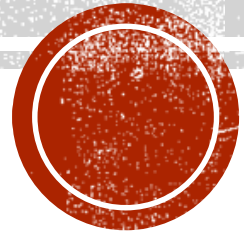
M4: BINARY TREES AND BINARY SEARCH THREES

Summer 2019 – CSC 395 – ST: Algorithm & Data Structure Concepts



SOME RESOURCES

- https://legacy.gitbook.com/book/cathyatseneca/data-structures-and-algorithms/details
- See chapter 5 from http://people.cs.vt.edu/~shaffer/Book/C++3e20130328.pdf

SOME RESOURCES(2)

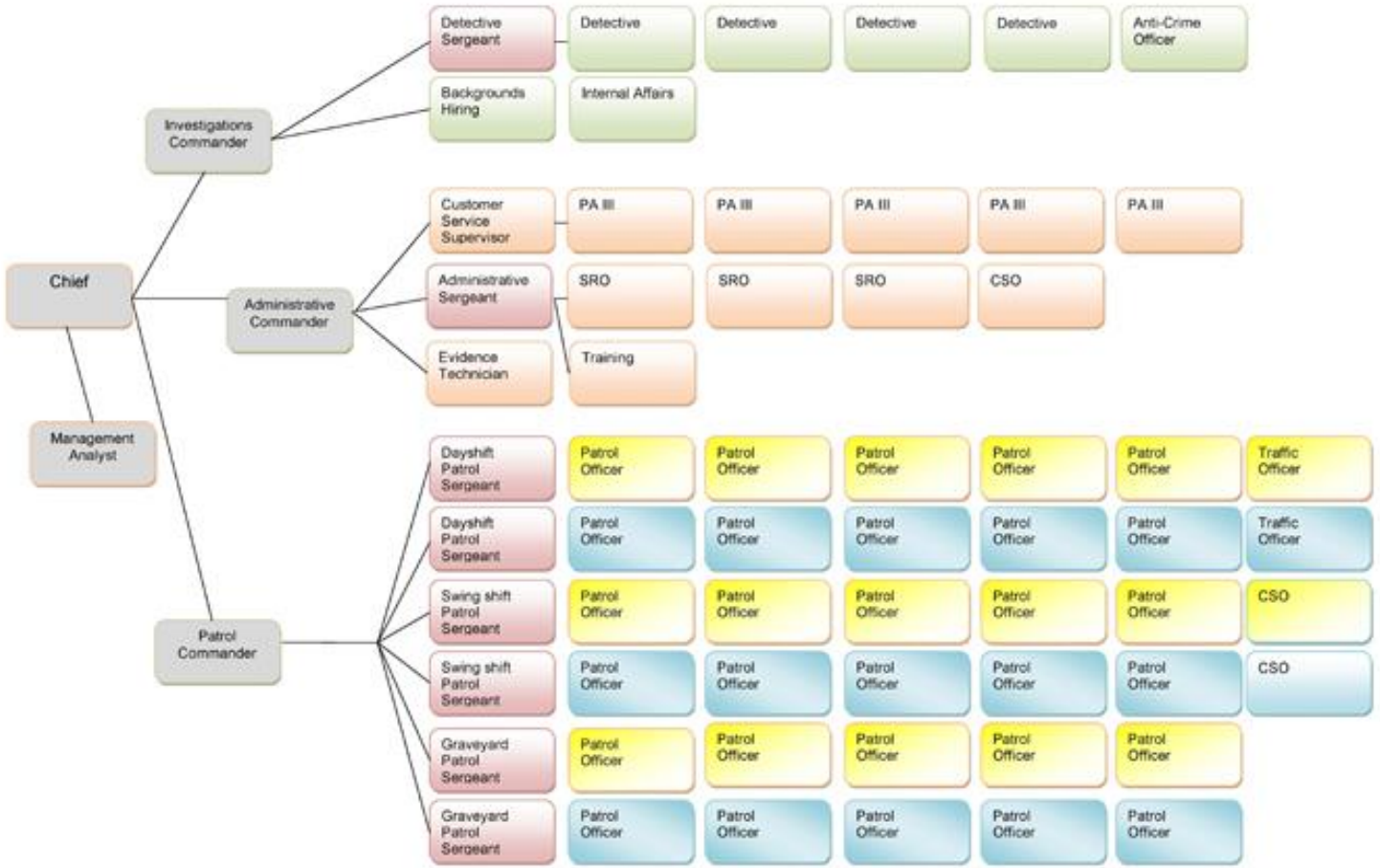
- visualization:
  - binary search tree: https://visualgo.net/en/bst?slide=1
  - liang - binary search tree: http://www.cs.armstrong.edu/liang/animation/web/BST.html

TREE

- A tree is a finite set of nodes connected by edges (containing no cycles).

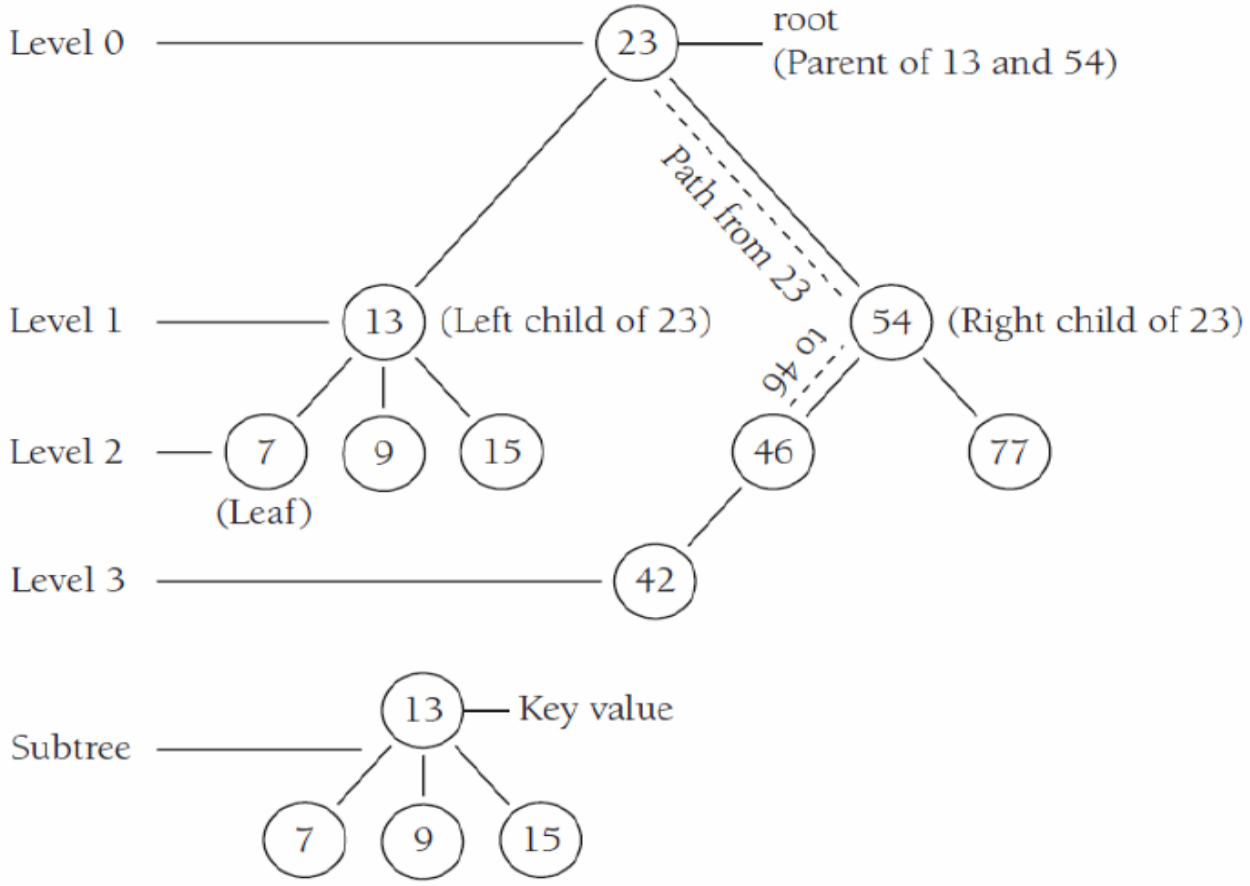
- An example of a tree is a company's organization chart (see Lacey PD chart below).
  - Each box is a node and the lines connecting the boxes are the edges.
  - The nodes represent the entities (people) that make up an organization.
  - The edges represent the relationship between the entities.
  - For example, the Administrative commander reports directly to the Chief, so there is an edge between these two nodes

