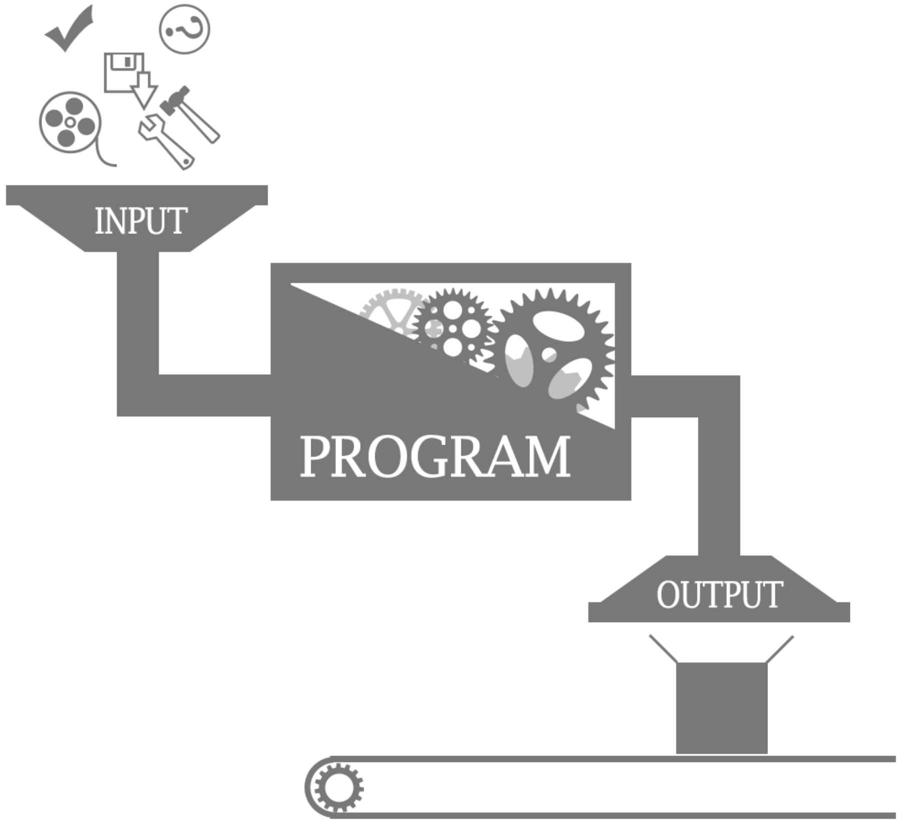
DSA\300 – Data Structures and Algorithms with C#

Data Structures & Algorithms with C# – Lecture #7

Ahmed Mohyeldin



Lecture #7

BINARY TREES

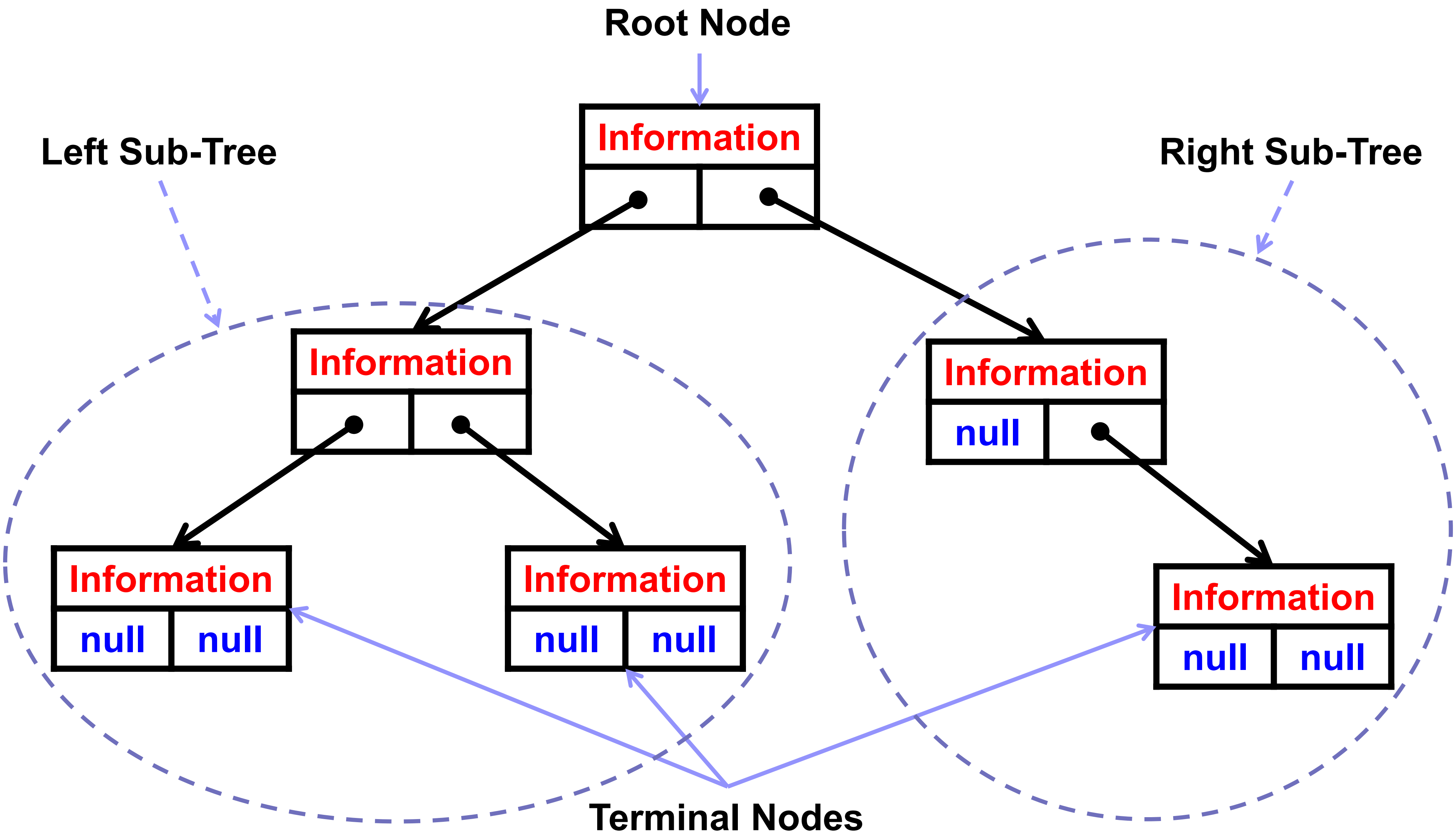
A Tree-Like Data Structure ☺

Binary Trees – The *What* ?

