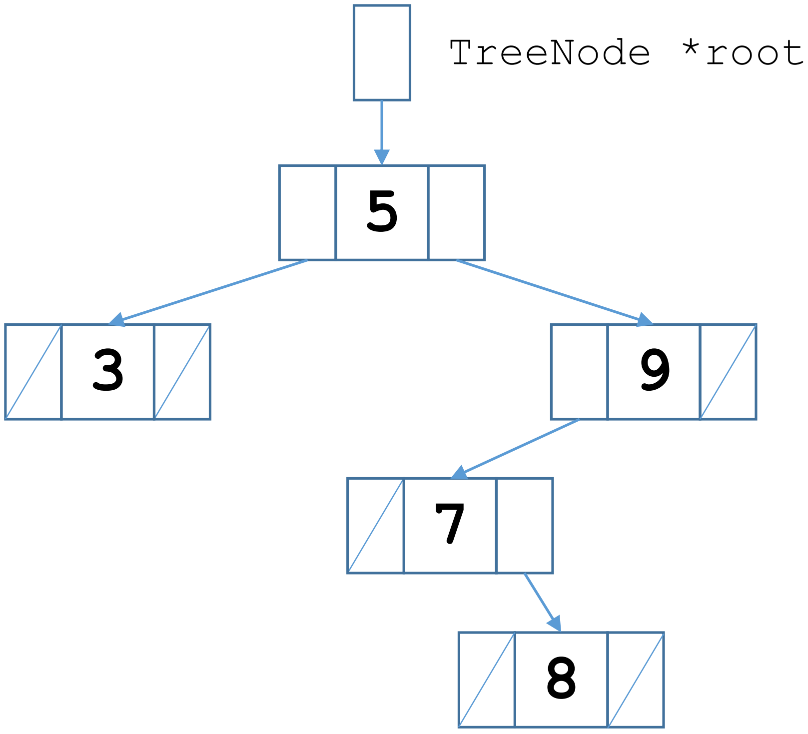
Function `InsertItem`

Insert 8



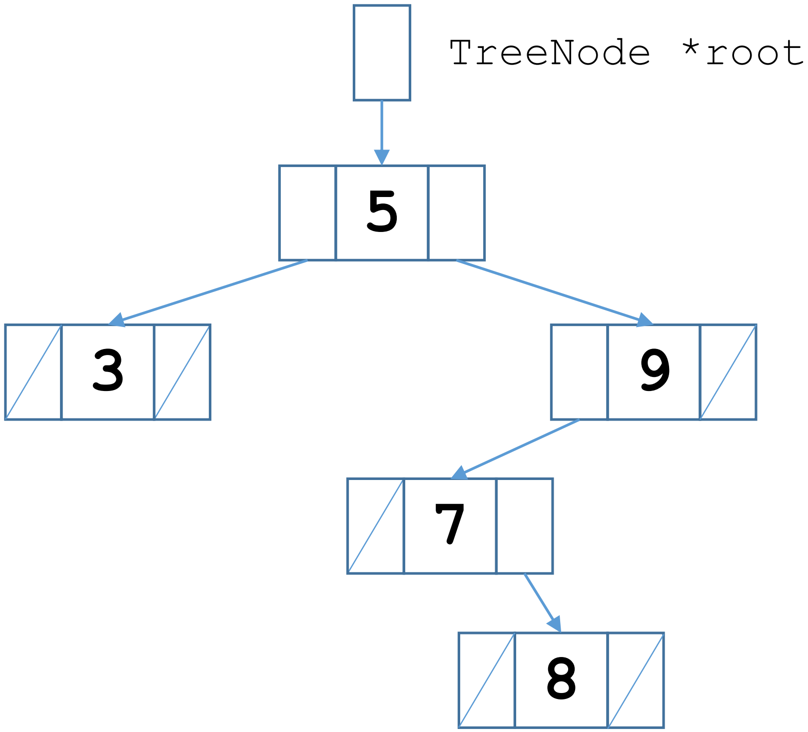
Function `InsertItem`

Insert 8



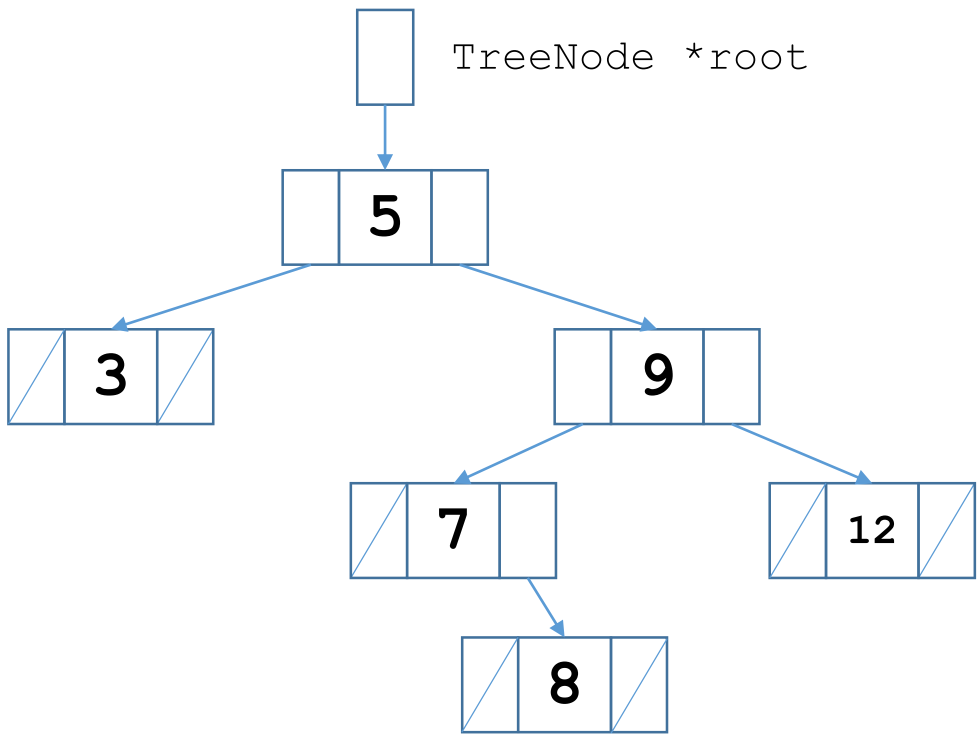
Function `InsertItem`

Insert 12



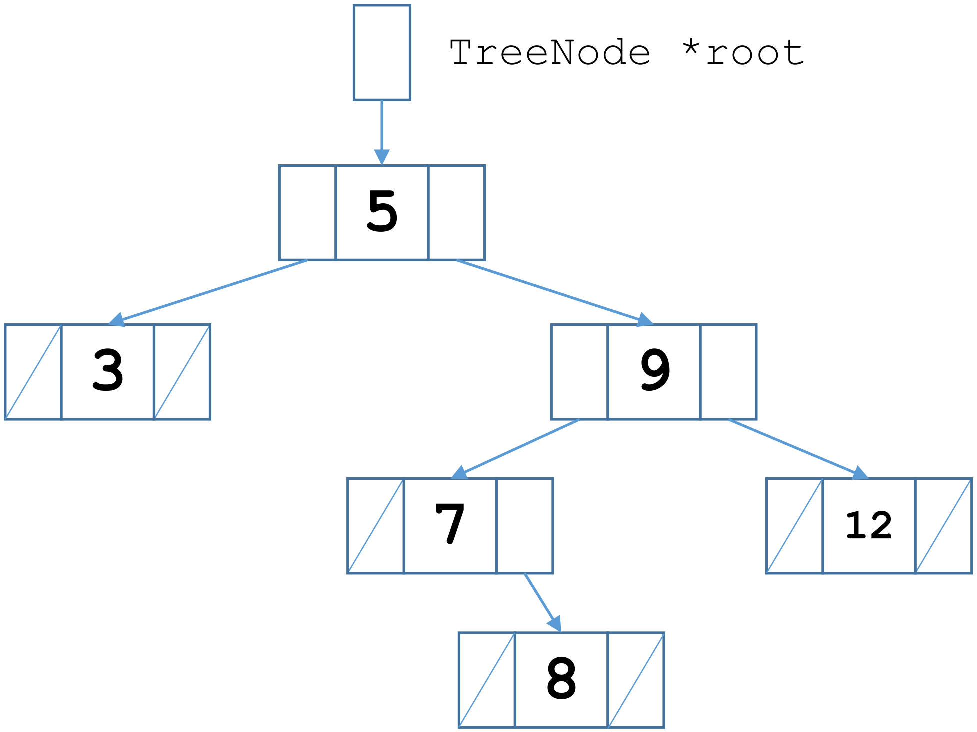
Function `InsertItem`

Insert 12



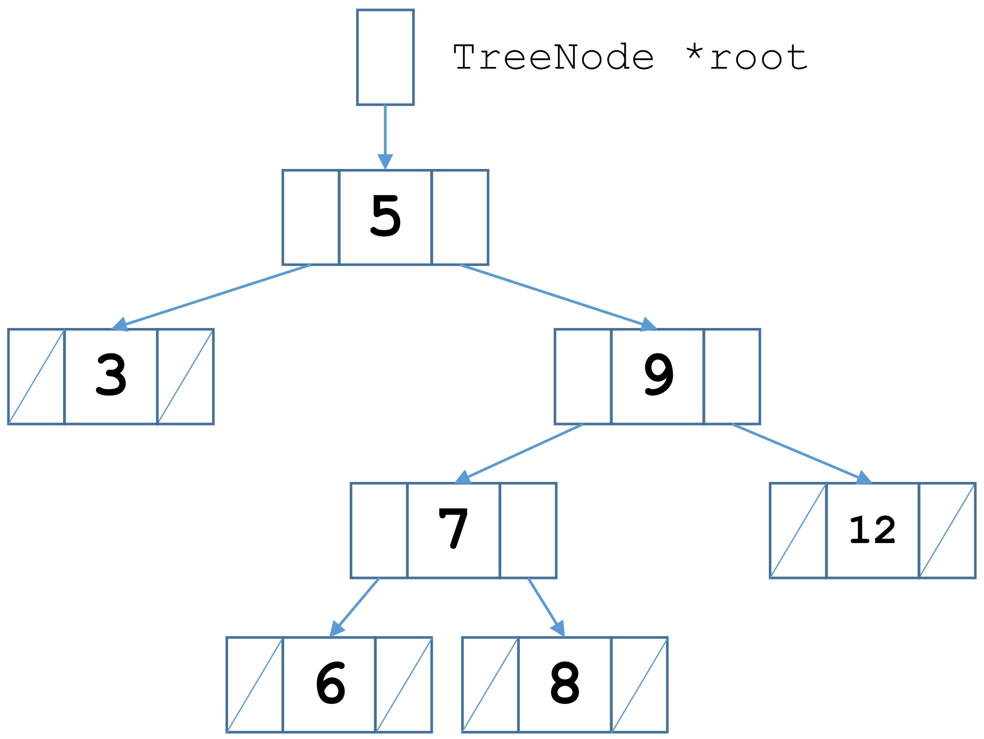
Function `InsertItem`

Insert 6



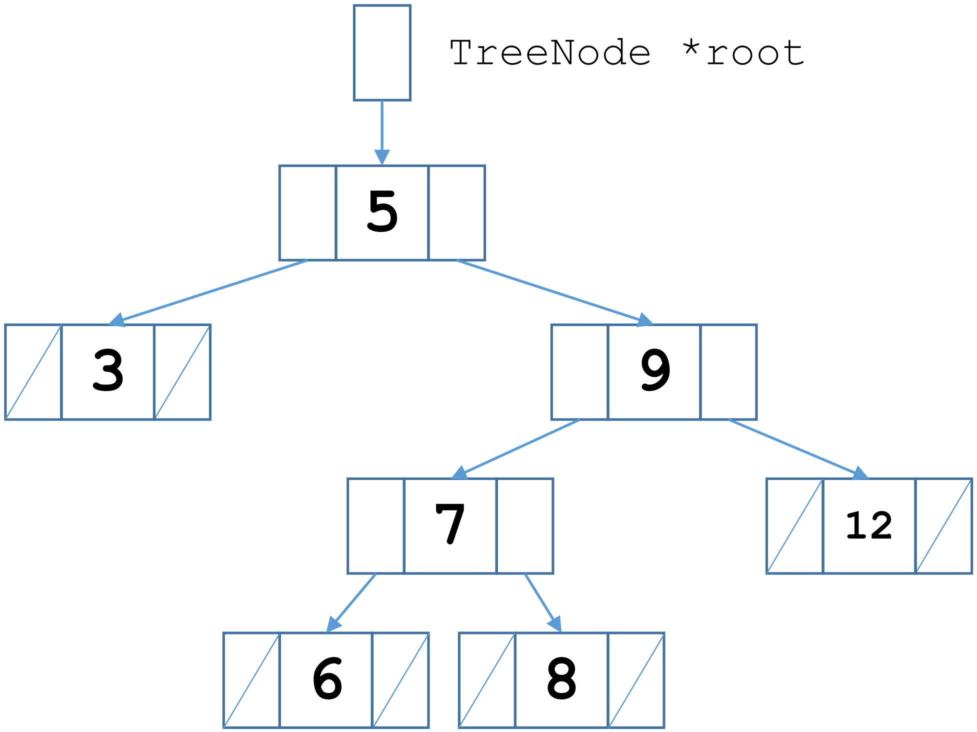