LACEY POLICE DEPARTMENT ORGANIZATIONAL CHART



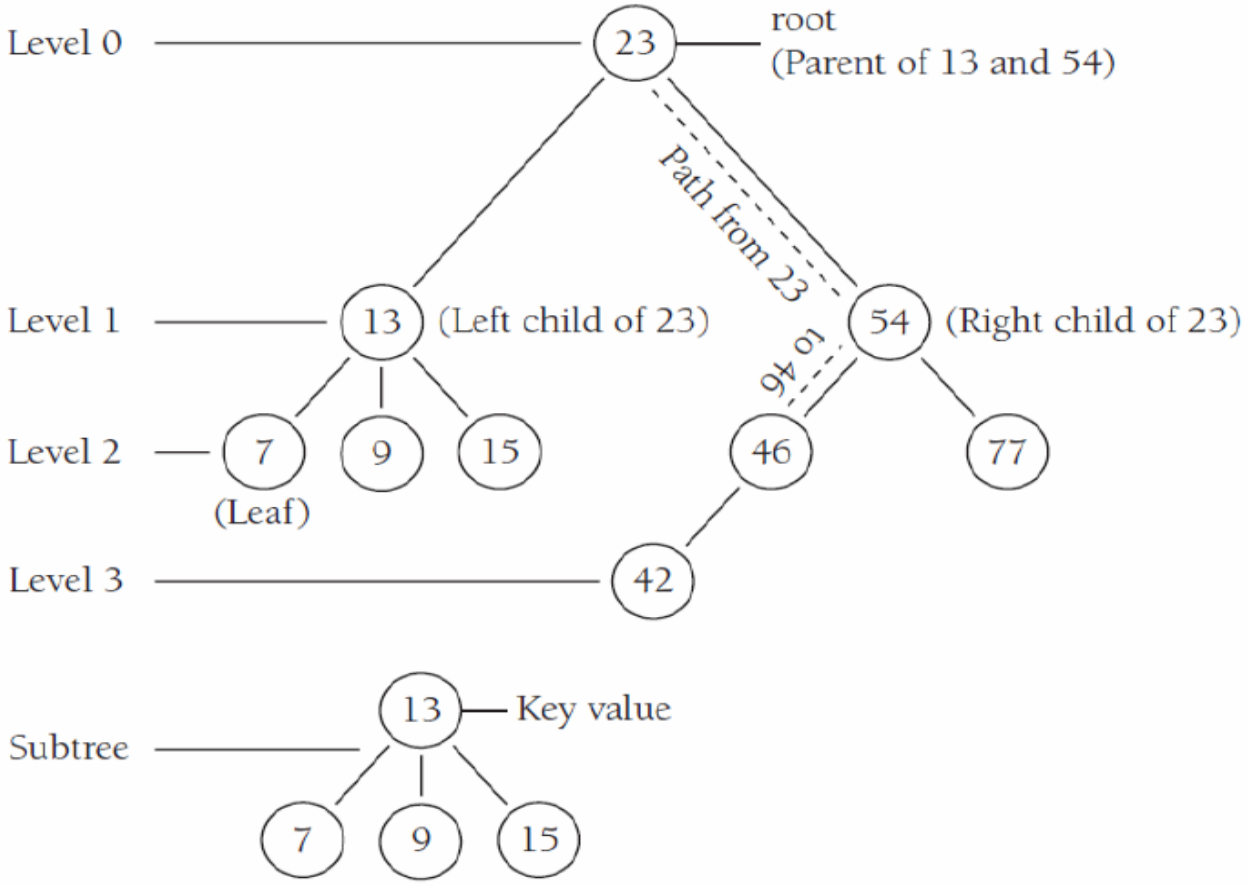
PARTS OF A TREE

- root: The top node of a tree is called the root node.
- leaf (external node): A node without any child node is called a leaf.
- internal node: A node that not a leaf
- parent and children: If a node is connected to other nodes below it, the top node is called the parent, and the nodes below it are called the parent's children.
- siblings: Nodes with the same parent
- edge: A connection between a parent and its child node



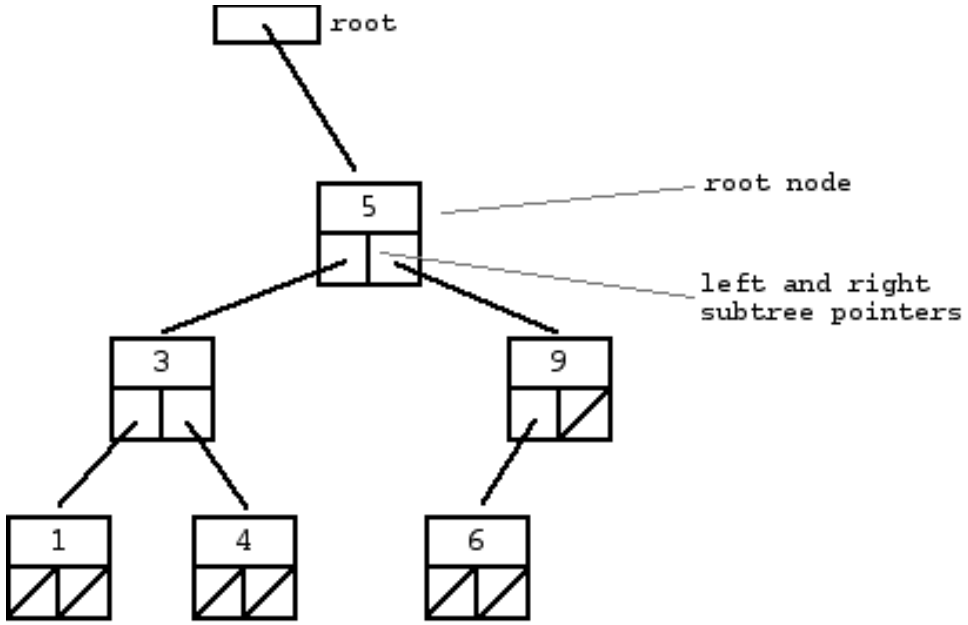
PARTS OF A TREE(2)

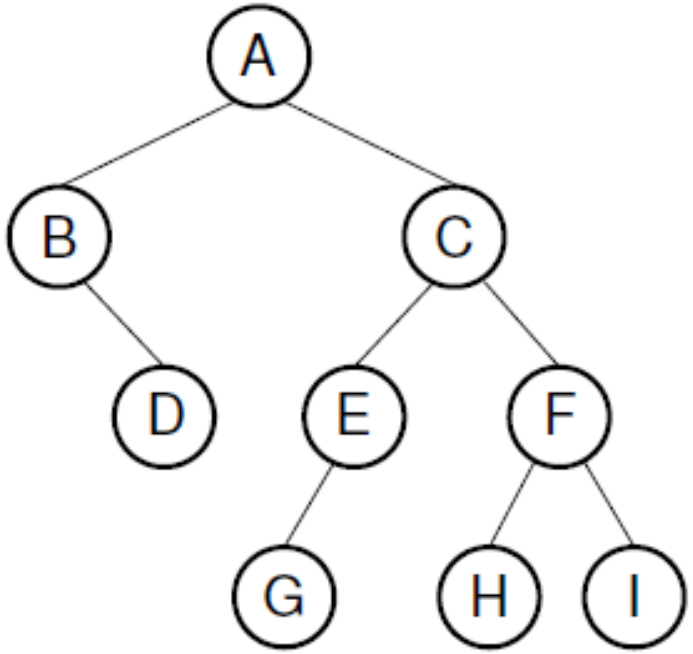
- levels: The root node is at Level 0, its children at Level 1, those node's children are at Level 2, and so on.
- subtree: A node at any level is considered the root of a subtree, which consists of that root node's children, its children's children, and so on.
- depth of a node: the number of edges from the root to the node
- depth of a tree: The number of edges from the root to the deepest leaf
- height of a node: The number of edges from the node to the deepest leaf



BINARY TREES

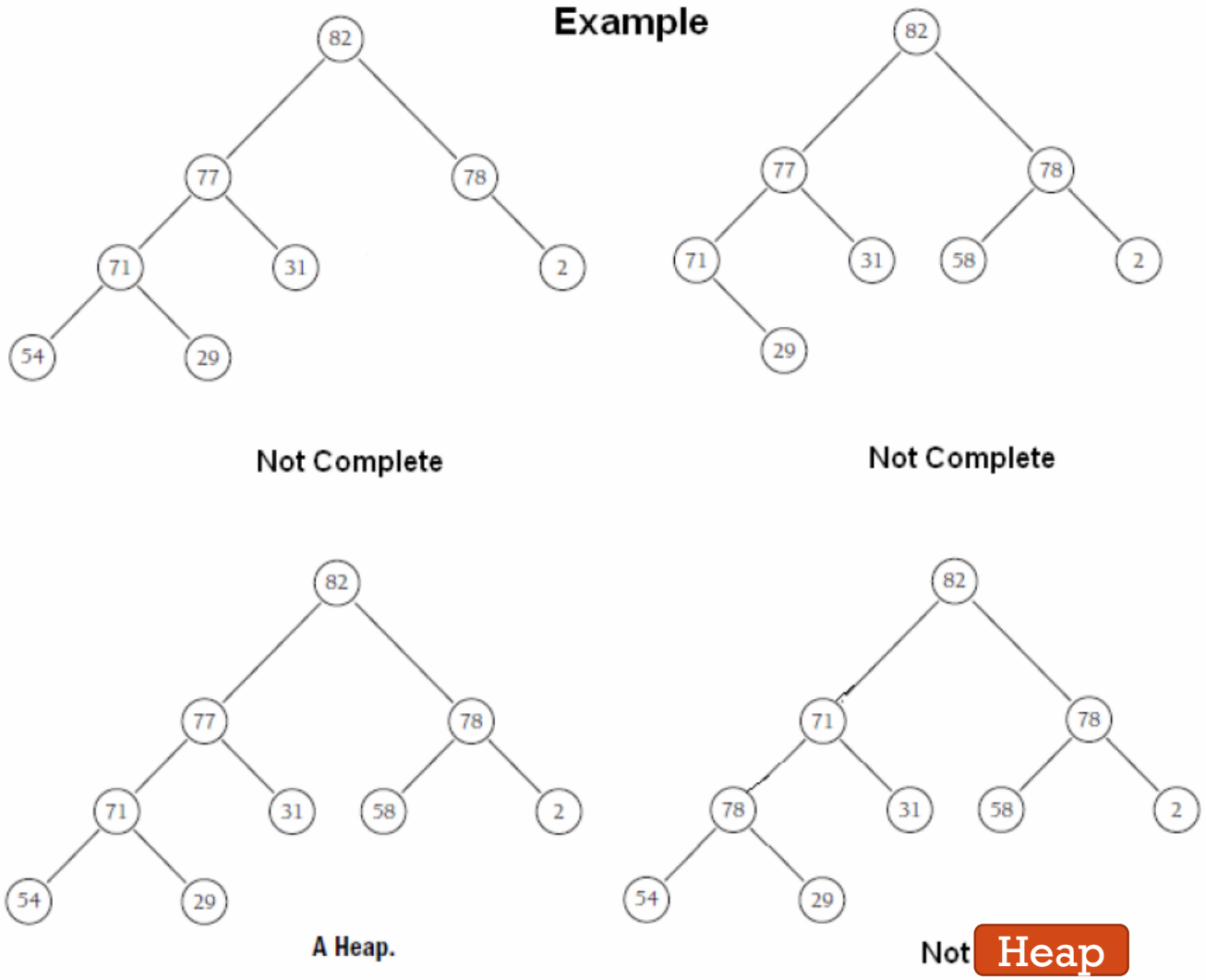
- binary trees: A tree where each node can have no more than two children
- Binary trees restrict the number of children to no more than two: left node and right node