- A *Binary Tree* is a tree data structure in which each leaf (*i.e.,* node) has at most two children.
  - Typically the first node is known as the *Root Node* or *Parent Node*, and
  - The child nodes are called *Left Node* and *Right Node*.

- Although there can be many different types of Trees, *Binary Trees* are special because, when they are sorted, they lend themselves to rapid searches, insertions, and deletions.
  - Therefore, *Binary Trees* are commonly used to implement *Binary Search Trees* (BST) and *Binary Heaps* (BH).

Binary Trees – The *Structure* ?



A Sample Binary Tree of Height 3

Binary Trees –The *Terminology* ?

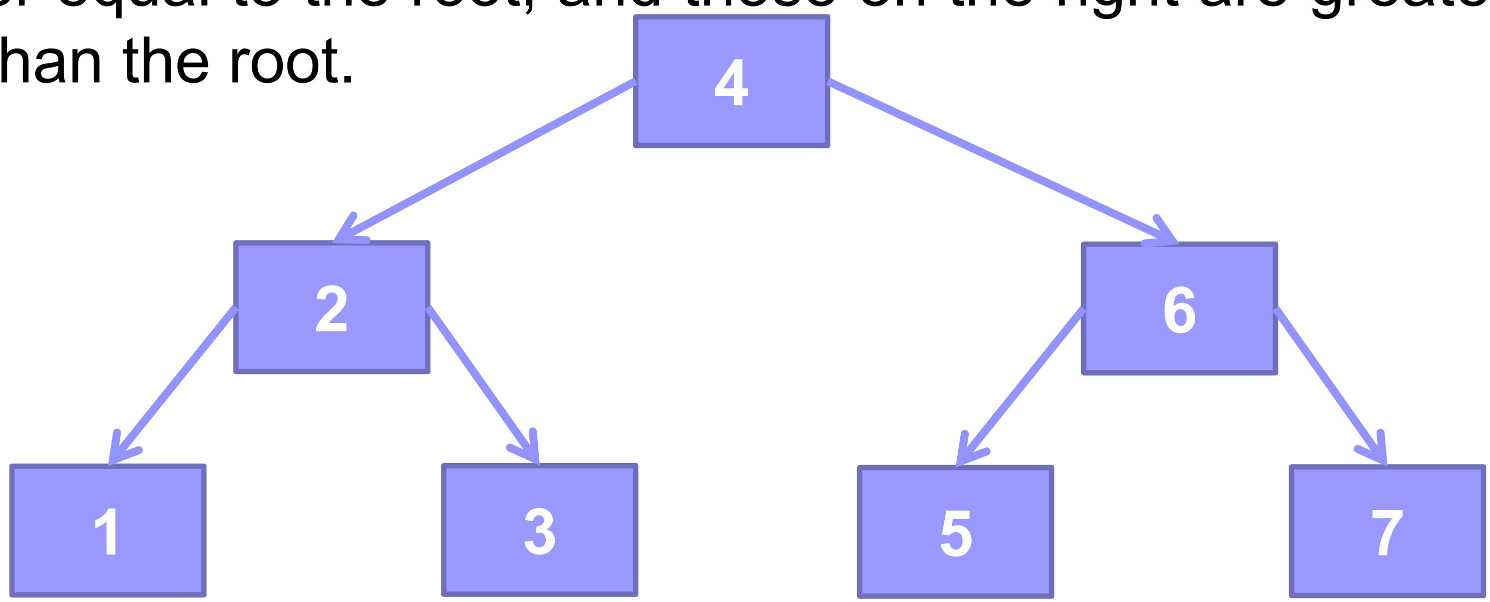
- The special terminology needed to discuss trees is a classic case of mixed metaphors:
  - *Node* (also called a *Leaf*) → is any data item in the tree.
  - *Root Node*(also called *Parent Node*) → is the first item in the tree.
  - *Subtree* → is any piece (*i.e.*, *Branch*) of thy tree.
  - *Terminal Node* → is a node that has no subtrees attached to it.
  - *Tree Height* → is equal to the number of layers deep that its root grows.

Binary Trees –The *Representation* ?

- Representation of Binary Trees
  - Logically, one can think of binary trees as appearing in memory as they do on paper, but remember that a tree is only a way to structure data in memory, which is linear in form.
  - The *Binary Tree* is a special form of *Linked List*
    - Items can be inserted, deleted, and accessed in any order.
    - Also, the retrieval operation is nondestructive.