Function `InsertItem`

Insert 6



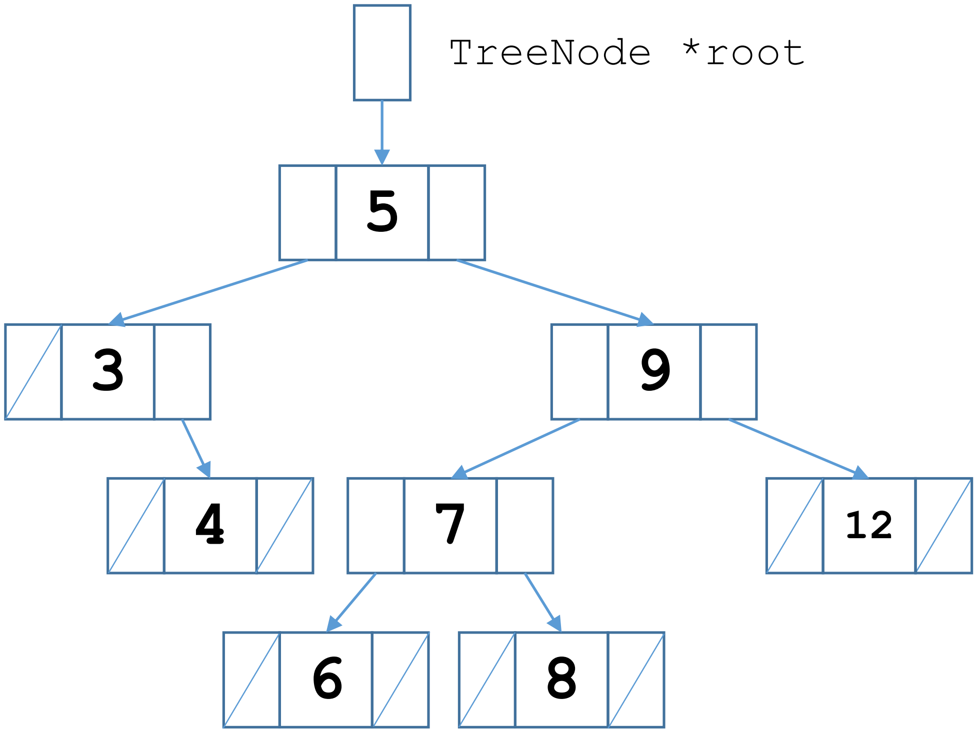
Function `InsertItem`

Insert 4



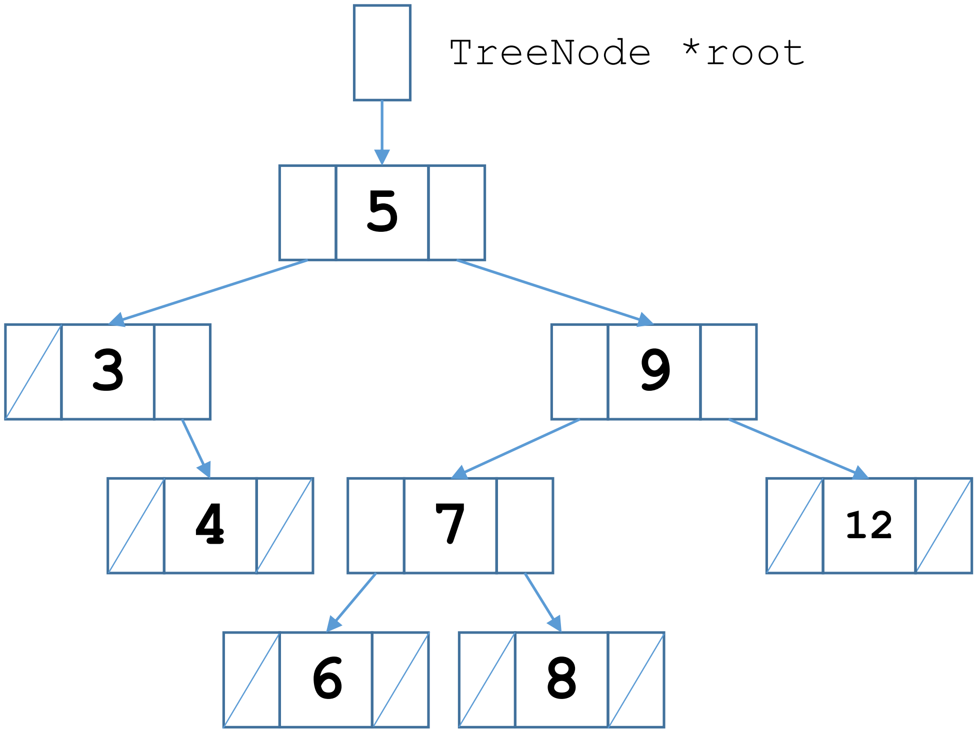
Function `InsertItem`

Insert 4



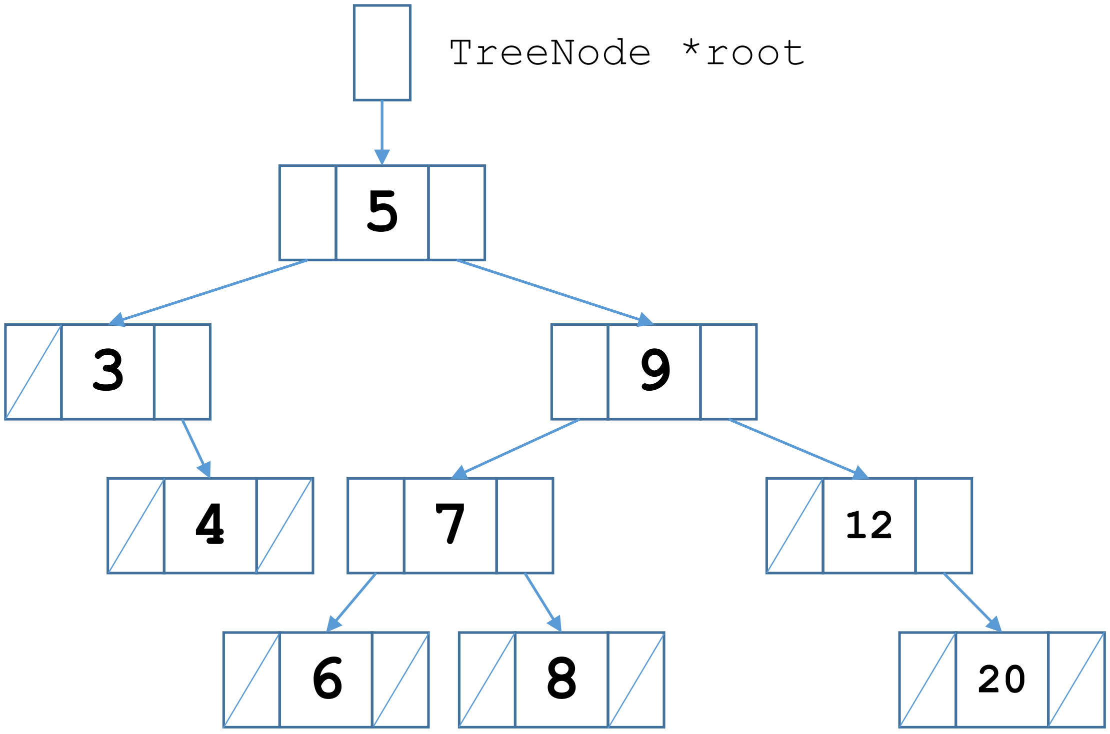
Function `InsertItem`

Insert 20



Function `InsertItem`

Insert 20

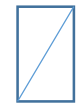


Function InsertItem

```
void Insert(TreeNode &tree, ItemType item)
{
  if (tree == NULL)//Base case:Insertion place found.
  {
    tree = new TreeNode;
    tree->right = NULL;
    tree->left = NULL:
    tree->info = item;
  }
  else if (item < tree->info)
    Insert(tree->left, item);//General case 1:Insert in left subtree.
  else
    Insert(tree->right, item);//General case 2:Insert in right subtree.
}

void TreeType::InsertItem(ItemType item)
{
  Insert(root, item):
}
```