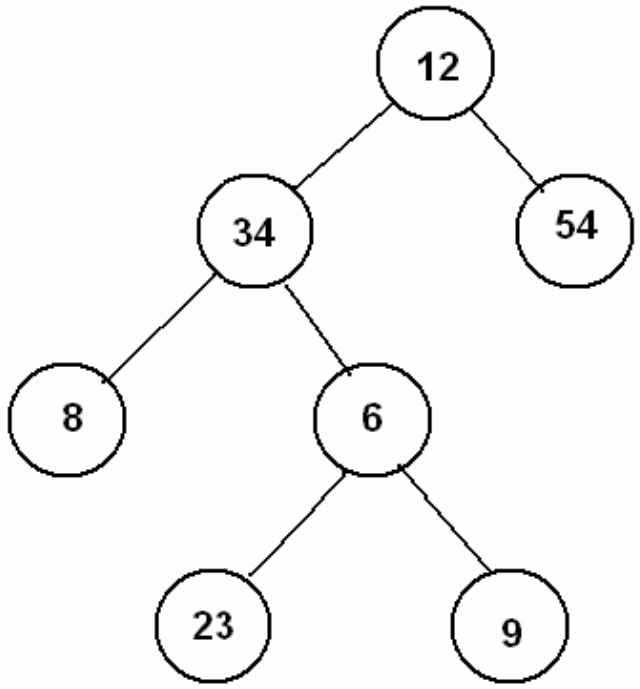
COMPLETE BINARY TREES

- A binary tree is a complete binary tree if
  - every level of the tree is full except that the last level may not be full and
  - all the leaves on the last level are placed left-most.
  - Heap is one example …

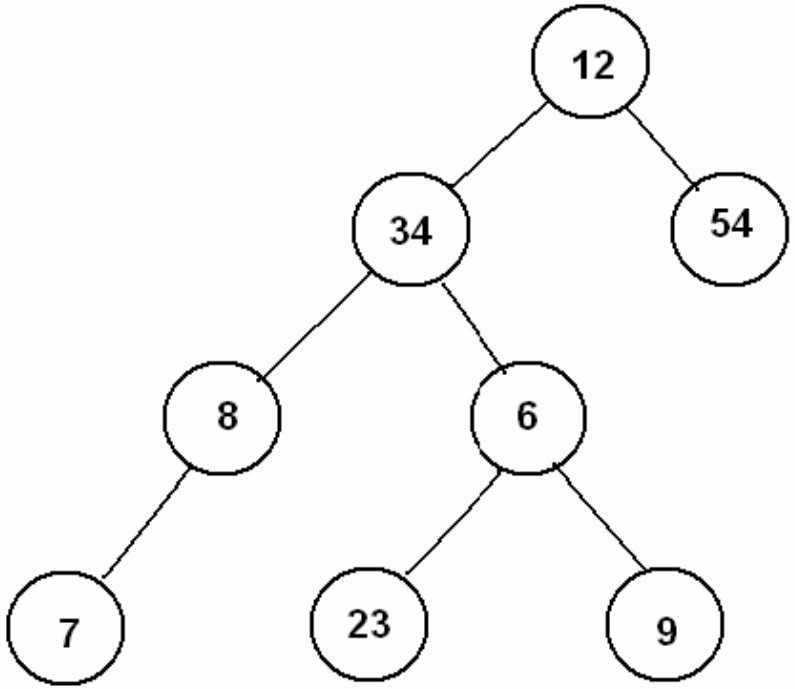


FULL BINARY TREES

- Full binary trees:
  - Each node has exactly zero or two children



a full binary tree

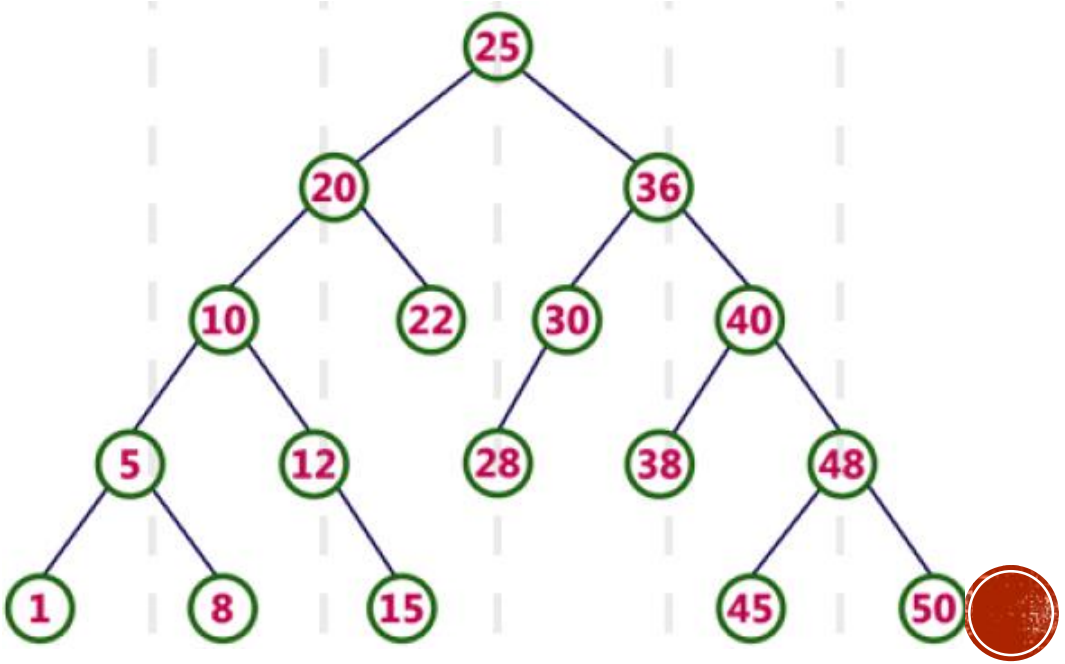


a not full binary tree

BINARY SEARCH TREES (BST)

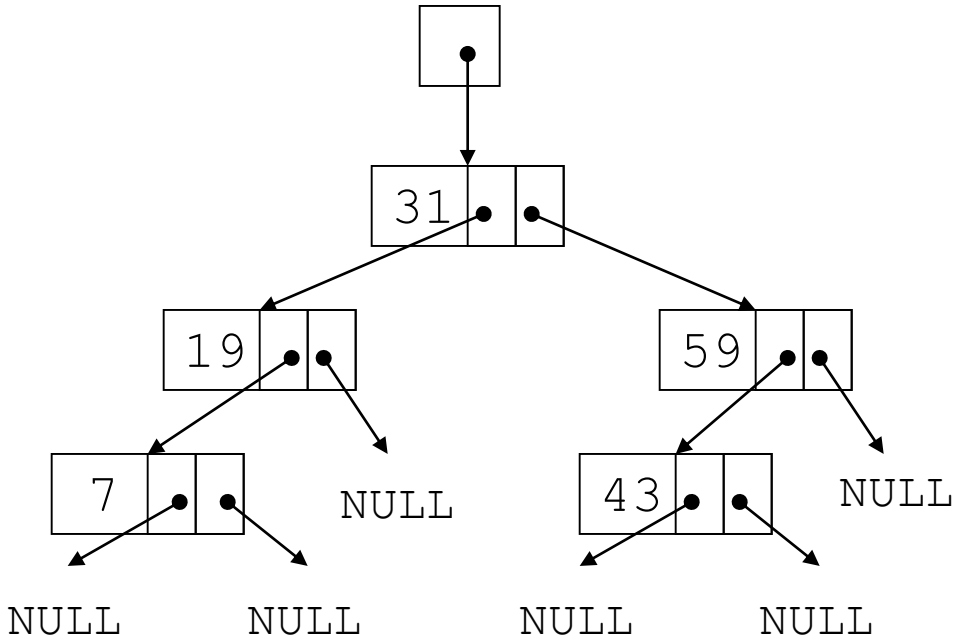
- A binary search tree is a binary tree where the nodes are arranged in order:
  - for each node:
    - all elements in its left subtree are less-or-equal to the node (<=), and
    - all the elements in its right subtree are greater than the node (>).

- Binary search trees are fast at insert and lookup
  - On average, a binary search tree algorithm can locate a node in an N node tree in order log(N) time
  - Therefore, binary search trees are good for "dictionary" problems where the code inserts and looks up information indexed by some key.
  - The log(N) behavior is the average case it's possible for a particular tree to be much slower depending on its shape.



BST – BST CLASS OPERATIONS

- Create a binary search tree – organize data into a binary search tree
- Insert a node into a binary tree – put node into tree in its correct position to maintain order
- Find a node in a binary tree – locate a node with particular data value
- Delete a node from a binary tree – remove a node and adjust links to maintain binary tree



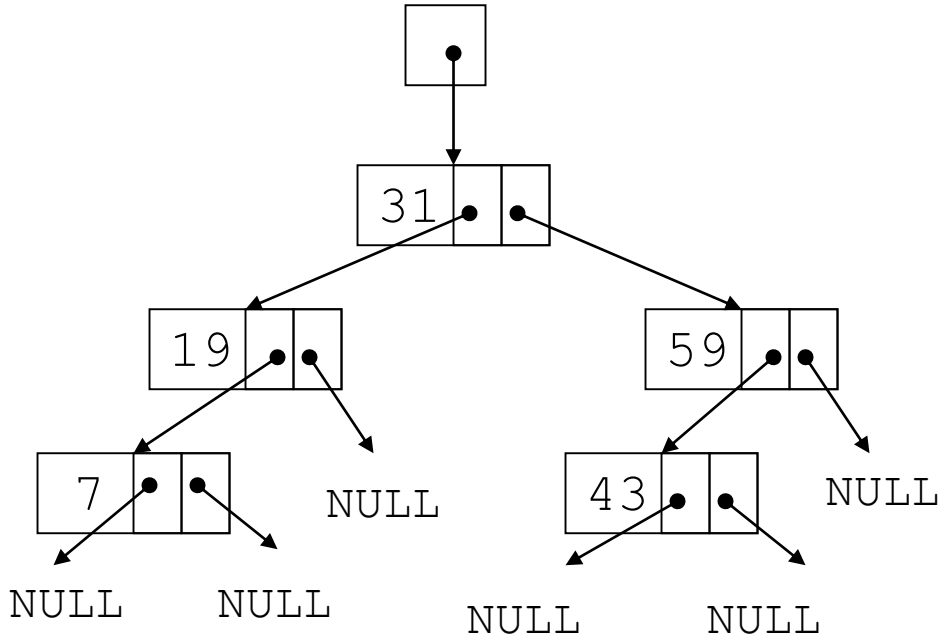
- Traverse the tree
- Optional: find Max/Min values.