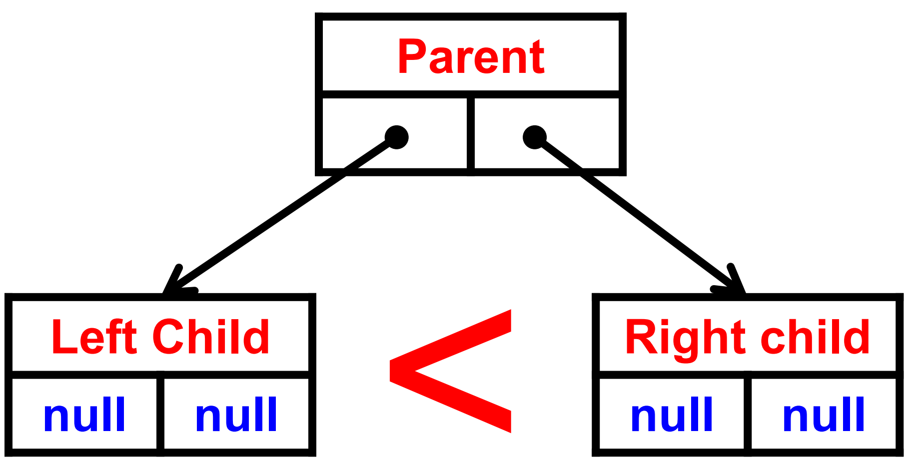
Binary Trees – The *Sorted Binary Trees* ?

- *Sorted Binary Tree* → *Binary Search Tree*
  - Although a tree need not be sorted, most uses require it. What constitutes a sorted binary tree depends on how it will be traversed. The rest of this discussion assumes *inorder traversing*.
  - Therefore, a *Sorted Binary Tree* is one where the subtree on the left contains nodes that are less than or equal to the root, and those on the right are greater than the root.



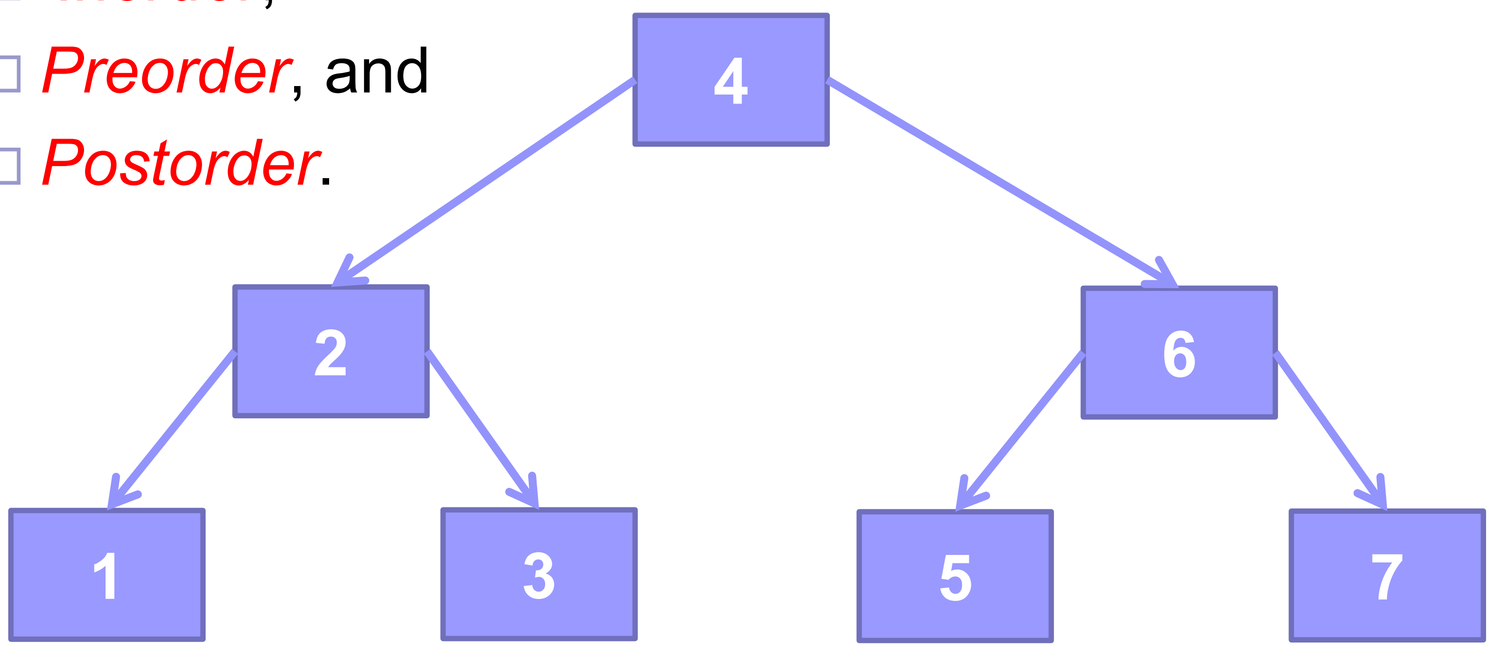
BSTs –The *Structure* (a formal definition)?

- A *Binary Search Tree* (*BST*) is a special kind of binary tree that exhibit the following property: for any node ***n***, every descendant node's value in the left subtree of ***n*** is less than the value of ***n***, and every descendant node's value in the right subtree is greater than the value of ***n***.
  - BSTs are designed this way to improve the efficiency of searching through the contents of a binary tree.