Function `InsertItem`



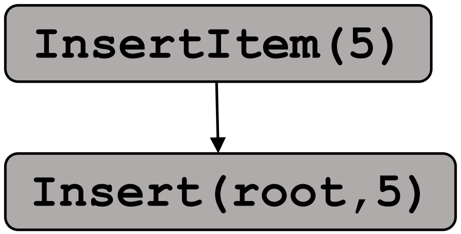
`root`

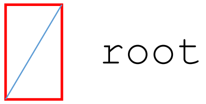
Function `InsertItem`

`InsertItem(5)`

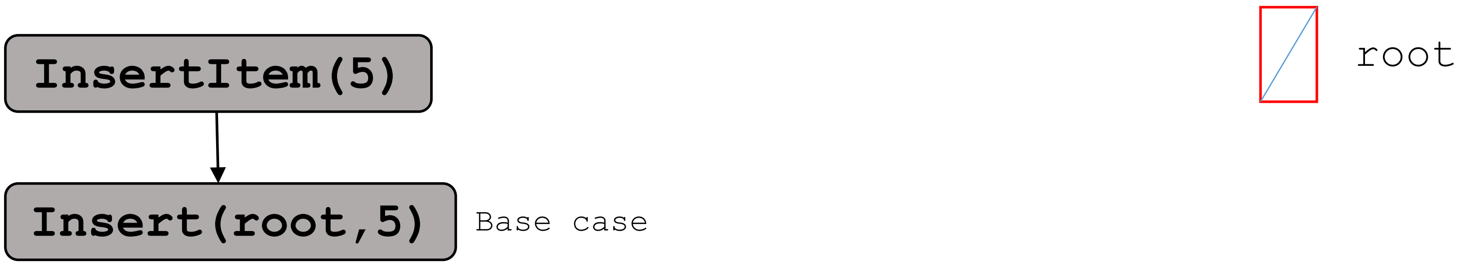
`root`

Function `InsertItem`





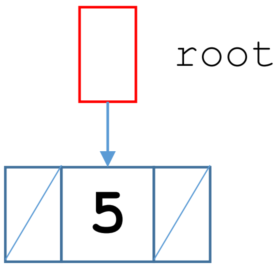
Function `InsertItem`



Function `InsertItem`

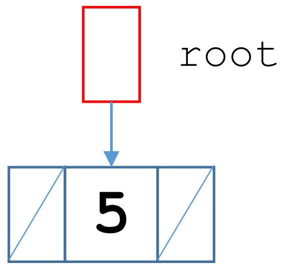
`InsertItem(5)`

`Insert(root,5)` `Base case`

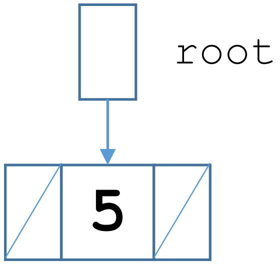


Function `InsertItem`

`InsertItem(5)`

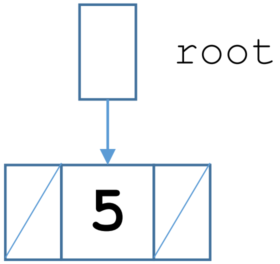


Function `InsertItem`

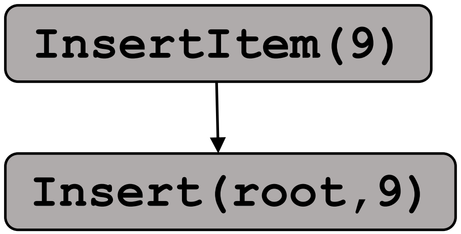


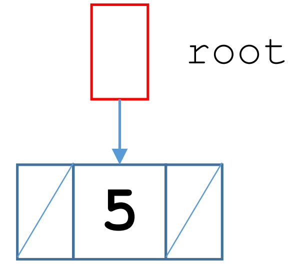
Function `InsertItem`

`InsertItem(9)`



Function `InsertItem`

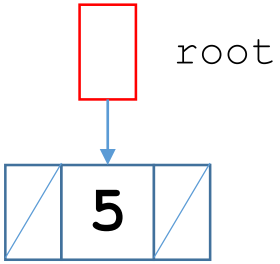




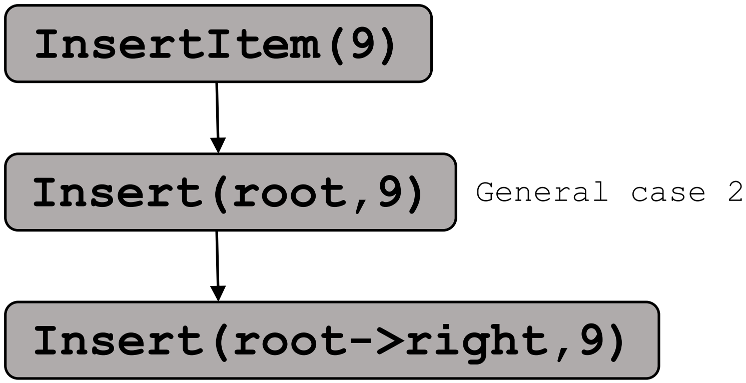
Function `InsertItem`

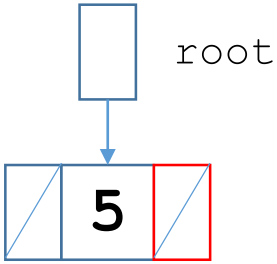
`InsertItem(9)`

`Insert(root,9)` `General case 2`