BST – NODE CLASS

- A binary search tree is made up of nodes, so we need a Node class
  - The displayNode method allows us to display the data stored in a node.
- This particular Node class holds integers, but we could adopt the class easily to hold any type of data, or even declare Data of Object type if we need to.

```
public class Node
{
    public int Data;
    public Node Left;
    public Node Right;
    public void DisplayNode()
    {
        Console.Write(Data + " ");
    }
 }
```

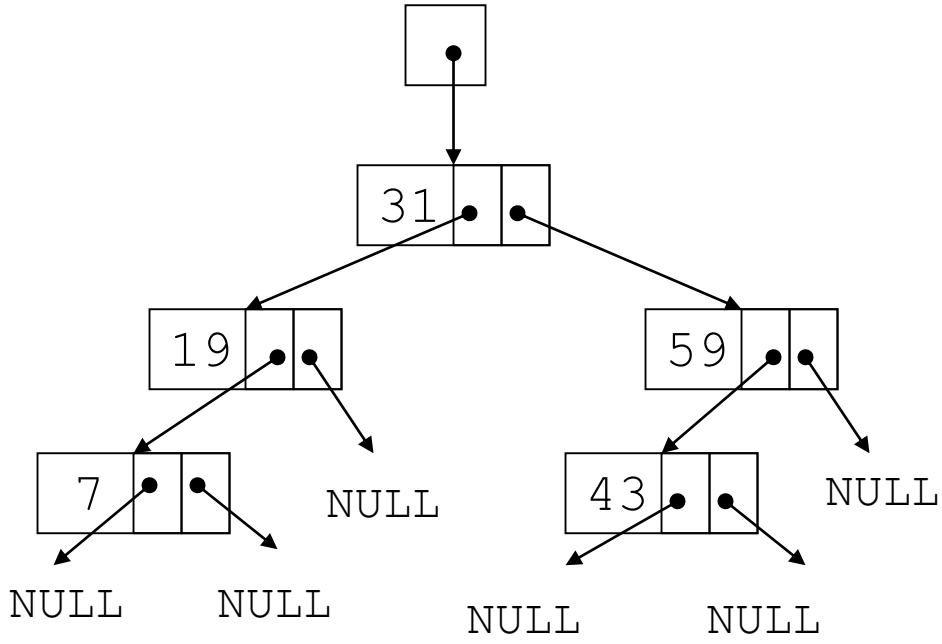
BST - SEARCHING



1) Start at root node
2) Examine node data:
   a) Is it desired value? Done
   b) Else, is desired data < node data? Repeat step 2 with left subtree
   c) Else, is desired data > node data? Repeat step 2 with right subtree
3) Continue until desired value found or `NULL` pointer reached
4) Can be implemented as a Boolean method
   a) Search for 43? return true
   b) Search for 17? return false

BST – SEARCHING (2)

```
public Node Find(int key) {
    Node current = root;
    while (current.iData != key) {
        if (key < current.iData)
            current = current.Left;
        Else
            current = current.Right;
        if (current == null)
            return null;
    }
    return current;
}
```



BST – INSERTING A NEW NODE

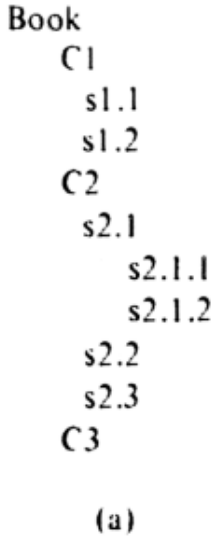
1. Create a Node object and assign the data the Node holds to the Data variable.
2. If tree is empty, insert the new node as the root node
3. Else, compare new node against left or right child, depending on whether data value of new node is < or > root node
4. Continue comparing and choosing left or right subtree until `NULL` pointer found
5. Set this `NULL` pointer to point to new node

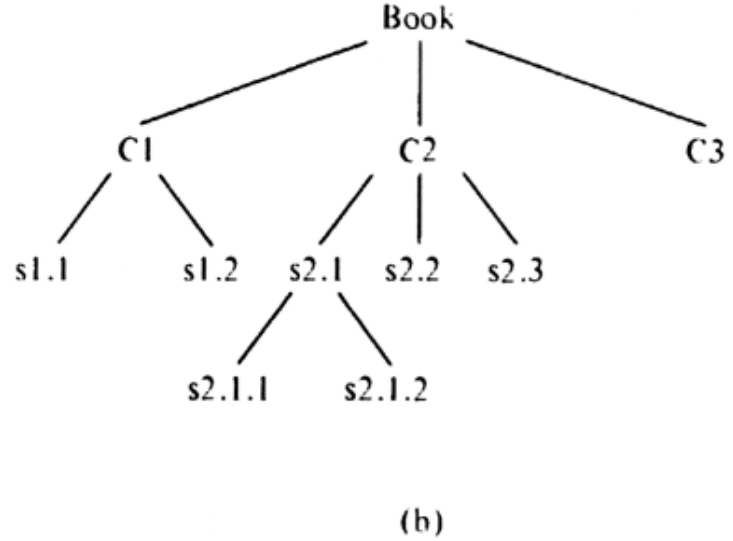
BST – INSERTING A NEW NODE (2)

```
public void Insert(int i)
{
    Node newNode = new Node();
    newNode.Data = i;

    if (root == null)
    {
        root = newNode;
    }
    else
    {
        Node current = root;
        Node parent;
        while (true)
        {
            parent = current;
            if (i < current.Data)
            {
                current = current.Left;
                if (current == null)
                {
                    parent.Left = newNode;
                    break;
                }
            }
            else
            {
                current = current.Right;
                if (current == null)
                {
                    parent.Right = newNode;
                    break;
                }
            }
        }//end of while(true)
    }//end of else
}//end of public void Insert(int i)
```

BST - TRAVERSALS





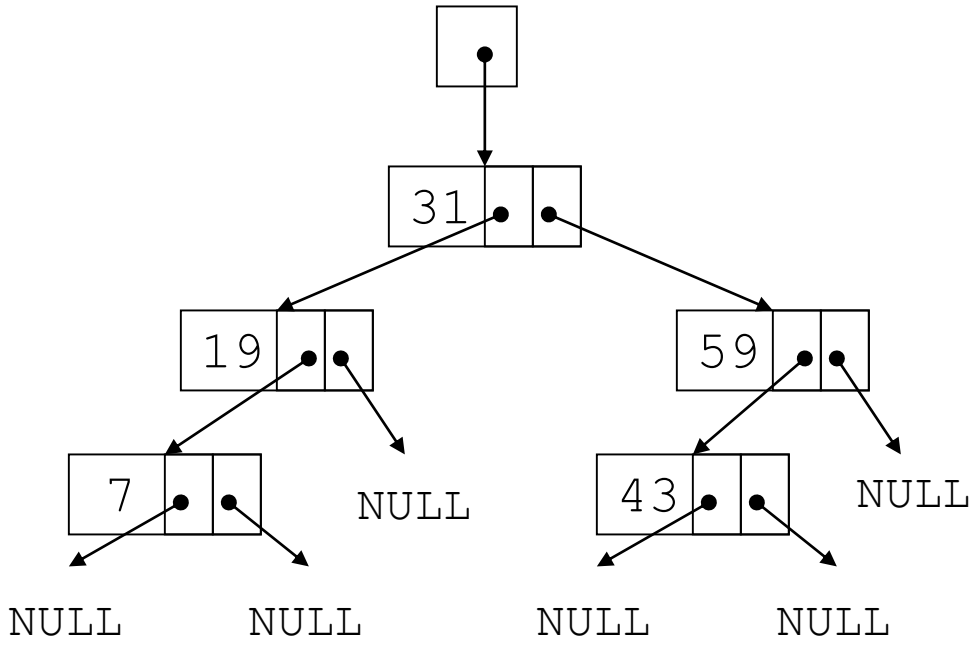
Three traversal methods:

1) Inorder: ←used for sorting
   a) Traverse left subtree of node
   b) Process data in node
   c) Traverse right subtree of node
2) Preorder: ←used for table of contents
   a) Process data in node
   b) Traverse left subtree of node
   c) Traverse right subtree of node
3) Postorder: ←used for computing the size of a folder (and its contents), Polish…
   a) Traverse left subtree of node
   b) Traverse right subtree of node
   c) Process data in node

http://www.cs.armstrong.edu/liang/animation/web/BST.html

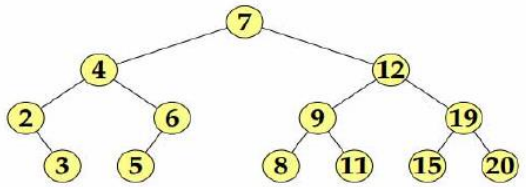
https://visualgo.net/en/bst

BST – TRAVERSALS (2)