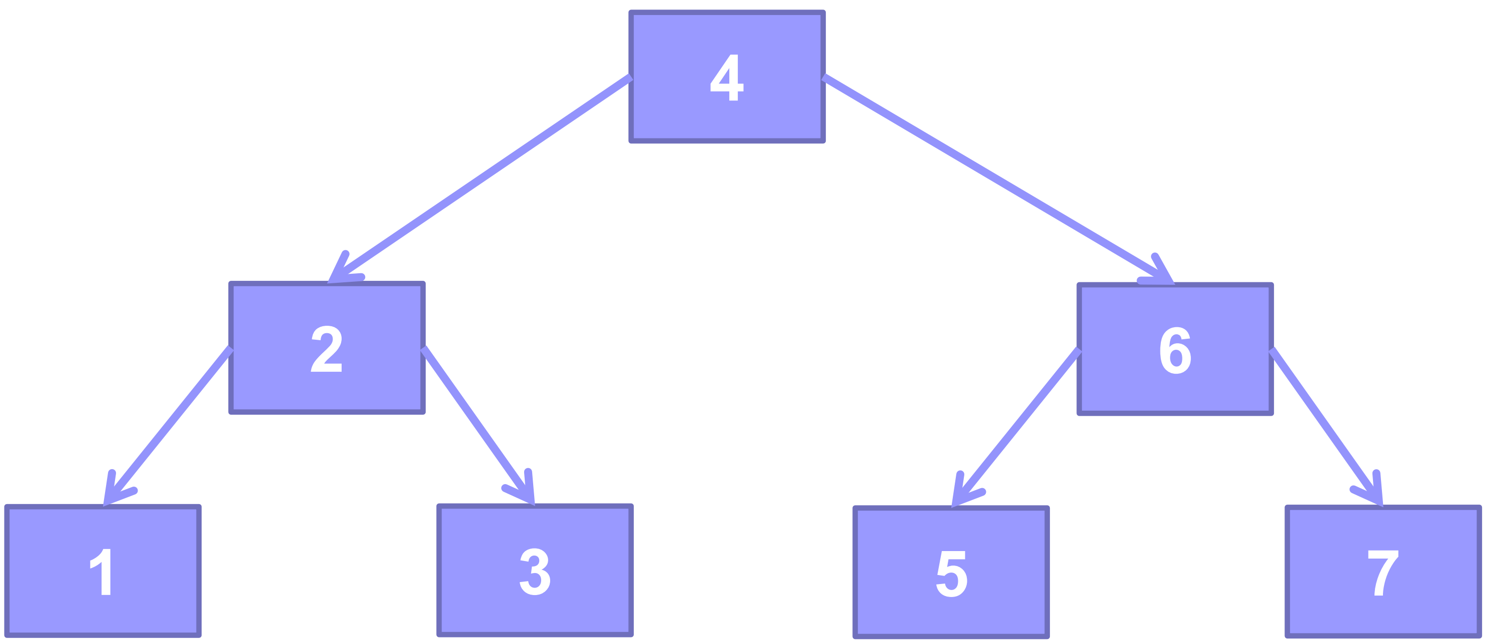
BSTs –The *Traversing* Methods?

- *Traversal of a Tree* is the process of accessing each node in the tree in specific sequence.
- How a tree is ordered depends on how it is going to be accessed. Generally, there are three ways to traverse a tree:
  - *Inorder*,
  - *Preorder*, and
  - *Postorder*.



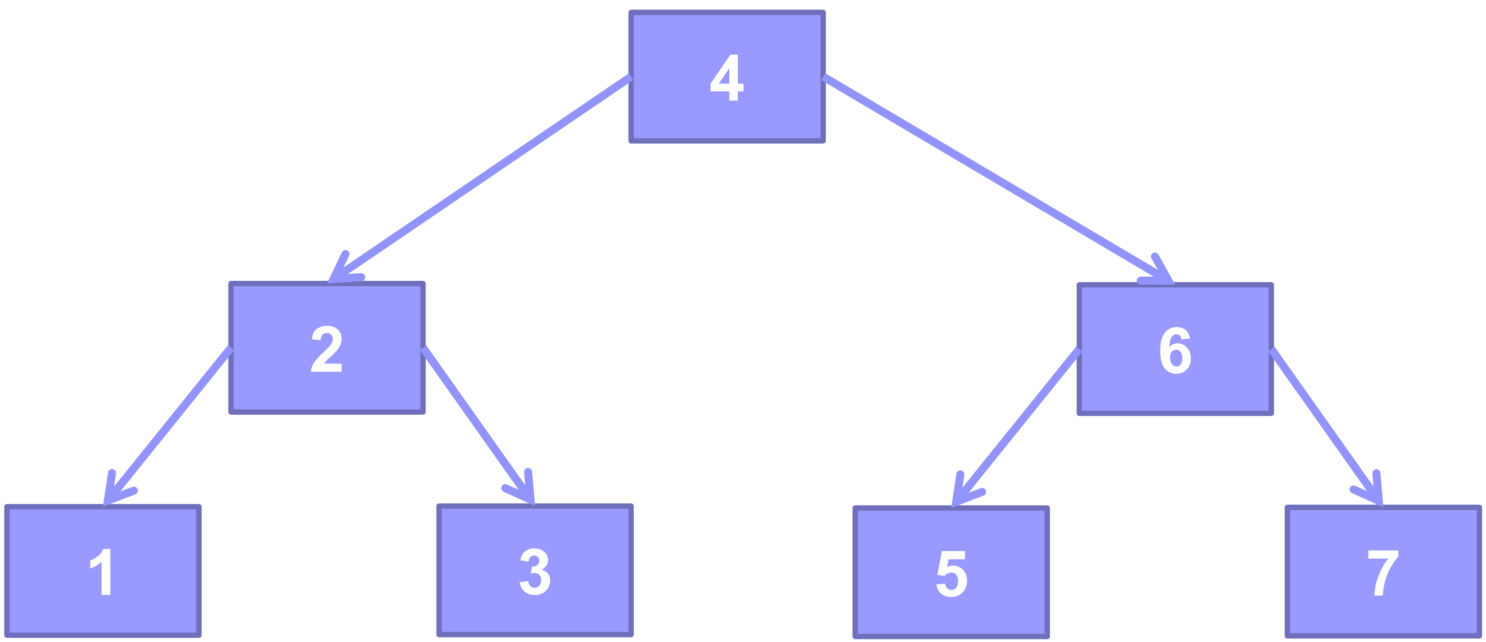
BSTs – The *Inorder Traversal* Method?