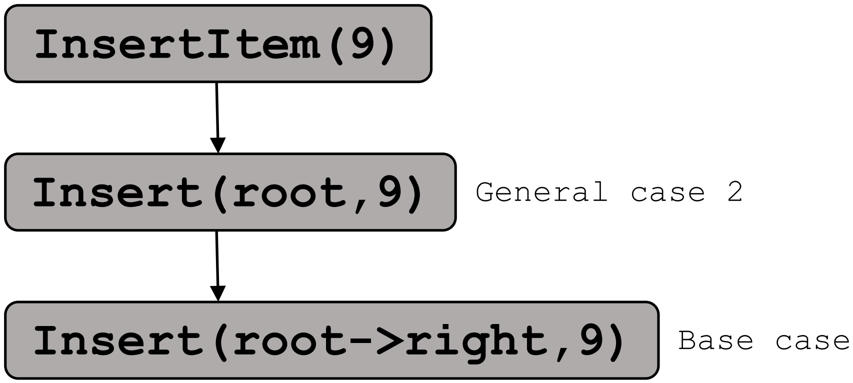


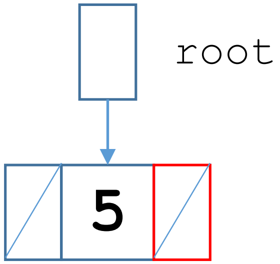
Function `InsertItem`





Function `InsertItem`



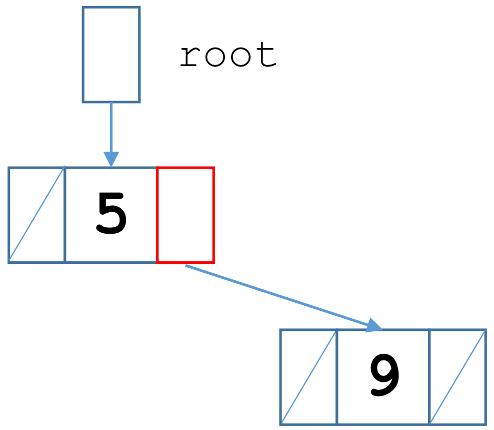


Function `InsertItem`

`InsertItem(9)`

`Insert(root,9)` General case 2

`Insert(root->right,9)` Base case

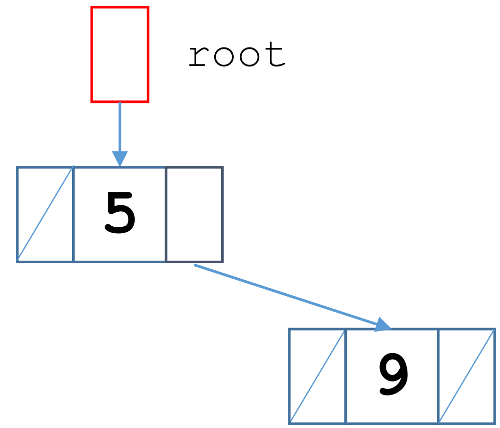


Function `InsertItem`

`InsertItem(9)`

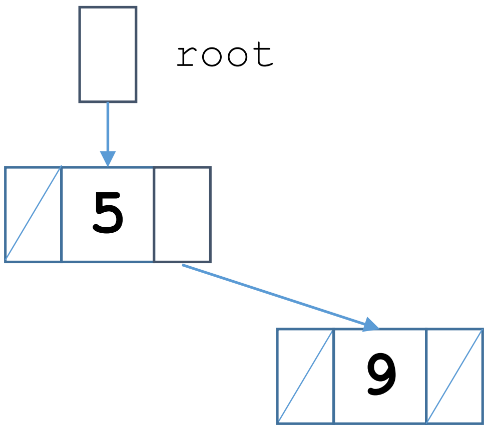
↓

`Insert(root,9)` `General case 2`

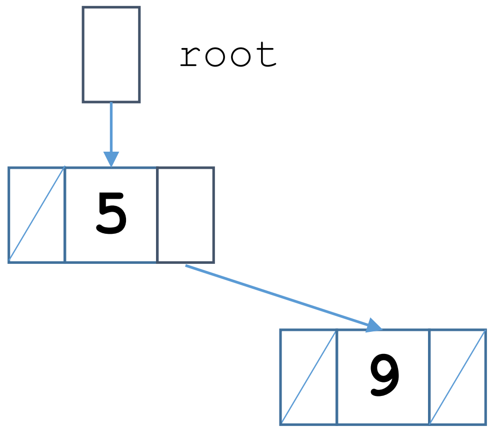


Function `InsertItem`

`InsertItem(9)`

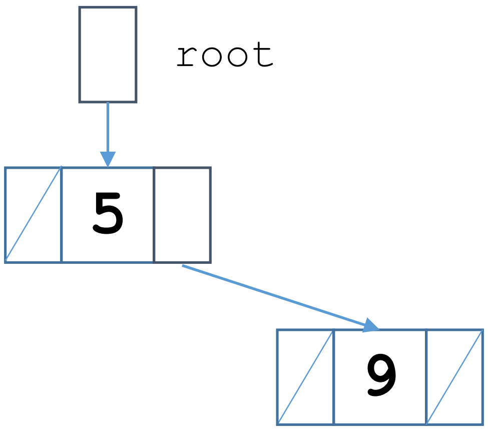


Function `InsertItem`



Function `InsertItem`

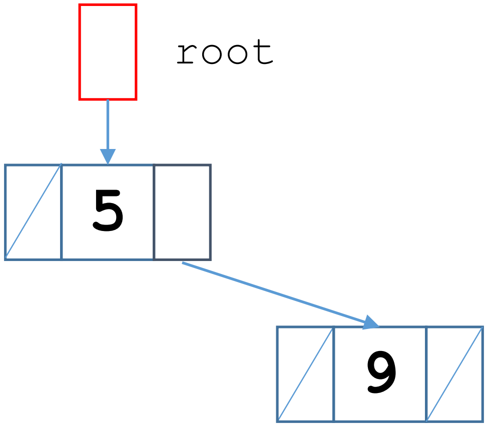
`InsertItem(7)`



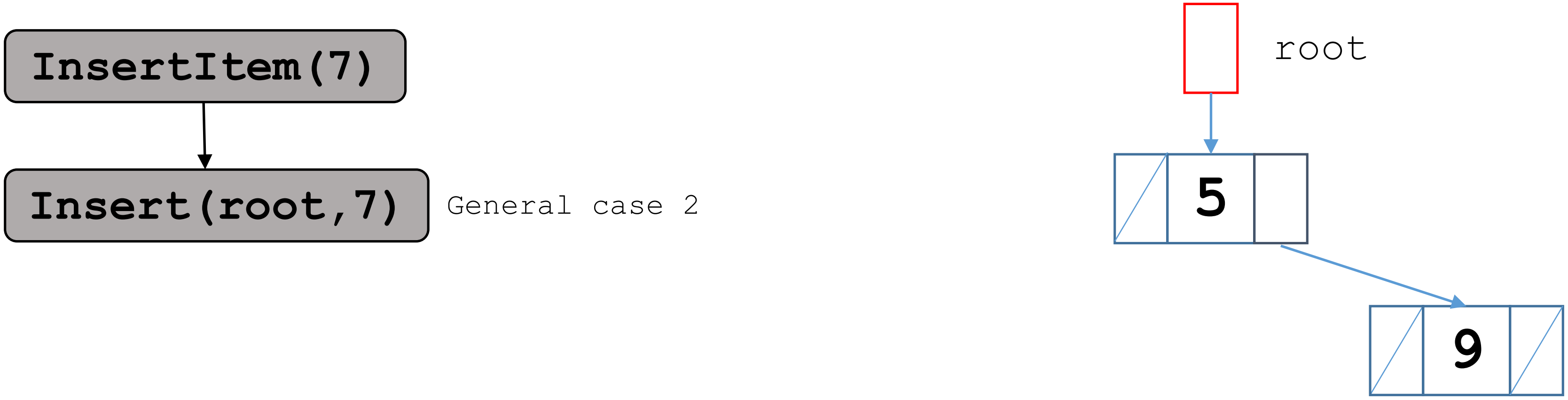
Function `InsertItem`

`InsertItem(7)`

`Insert(root,7)`



Function `InsertItem`

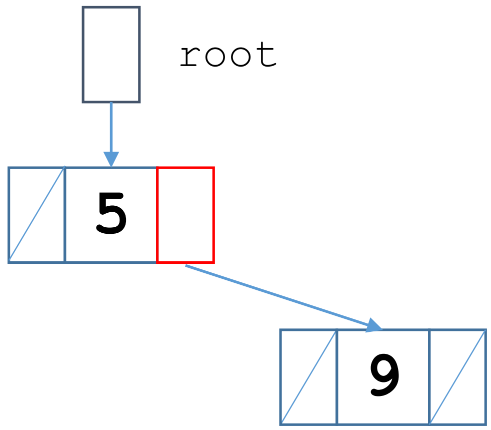


Function `InsertItem`

`InsertItem(7)`

`Insert(root,7)` `General case 2`