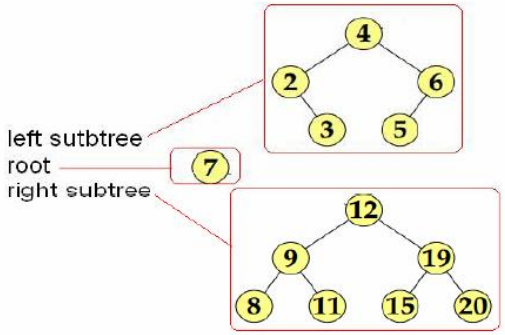
TRAVERSAL METHOD	NODES VISITED IN ORDER
Inorder	7, 19, 31, 43, 59
Preorder	31, 19, 7, 59, 43
Postorder	7, 19, 43, 59, 31

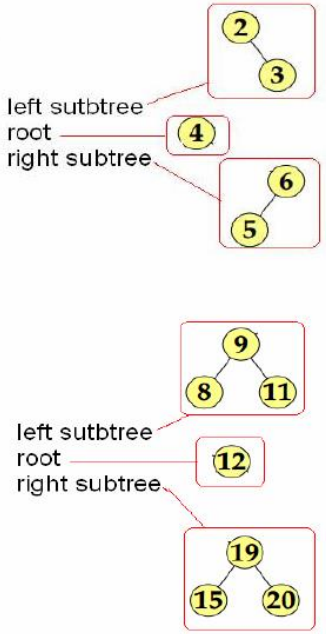
BST – TRAVERSALS (3)

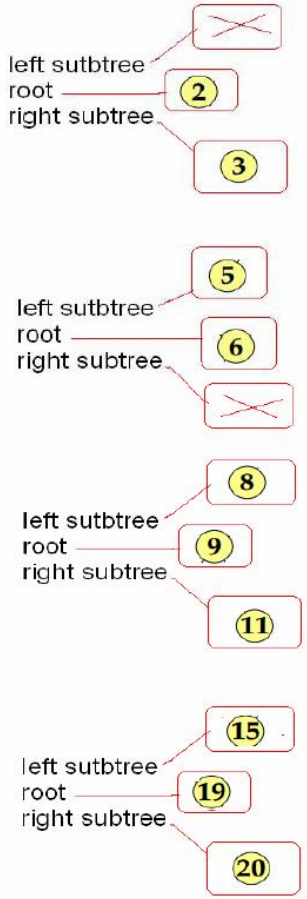


Inorder traversal gives:

2, 3, 4, 5, 6, 7, 8 , 9, 11, 12, 15, 19, 20.







BST – TRAVERSALS (4)

InOrder(root) visits nodes in the following order:

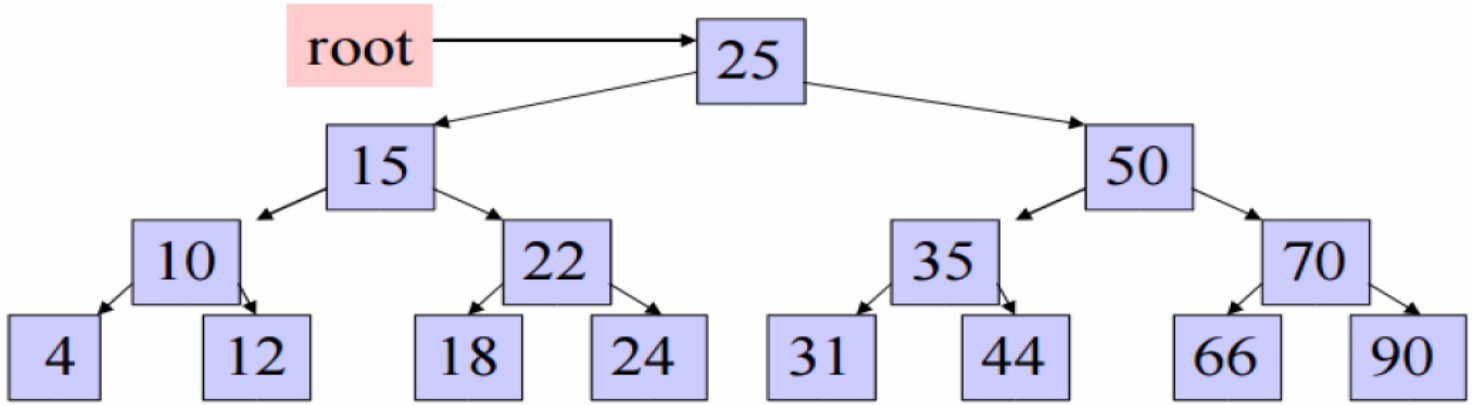
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

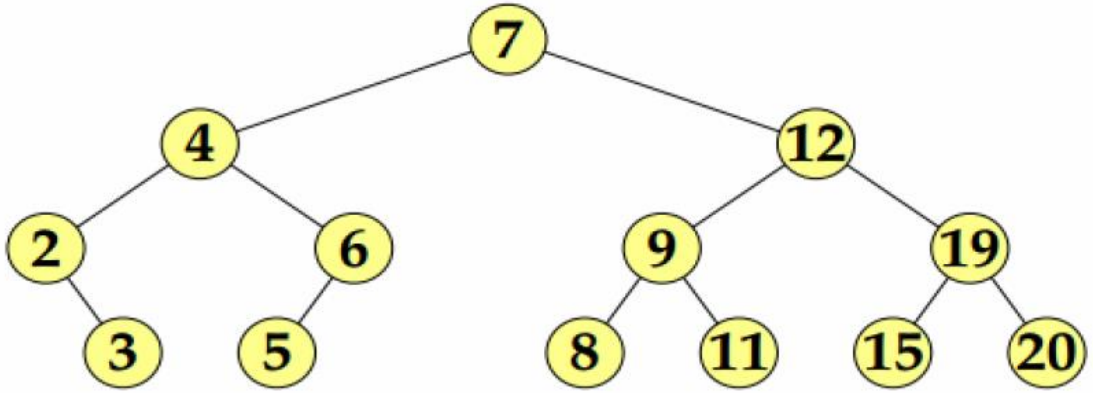
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



BST – TRAVERSALS (5)



Inorder traversal gives:

2, 3, 4, 5, 6, 7, 8 , 9, 11, 12, 15, 19, 20.

Preorder traversal gives:

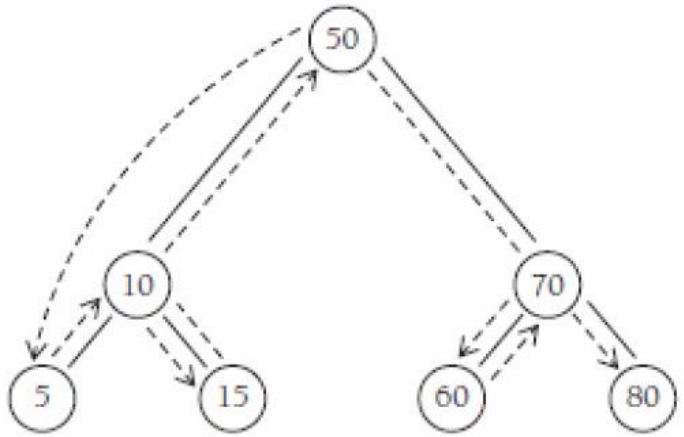
7, 4, 2, 3, 6, 5, 12, 9, 8, 11, 19, 15, 20.

Postorder traversal gives:

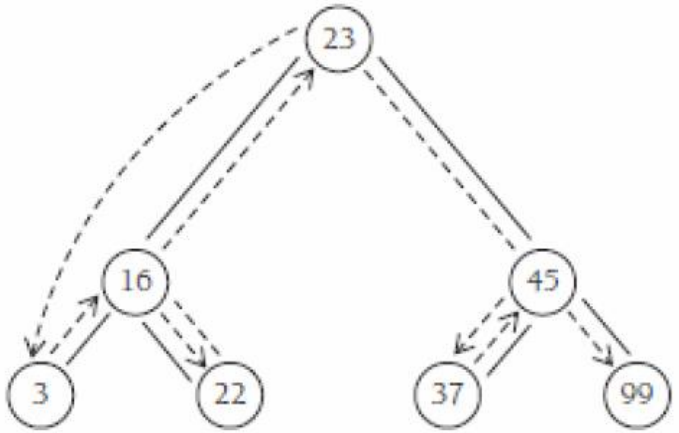
3, 2, 5, 6, 4, 8, 11, 9, 15, 20, 19, 12, 7.

BST – TRAVERSALS (6)

```
public void InOrder(Node theRoot) {
    if (!(theRoot == null)) {
       InOrder(theRoot.Left);
       theRoot.DisplayNode();
       InOrder(theRoot.Right);
    }
```



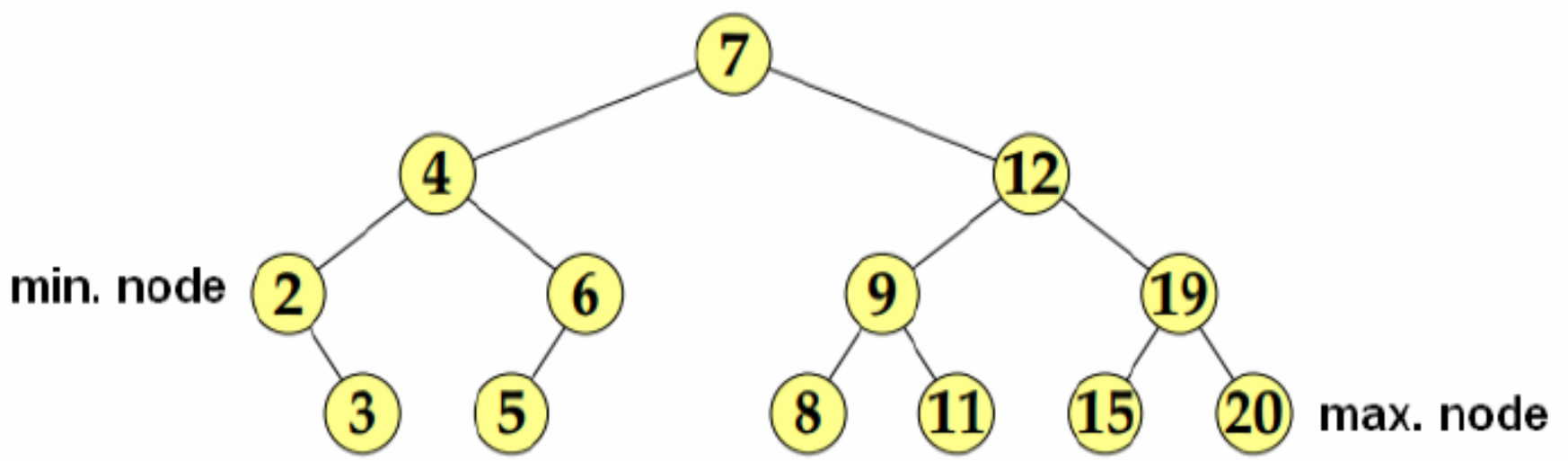
Inorder Traversal Order.