- To Traverse a BSTs using *Inorder Traversal*:
  - You visit the left subtree, the root, and then the right subtree.
- Example:
  - Inorder Accessing Sequence: 1 2 3 4 5 6 7



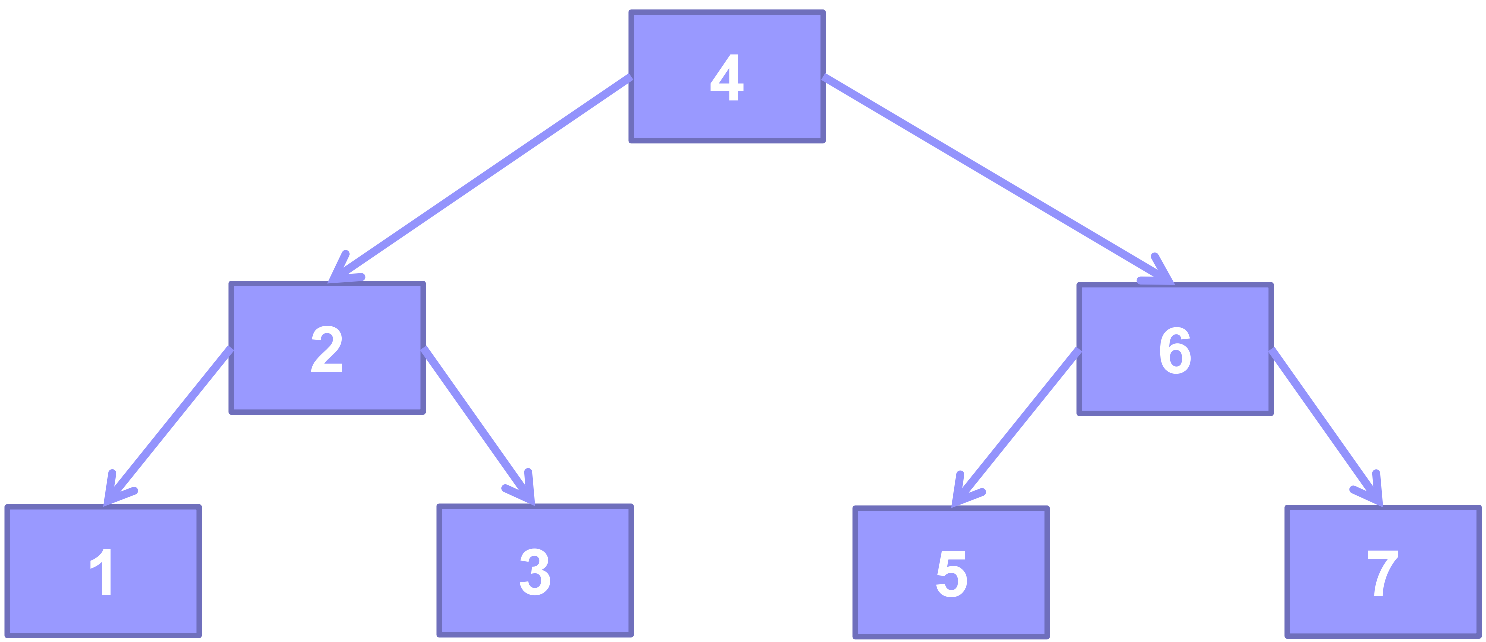
BSTs – The *Preorder Traversal* Method?

- To Traverse a BSTs using *Preorder Traversal*:
  - You visit the root, the left subtree and then the right subtree.
- Example:
  - Preorder Accessing Sequence: 4 2 1 3 6 5 7



BSTs – The *Postorder Traversal* Method?

- To Traverse a BSTs using *Postorder Traversal*:
  - You visit the left subtree, the right subtree, and then the root.
- Example:
  - Postorder Accessing Sequence: 1 3 2 5 7 6 4



BSTs – The *Operations* ?

- The primary operations that can be performed on a binary tree are:
  - Creating a Tree (*i.e.*, *Inserting* New Nodes)
    - *Case 1:* insert root node.
    - *Case 2:* insert a leaf node.
  - *Traversing* the Tree
    - *Method 1:* Inorder.
    - *Method 2:* Preorder.
    - *Method 3:* Postorder.
    - Deleting Nodes
      - *Case 1*: delete a root node.
      - *Case 2*: delete a leaf node with one subtree.
      - *Case 3*: delete a leaf node with two subtree.
    - Searching for a particular Node.

Binary Trees – The *Construction* ?

- The required steps for building a binary tree are:
  - Step #1 → Node element definition.
  - Step #2 → Initializing the empty tree root pointer.
  - Step #3 → Adding nodes (leaves).
  - Step #4 → Traversing the tree.
    - Step #4(a) → Traversing the tree inorder.
    - Step #4(b) → Traversing the tree preorder.
    - Step #4(c) → Traversing the tree postorder.
  - Step #5 → Searching for a node.
  - Step #6 → Deleting nodes.
  - Step #7 → Removing the entire tree.

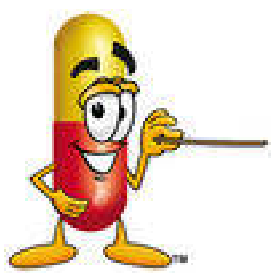
BTs – The *BinaryTree<T>* Class?

OOP Implementation

Basic object-oriented modelling and implementation of the "*Binary Tree* " data structure – The BinaryTree<T> and BinaryTreeNode<T> generic classes.

BTs – The *BinaryTreeNode<T>* Class?

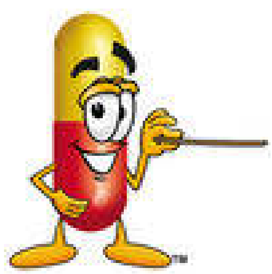
BinaryTreeNode<T>
Logical View
(Basic Version)



Value: T	
Left: BinaryTreeNode	Right: BinaryTreeNode

BTs – The BinaryTreeNode<T> Class?