`Insert(root->right,7)`

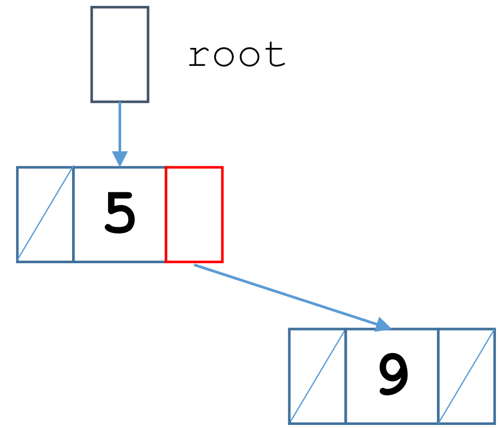


Function `InsertItem`

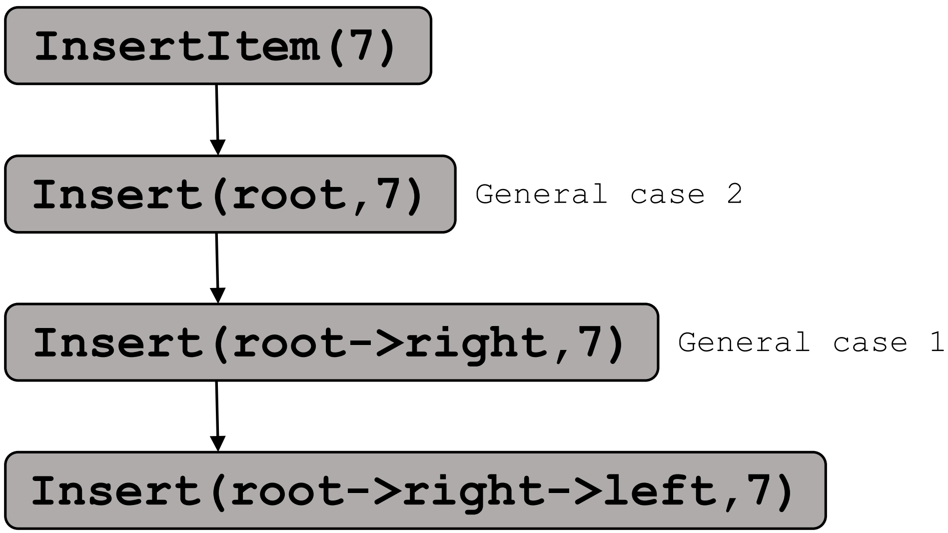
`InsertItem(7)`

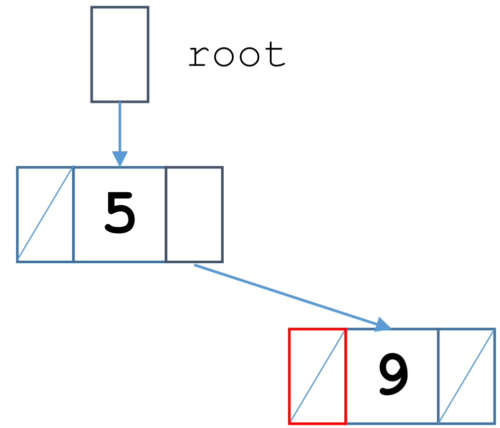
`Insert(root,7)` General case 2

`Insert(root->right,7)` General case 1



Function `InsertItem`





Function `InsertItem`

`InsertItem(7)`

↓

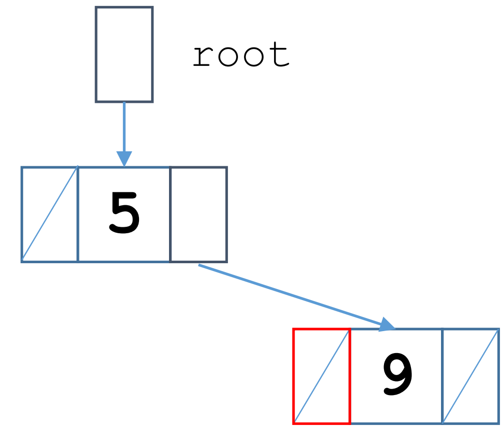
`Insert(root,7)` General case 2

↓

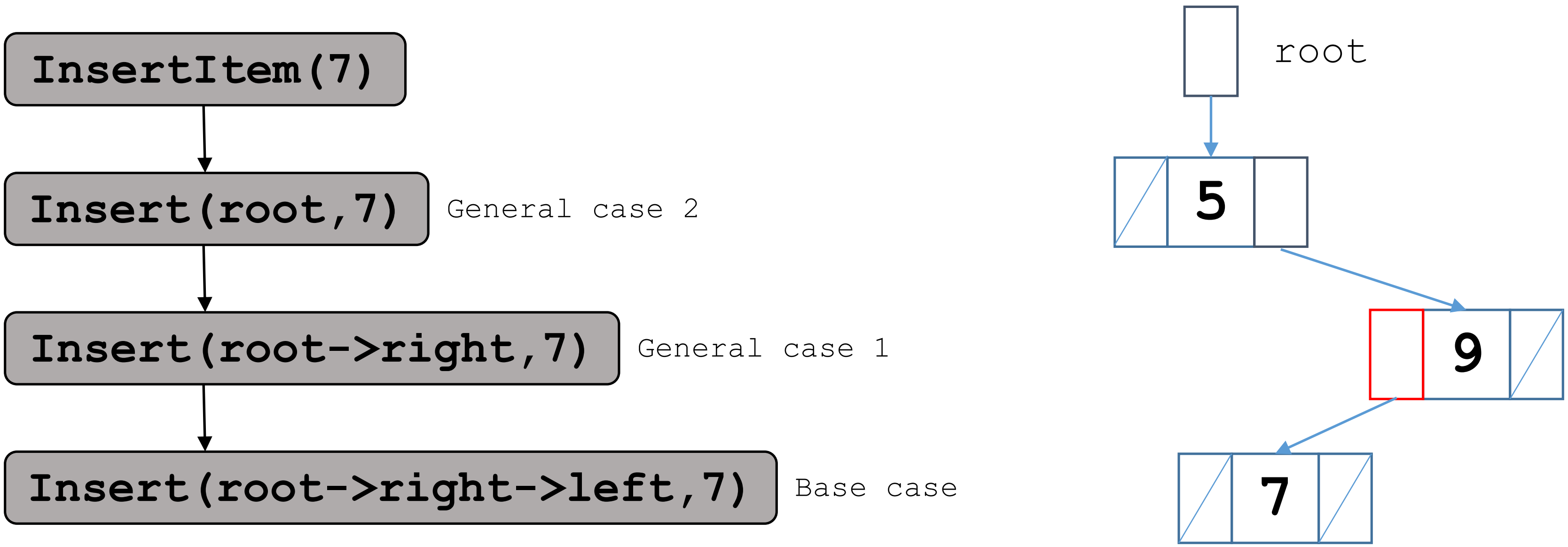
`Insert(root->right,7)` General case 1

↓

`Insert(root->right->left,7)` Base case



Function `InsertItem`

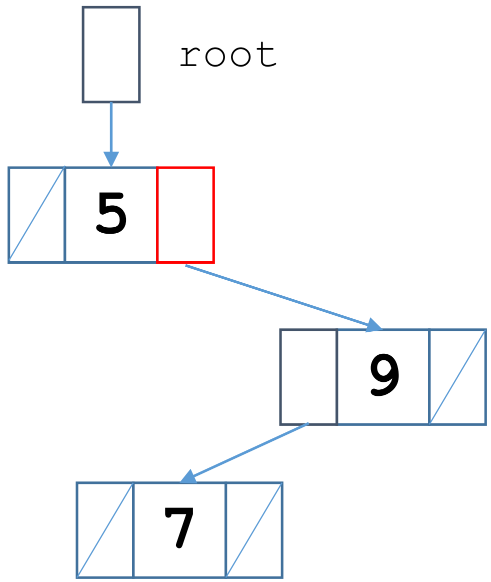


Function `InsertItem`

`InsertItem(7)`

`Insert(root,7)` General case 2

`Insert(root->right,7)` General case 1

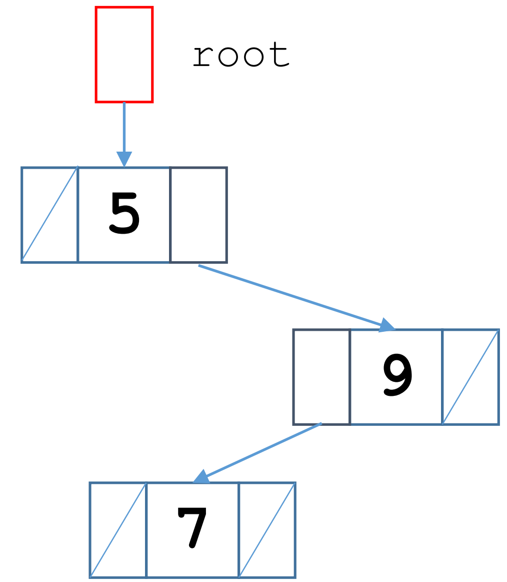


Function `InsertItem`

`InsertItem(7)`

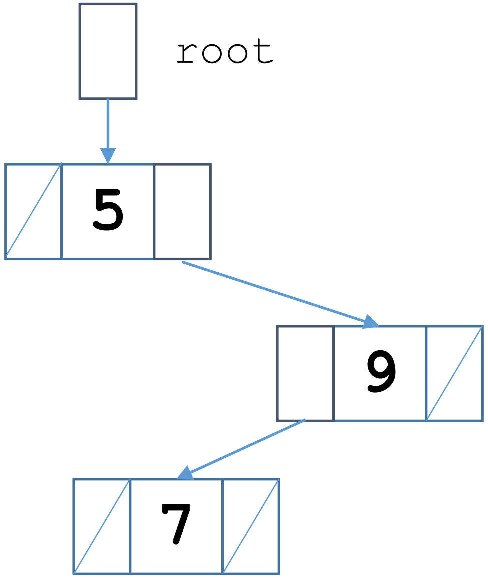
↓

`Insert(root,7)` `General case 2`

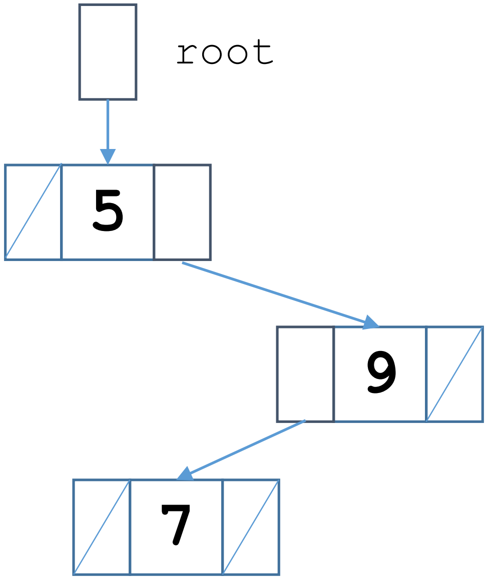