Inorder Traversal Path.

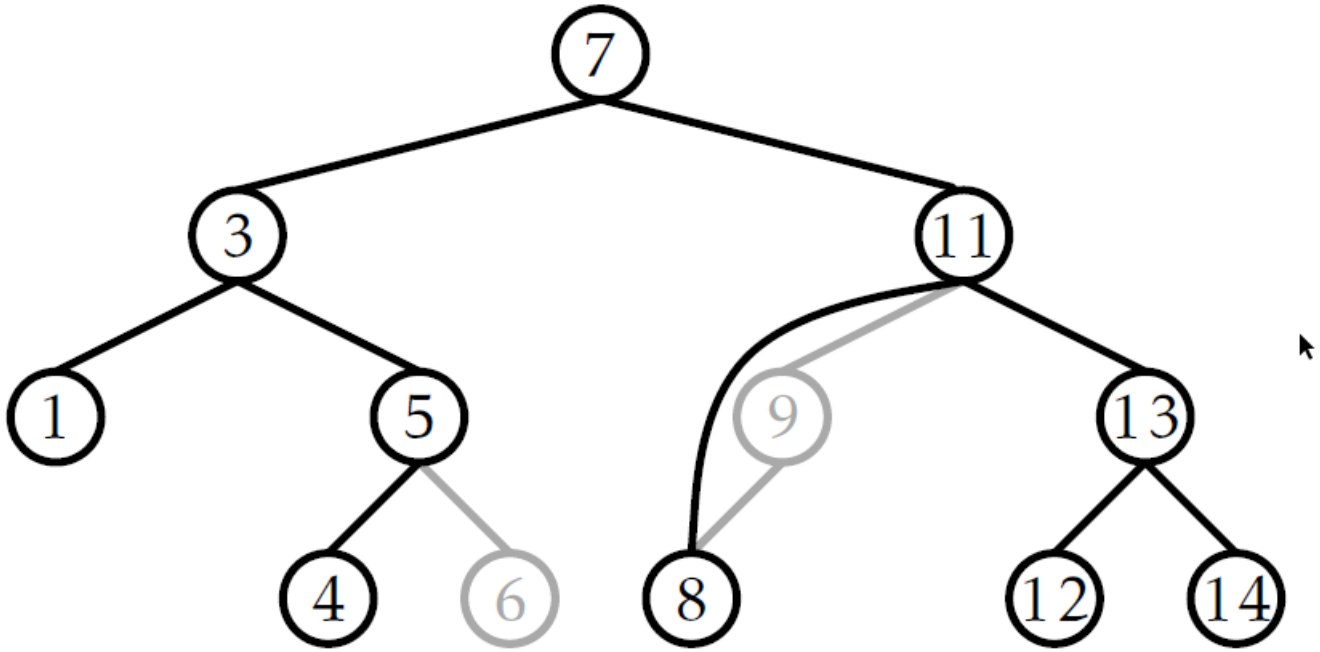
BST – FINDING THE MAX/MIN VALUE

- The smallest value in a BST will always be found at the last left child node of a subtree beginning with the left child of the root node.
- The largest value in a BST is found at the last right child node of a subtree beginning with the right child of the root node.



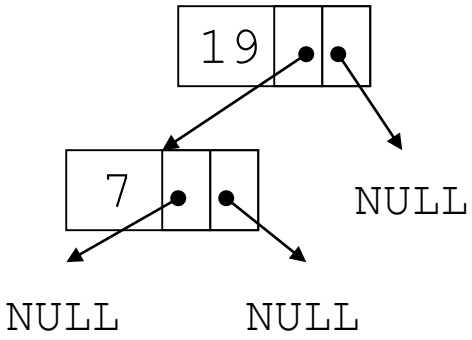
BST – DELETING A NODE – ONE CHILD/LEAF

- Removing a leaf (6) or a node with only one child (9) is easy.
- Things get tricky, though, when u has two children

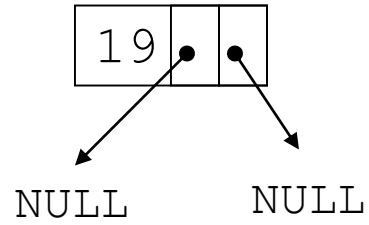


BST – DELETING A NODE – A LEAF NODE

- If node to be deleted is a leaf node, replace parent node's pointer to it with a `NULL` pointer, then delete the node



Deleting node with 7 – before deletion

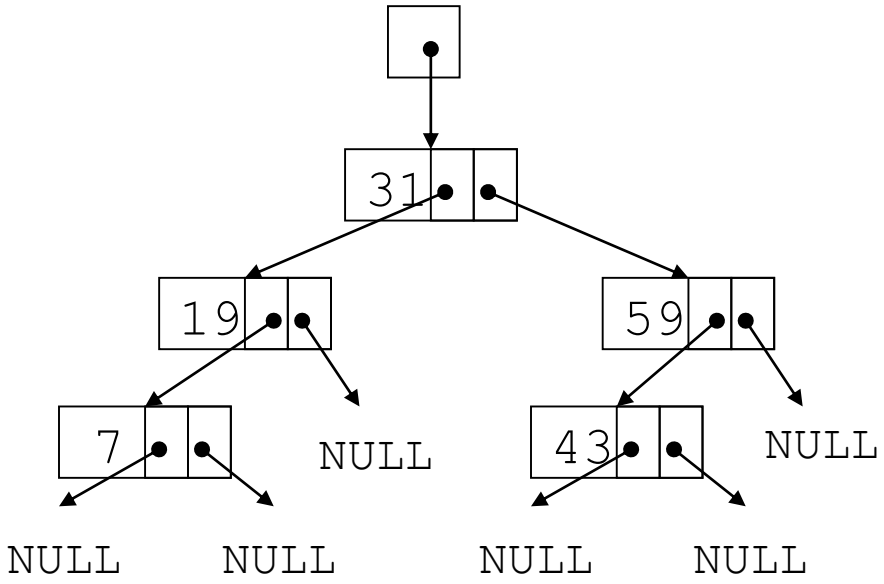


Deleting node with 7 – after deletion

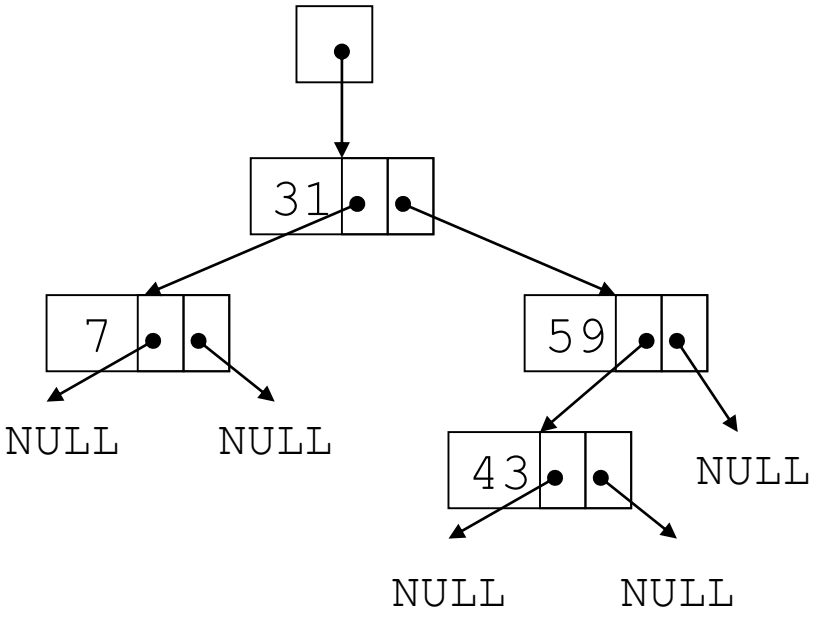
```
if ((current.Left == null) && (current.Right == null))
{
    if (current == root)
        root = null;
    else if (isLeftChild)
        parent.Left = null;
    else
        parent.Right = null;
}
```

BST – DELETING A NODE – ONE CHILD

- If node to be deleted has one child node, adjust pointers so that parent of node to be deleted points to child of node to be deleted, then delete the node