BinaryTreeNode<T> Class Diagram (Basic Version)



BinaryTreeNode — T

- left: BinaryTreeNode<T> = null
- right: BinaryTreeNode<T> = null
- value: T = default(T)

\+ BinaryTreeNode( )
\+ BinaryTreeNode (value: T)
\+ BinaryTreeNode (value: T, left: BinaryTreeNode<T>, right: BinaryTreeNode<T>)

\+ ~BinaryTreeNode ( )

\+ Left: BinaryTreeNode<T> {READWRITE}
\+ Right: BinaryTreeNode<T> {READWRITE}
\+ Value: T {READWRITE}

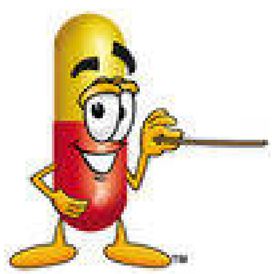
BTs – The *BinaryTreeNode<T>* Class? *(…cont'd)*

```
// Example: Binary Tree Data Structure - BinaryTreeNode<T>.
using System;

namespace Mohyeldin.DSA
{
  public sealed class BinaryTreeNode<T>
  {
      public BinaryTreeNode( )
      { // TODO: ... }
      public BinaryTreeNode(T value)
      { // TODO: ... }
      internal BinaryTreeNode(T value, BinaryTreeNode<T> left,
                                       BinaryTreeNode<T> right)
      { // TODO: ... }

      public T Value { get; set; } = default(T);

      public BinaryTreeNode<T> Left { get; internal set; } = null;

      public BinaryTreeNode<T> Right {get; internal set;} = null;
      // TODO: Any other enhancements may be added here...
    }
}   /* (^_^) The BT Node Class Definition - Basic Version. (^_^) */
```

Value: T	
Left: BinaryTreeNode	**Right: BinaryTreeNode**

BTs – The *BinaryTree<T>* Class?

BinaryTree<T> Class Diagram (Basic Version)



T
BinaryTree
- root: BinaryTreeNode<T> = null
+ BinaryTree ( ) **+ ~BinaryTree ( )**
+ Clear (value: T) : void **+ <u>InorderTraversal (current: BinaryTreeNode<T>) : void</u>** **+ <u>PreorderTraversal (current: BinaryTreeNode<T>) : void</u>** **+ <u>PostorderTraversal (current: BinaryTreeNode<T>) : void</u>**
+ Root: BinaryTreeNode<T> {READWRITE}

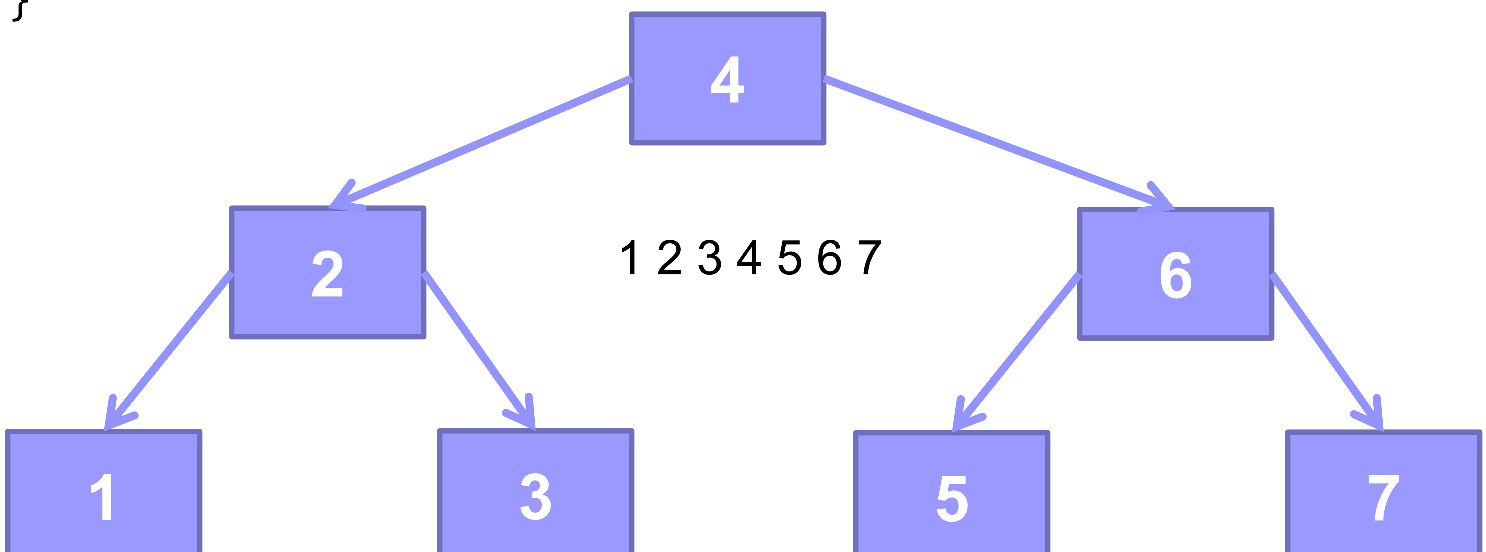
BTs – The *BinaryTree<T>* Class? *(…cont'd)*

```
// Example: Binary Tree Data Structure - BinaryTree<T>.
using System;

namespace Mohyeldin.DSA
{
  public class BinaryTree<T>
  {
    public BinaryTree( ) { Root = null; }

    public virtual void Clear() { Root = null; }

    public static void InorderTraversal(BinaryTreeNode<T> current)
    { // TODO: ... }
    public static void PreorderTraversal(BinaryTreeNode<T> current)
    { // TODO: ... }
    public static void PostorderTraversal(BinaryTreeNode<T> current)
    { // TODO: ... }

    public BinaryTreeNode<T> Root {get; internal set;} = null;

      // TODO: Any other enhancements may be added here...
    }
}/* (^_^) The Binary Tree Class Definition - Basic Version. (^_^) */
```