Function `InsertItem`

`InsertItem(7)`

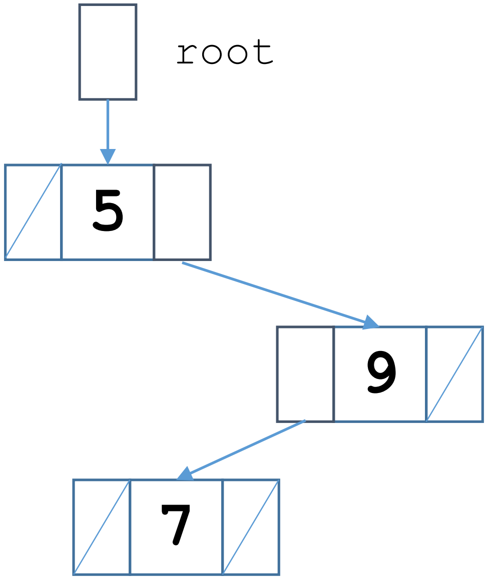


Function `InsertItem`

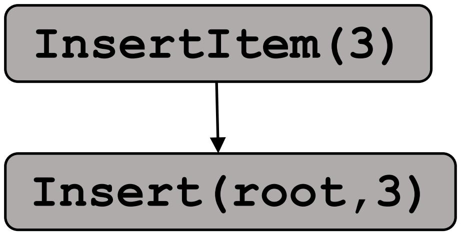


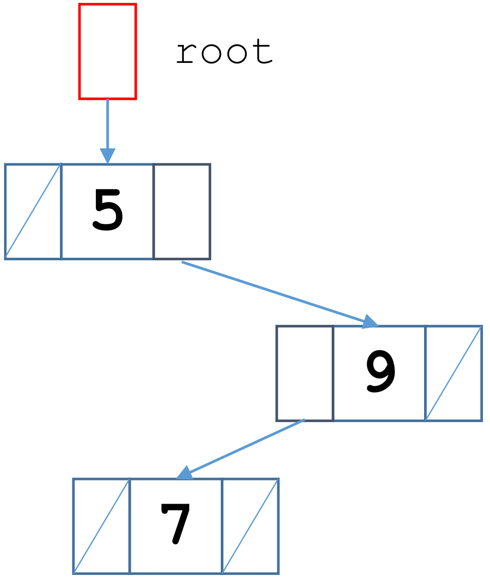
Function `InsertItem`

`InsertItem(3)`



Function `InsertItem`

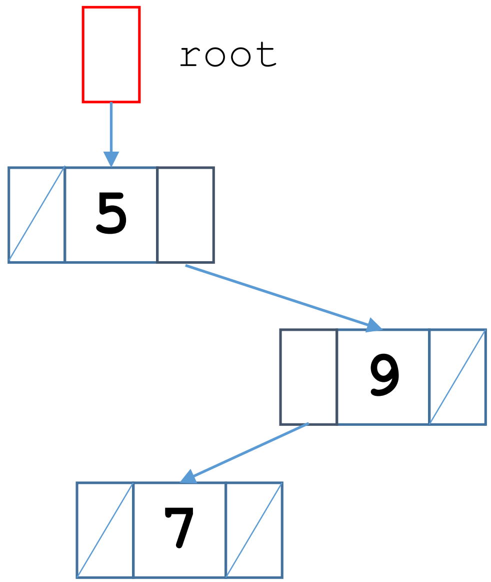




Function `InsertItem`

`InsertItem(3)`

`Insert(root,3)` General case 1



Function `InsertItem`

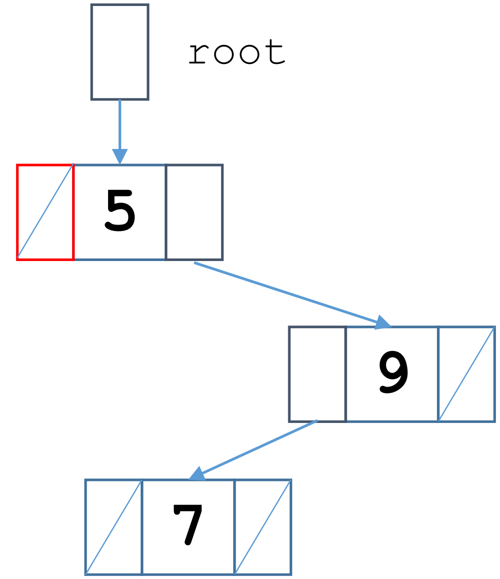
`InsertItem(3)`

↓

`Insert(root,3)` General case 1

↓

`Insert(root->left,3)`



Function `InsertItem`

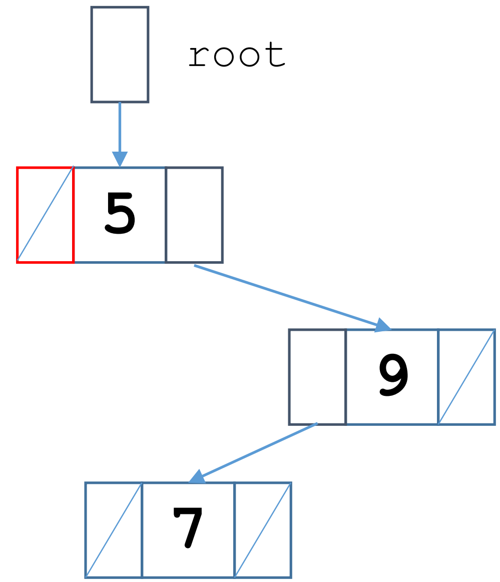
`InsertItem(3)`

↓

`Insert(root,3)` General case 1

↓

`Insert(root->left,3)` Base case

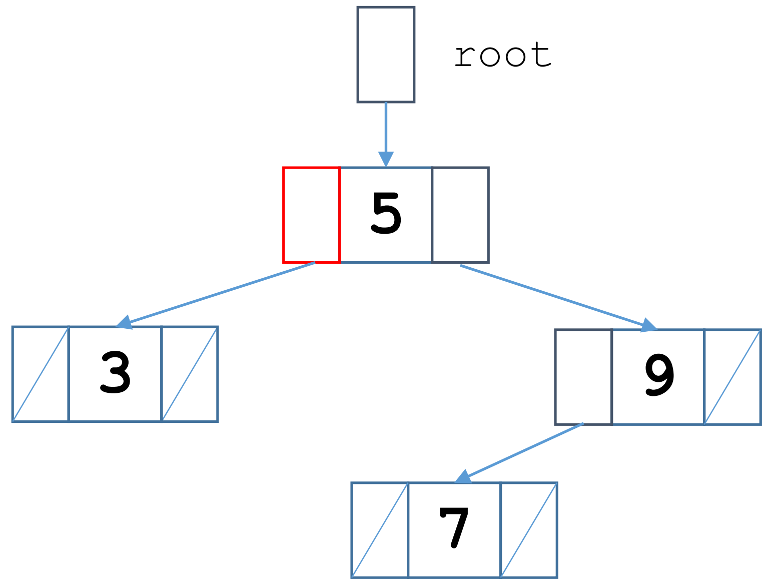


Function `InsertItem`

`InsertItem(3)`

`Insert(root,3)` General case 1

`Insert(root->left,3)` Base case

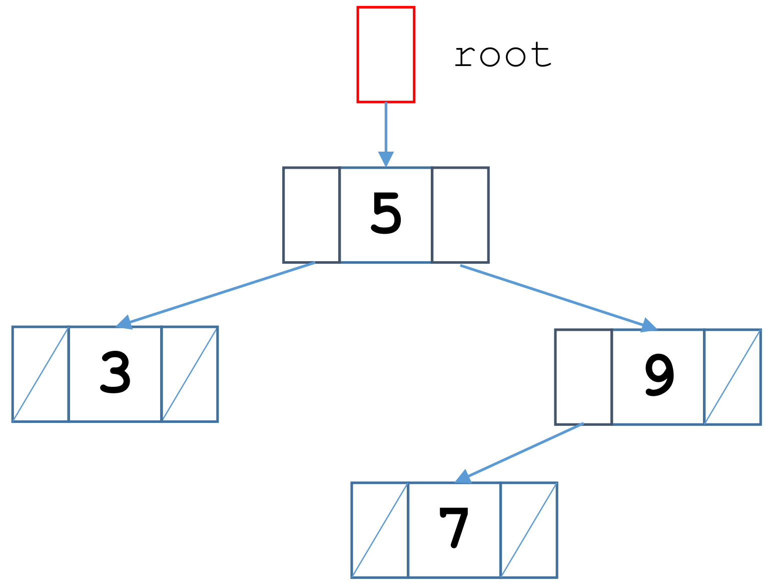


Function `InsertItem`

InsertItem(3)