Deleting node with 19
– before deletion

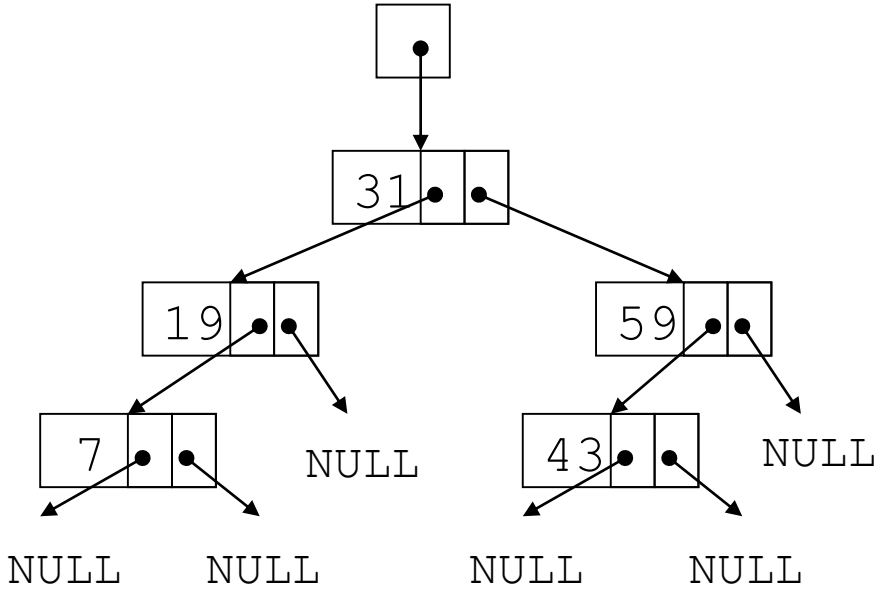


Deleting node with 19
– after deletion

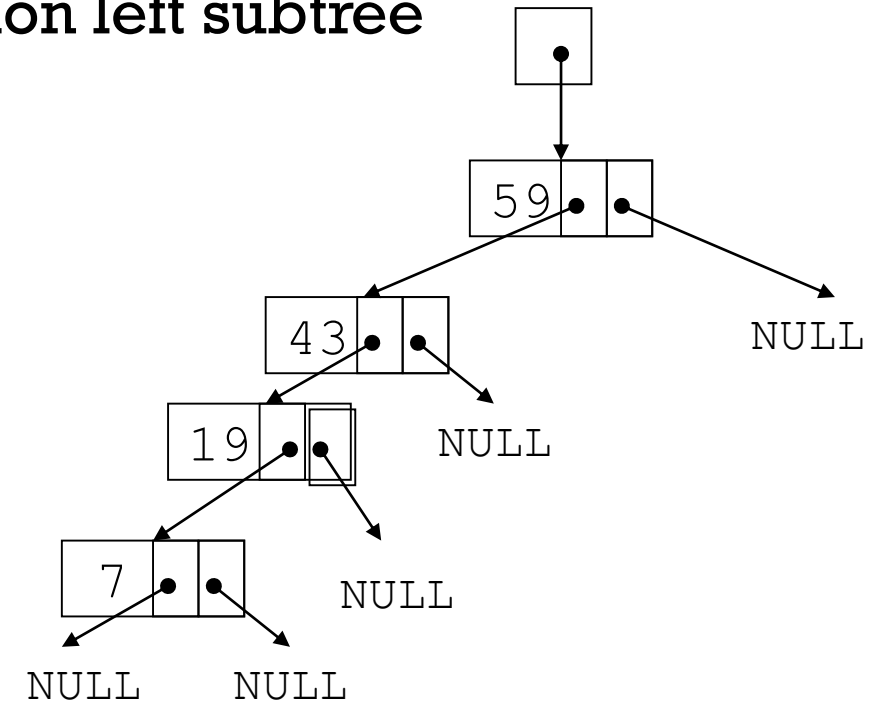
```
else if (current.Right == null)
{
    if (current == root)
        root = current.Left;
    else if (isLeftChild)
        parent.Left = current.Left;
    else
        //parent.Right = current.Right;
        parent.Right = current.Left;
}
else if (current.Left == null)
{
    if (current == root)
        root = current.Right;
    else if (isLeftChild)
        parent.Left = current.Right;
    else
        parent.Right = current.Right;
}
```

BST – DELETING A NODE – TWO CHILDREN

- If node to be deleted has left and right children,
  - 'Promote' one child to take the place of the deleted node
  - Locate correct position for other child in subtree of promoted child
- One way: promote the right child, position left subtree underneath



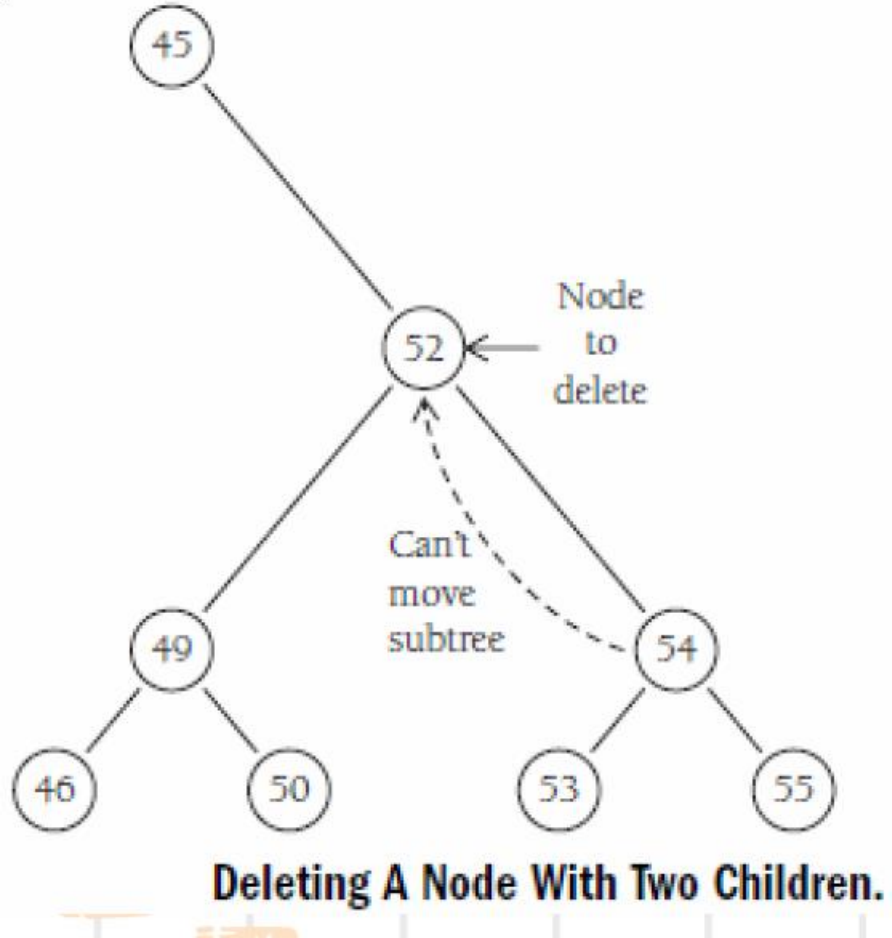
Deleting node with 31 – before deletion



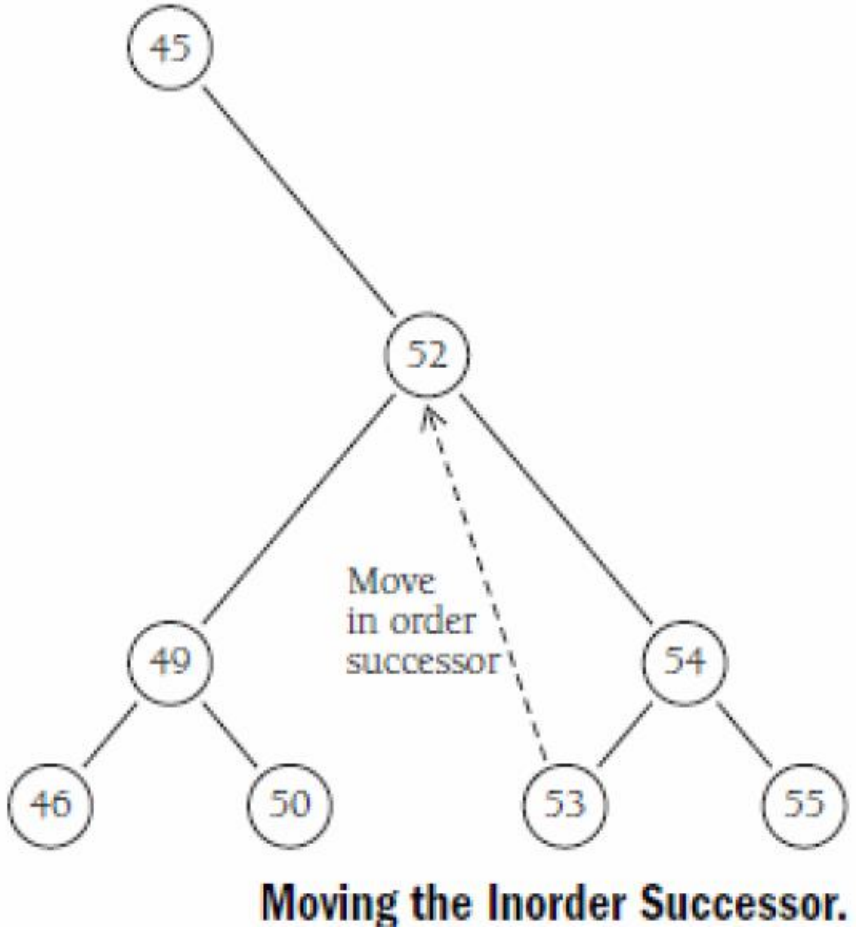
Deleting node with 31 – after deletion

BST – DELETING A NODE – TWO CHILDREN (2)

- Here is another solution



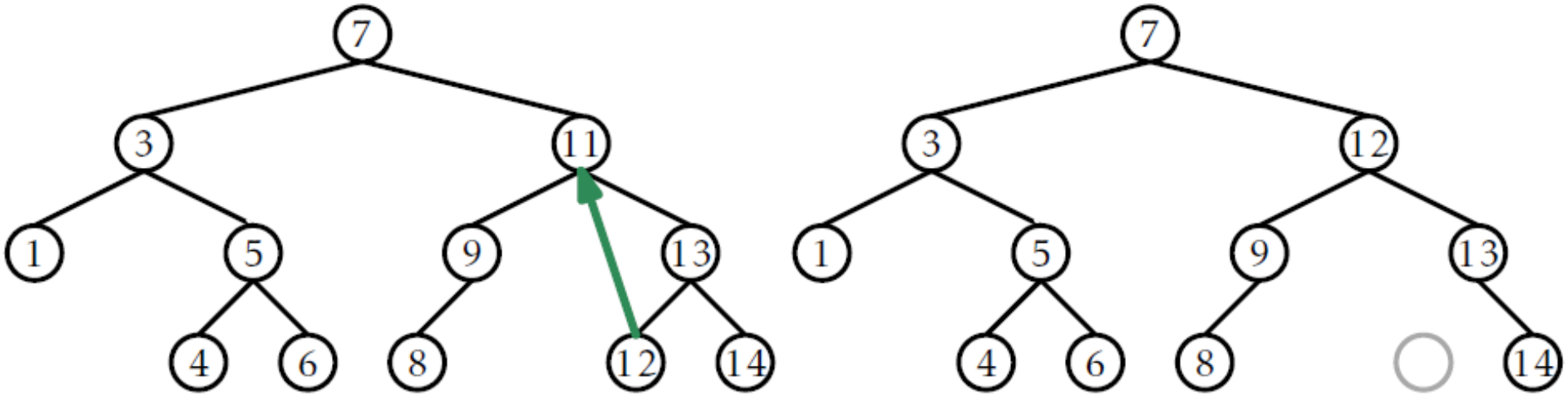
Deleting A Node With Two Children.