Value: T	
Left: BinaryTreeNode	**Right:** BinaryTreeNode

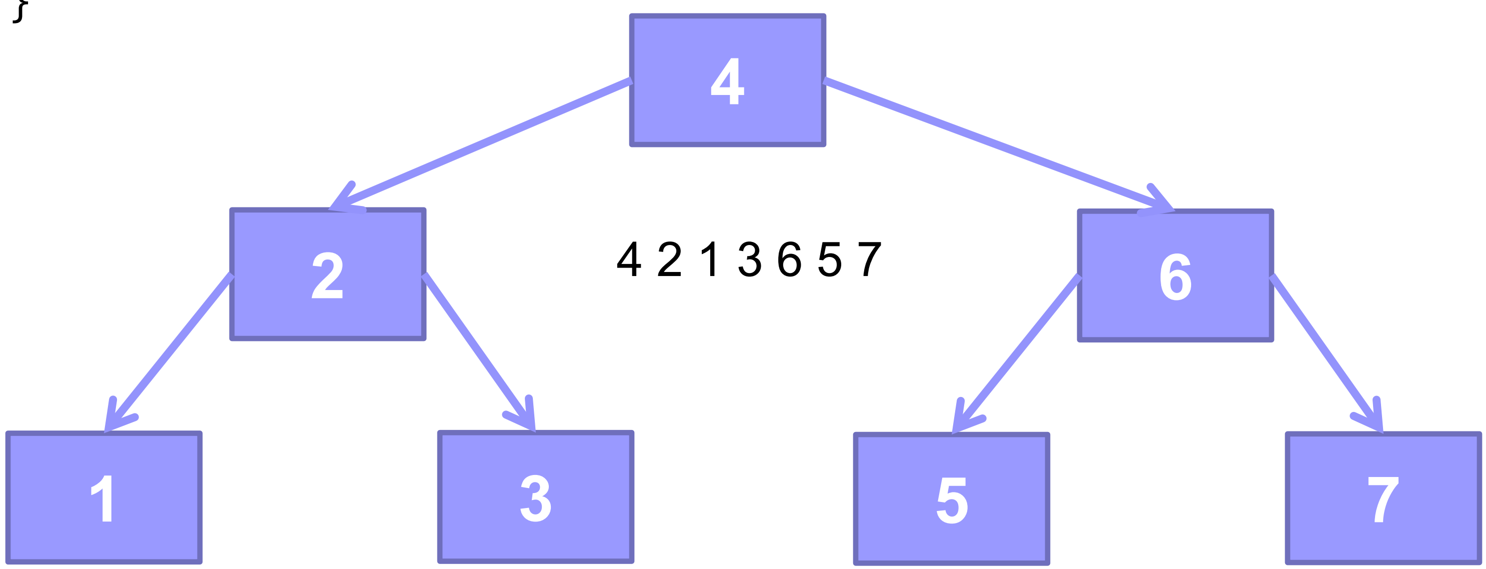
BTs – The *BinaryTree<T>* Class? *(…cont'd)*

```
// InorderTraversal( ) - Performs an inorder traversal of a binary
   tree, given a "BinaryTreeNode<T>" as the root node.
public static void InorderTraversal(BinaryTreeNode<T> current)
{
    if (current != null)
    {
        // Visit the left child... (a recursive process!).
        InorderTraversal(current.Left);
        // Visit the node (Output the value of the current node).
        Console.WriteLine(current.Value);
        // Visit the right child... (a recursive process!).
        InorderTraversal(current.Right);
    }
}
```



BTs – The *BinaryTree<T>* Class? *(…cont'd)*

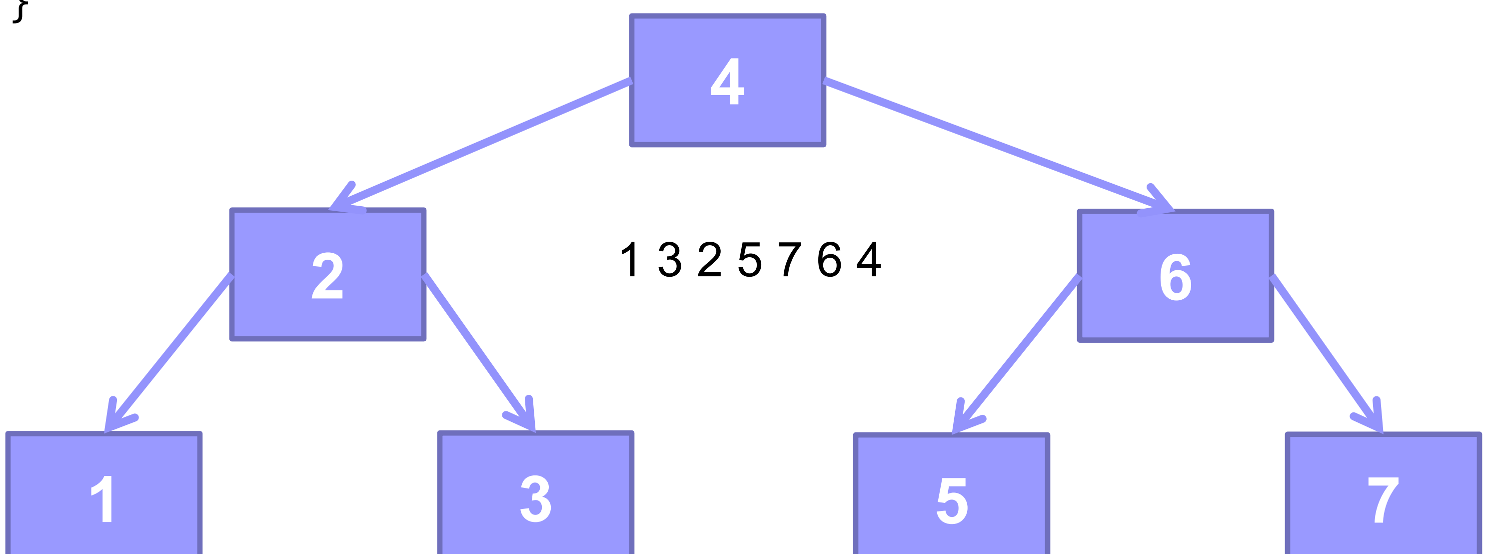
```
// PreorderTraversal( ) - Performs a preorder traversal of a binary
   tree, given a "BinaryTreeNode<T>" as the root node.
public static void PreorderTraversal(BinaryTreeNode<T> current)
{
    if (current != null)
    {
        // Output the value of the current node.
        Console.WriteLine(current.Value);

        // Recursively print the left and right children.
        PreorderTraversal(current.Left);
        PreorderTraversal(current.Right);
    }
}
```



BTs – The *BinaryTree<T>* Class?