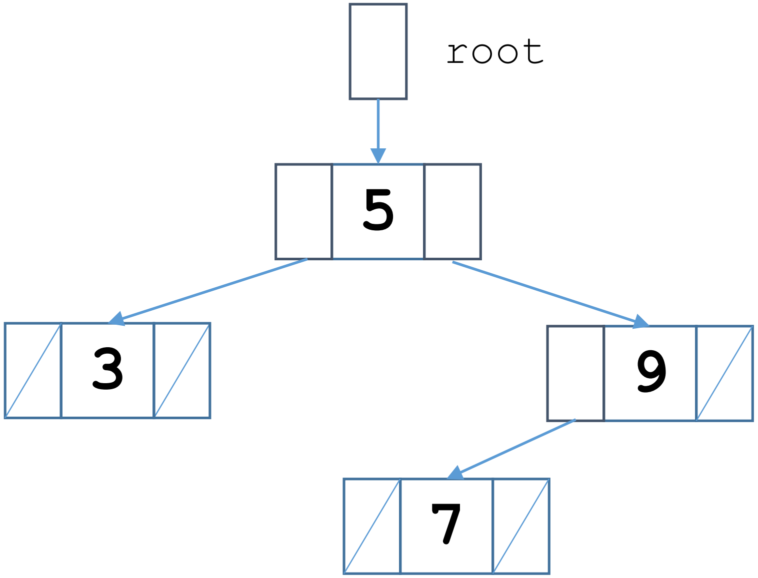
↓

Insert(root,3) General case 1

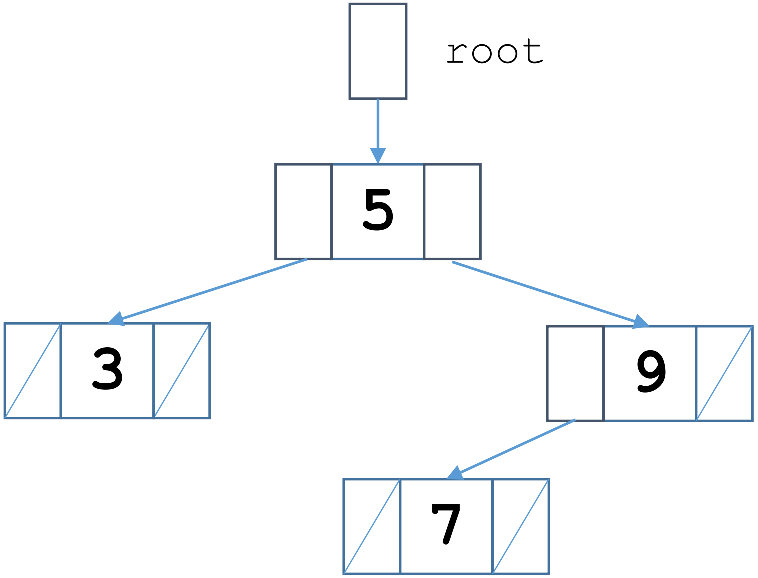


Function `InsertItem`

`InsertItem(3)`

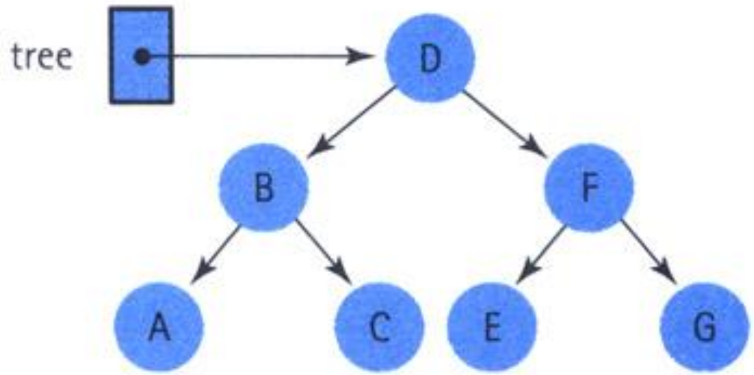


Function `InsertItem`



Function `InsertItem`

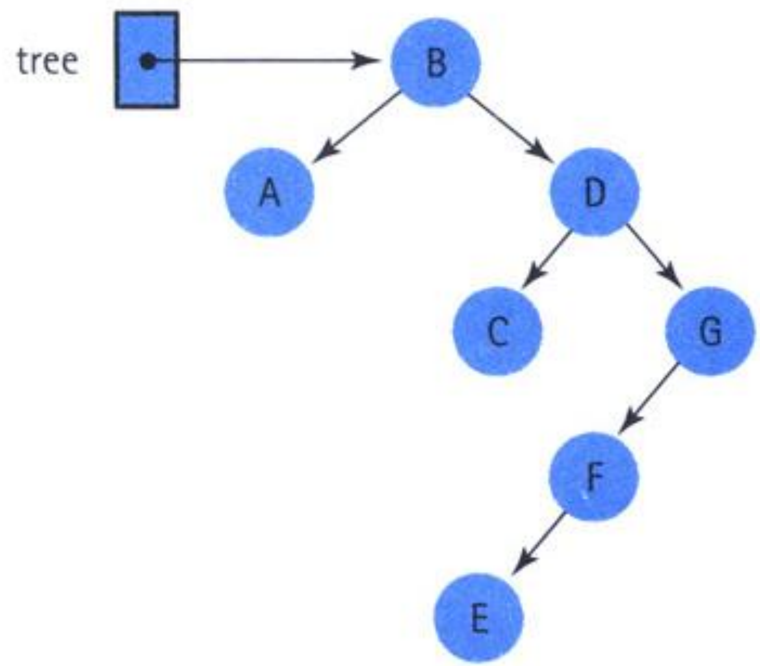
Impact of order of insertion on tree height



(a) Input: D B F A C E G

Function `InsertItem`

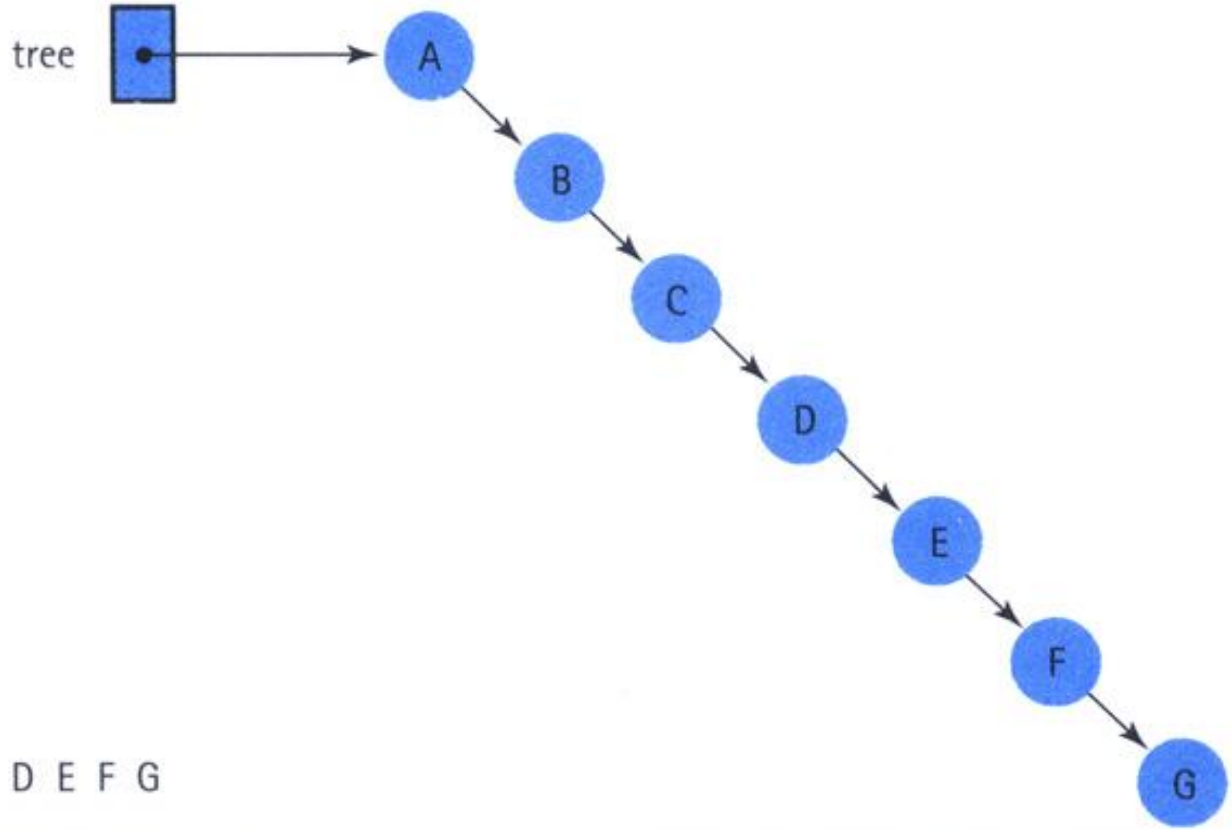
Impact of order of insertion on tree height



(b) Input: B A D C G F E

Function `InsertItem`

Impact of order of insertion on tree height



(c) Input: A B C D E F G

Function `InsertItem`

```
void Insert(TreeNode *&tree, ItemType item)
{
  if (tree == NULL)//Base case:Insertion place found.
  {
    tree = new TreeNode;
    tree->right = NULL;
    tree->left = NULL:
    tree->info = item;
  }
  else if (item < tree->info)
    Insert(tree->left, item);//General case 1:Insert in left subtree.
  else
    Insert(tree->right, item);//General case 2:Insert in right subtree.
}

void TreeType::InsertItem(ItemType item)
{
  Insert(root, item):
}
```

Worst case: **O(N)**

Best case: **O(logN)**