Moving the Inorder Successor.

BST – DELETING A NODE – TWO CHILDREN (3)

- Deleting a value (11) from a node, u, with two children is done by replacing u's value with the smallest value in the right subtree of u.



BST – DELETING A NODE – TWO CHILDREN (3)

- Here is another solution

```
public Node GetSuccessor(Node delNode)
{
    Node successorParent = delNode;
    Node successor = delNode.Right;
    Node current = delNode.Right;
    while (current.Left != null)
    {
        successorParent = current;
        successor = current.Left;
        current = current.Left;
    }
    if (successor != delNode.Right)
    {
        successorParent.Left = successor.Right;
        successor.Right = delNode.Right;
    }
    return successor;
}
```

Implementation in C#

```
else
{
    Node successor = GetSuccessor(current);
    if (current == root)
        root = successor;
    else if (isLeftChild)
        parent.Left = successor;
    else
        parent.Right = successor;

    successor.Left = current.Left;
}

public Node GetSuccessor(Node delNode)
{
    Node successorParent = delNode;
    Node successor = delNode.Right;
    Node current = delNode.Right;
    while (current.Left != null)
    {
        successorParent = current;
        successor = current.Left;
        current = current.Left;
    }
    if (successor != delNode.Right)
    {
        successorParent.Left = successor.Right;
        successor.Right = delNode.Right;
    }
    return successor;
}
```

step 3 — parent.Left = successor;

step 4 — successor.Left = current.Left;

step 1 — successorParent.Left = successor.Right;

step 2 — successor.Right = delNode.Right;

OTHER PROBLEMS

- Given a binary tree, count the number of nodes in the tree
- Given a binary tree, compute its "maxDepth" -- the number of nodes along the longest path from the root node down to the farthest leaf node. The maxDepth of the empty tree is 0
- Given a binary tree, print out all of its root-to-leaf paths as defined above. This problem is a little harder than it looks, since the "path so far" needs to be communicated between the recursive calls
- Change a tree so that the roles of the left and right pointers are swapped at every node.
- Write an isBST() function that returns true if a tree is a binary search tree and false otherwise

INTERVIEW QUESTIONS (JUST FOR PRACTICE!)

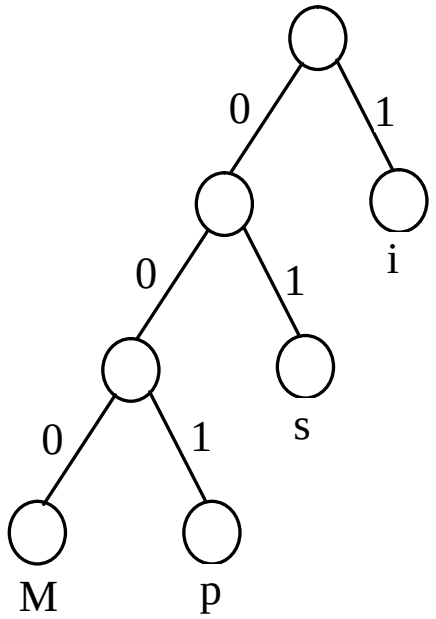
- What are binary trees?
- Explain Binary Search Tree
- Give a basic algorithm for searching a binary search tree.
- Differentiate linear from a nonlinear data structure.
- What are linear and non linear data Structures?

INTERVIEW PROBLEMS (JUST FOR PRACTICE!)

- Check if a given Binary Tree is BST or not?
- Height of Binary Tree
- Number of leaf nodes
- find the least common ancestor of two elements in a binary search tree? [Glassdoor.com: Microsoft interview question]
- Find the nth largest node in a binary search tree [Glassdoor.com: Microsoft interview question]

DATA COMPRESSION: HUFFMAN CODING

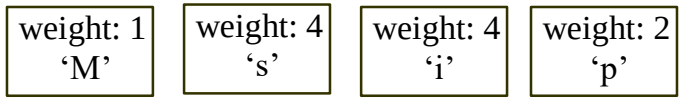
In ASCII, every character is encoded in 8 bits. Huffman coding compresses data by using fewer bits to encode more frequently occurring characters. The codes for characters are constructed based on the occurrence of characters in the text using a binary tree, called the *Huffman coding tree*.

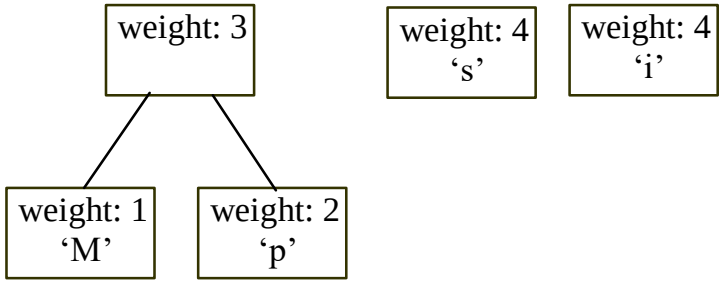


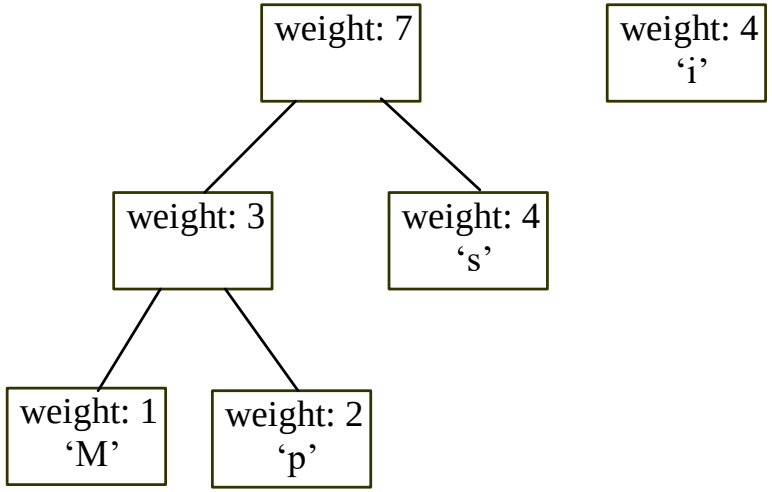
Character	Code	Frequency
M	000	1
p	001	2
s	01	4
i	1	4

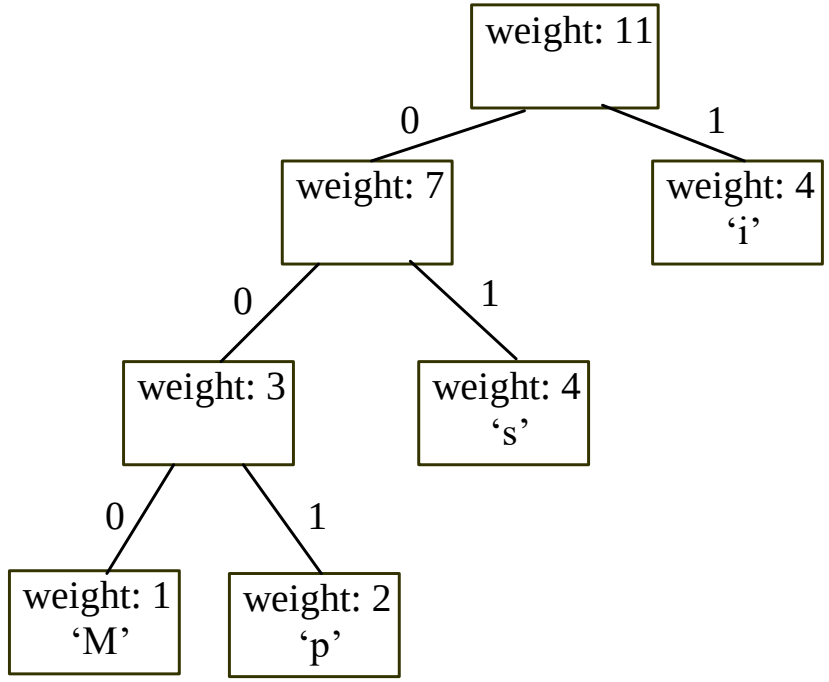
www.cs.armstrong.edu/liang/animation/HuffmanCodingAnimation.htm

CONSTRUCTING HUFFMAN TREE









THERE ARE VARIOUS WAYS TO IMPLEMENT TREES DATA STRUCTURES

- If time let's see other ways (maybe check some other book and talk about those implementations)

HOMEWORK FOR MODULE 4

DUE: see moodle, 11:59 pm

- Create a Tree class that implements the binary search tree (BST) methods shown below, and stores objects of class **Student**. You need to implement a **Student** class that stores name, major, and state of origin. The students should be sorted alphabetically by names in the tree:
  - [20 points] Insert (a new value into the tree while maintaining the BST structure):
    void insert(str studName, str major, str originState)
  - [20 points] Insert (a new value into the tree while maintaining the BST structure):
    void insert(Student newStudent)
  - [20 points] Delete (an existing value from the tree. If the value doesn't exist in the tree output a "Student not found" error message.
    void delete(str studName)
  - [10 points] Traversal: PrintInOrder, PrintPreOrder, PrintPostOrder
  - [10 points] Search (for a value in the tree): **bool search (str studName)**
  - [10 points] Height of Binary Tree: **int height()**
  - [10 points] Number of leaf nodes: **int numLeafNodes()**