(…cont'd)

```
// PostorderTraversal( )- Performs a postorder traversal of a binary
   tree, given a "BinaryTreeNode<T>" as the root node.
public static void PostorderTraversal(BinaryTreeNode<T> current)
{
    if (current != null)
    {
        // Visit the left child... (a recursive process!).
        PostorderTraversal(current.Left);
        // Visit the right child... (a recursive process!).
        PostorderTraversal(current.Right);
        // Visit the node (Output the value of the current node).
        Console.WriteLine(current.Value);
    }
}
```



BTs – The *BinaryTree<T>* Class? *(…cont'd)*

```
// Binary Tree Data Structure - BinaryTree<T> Class.

?

// TODO: Write a C#  generic classes that implements the BT data
// structure ("BinaryTree<T>" and "BinaryTreeNode<T>") UML class
// diagrams with a complete "Test Driver" to demonstrate using the
// different binary tree traversal( ) methods.
```

DEMO

Object-Oriented Implementation of Binary-Trees in C# – The BinaryTree<T> Generic Class.

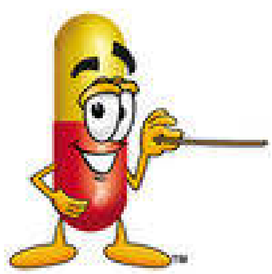
BSTs – The *BinarySearchTree<T>* Class?

OOP Implementation

Basic object-oriented modelling and implementation of the "*Binary Search Tree* " data structure – The BinarySearchTree<T> and BinaryTreeNode<T> generic classes.

BSTs – The *BinarySearchTree<T>* Class?

BinarySearchTree<T> Class Diagram (Basic Version)



T

BinarySearchTree

- root: BinaryTreeNode<T> = null
- count: int = 0
- comparer: IComparer<T> = Comparer<T>.Default.

+ BinarySearchTree ( )
+ BinarySearchTree (comparer: IComparer<T>)

+ ~BinarySearchTree ( )

+ Add (value: T) : BinaryTreeNode<T>

+ Remove (value: T) : bool
+ Clear () : void

+ Contains (value: T) : bool
+ Find (value: T) : BinaryTreeNode<T>

+ InorderTraversal () : void
+ PreorderTraversal () : void
+ PostorderTraversal () : void

+ Count: int {READONLY}

BSTs – The *BinarySearchTree<T>* Class? *(…cont’d)*

```csharp
// Example: Binary Tree Data Structure - BinarySearchTree<T>.
using System;

namespace Mohyeldin.DSA
{
  public class BinarySearchTree<T>
  {
    public BinarySearchTree( ) {// TODO: ... }
    public BinarySearchTree(IComparer<T> comparer) {// TODO: ... }

    public BinaryTreeNode<T> Add(T value) {// TODO: ... }
    public bool Remove(T value) {// TODO: ... }
    public virtual void Clear() {// TODO: ... }
    public bool Contains(T value) { // TODO: ... }
    public BinaryTreeNode<T> Find(T value) { // TODO: ... }

    public void InorderTraversal() { // TODO: ... }
    public void PreorderTraversal() { // TODO: ... }
    public void PostorderTraversal() { // TODO: ... }
    public int Count {get; private set;} = 0;
    // TODO: Any other enhancements may be added here...
  }
} /* (^_^) The BST Class Definition - Basic Version. (^_^) */

```

Value: T	
Left: BinaryTreeNode	Right: BinaryTreeNode

BSTs – The *Add(T value) Method* (1/3)?

(…cont'd)

Iterative Implementation

```
// Add( ) - Adds a new node containing the specified value to the
//          BinarySearchTree<T> --- Iterative implementation(1/3).
public virtual BinaryTreeNode<T> Add(T value)
{
    // Create a new node instance.
    BinaryTreeNode<T> node = new BinaryTreeNode<T>(value);
    int result;

    // Now, insert node into the tree
    // trace down the tree until we hit a NULL
    BinaryTreeNode<T> current = root, parent = null;
    while (current != null)
    {
        result = comparer.Compare(current.Value, value);
        if (result == 0)
            // They are equal - attempting to enter a duplicate - do
            //                  nothing.
            return null;
        else if (result > 0)
        {
            // Current.Value > value, must add node to current's
            // left subtree.
                                                   // continued...
```

BSTs – The *Add(T value) Method* (2/3)? *(…cont'd)*

```
// Add( ) - Adds a new node containing the specified value to the
//            BinarySearchTree<T> --- Iterative implementation(2/3).

            parent = current;
            current = current.Left;
        }
        else if (result < 0)
        {
            // Current.Value < value, must add node to current's
            // right subtree.
            parent = current;
            current = current.Right;
        }
    }

    // We're ready to add the node!
    count++;
    if (parent == null)
    {
        // The tree was empty, make node the root.
        root = node;
    }
                                                  // continued...
```

BSTs – The *Add(T value)* Method (3/3)? *(…cont'd)*

```
// Add( ) - Adds a new node containing the specified value to the
//           BinarySearchTree<T> --- Iterative implementation(3/3).
    else
    {
        result = comparer.Compare(parent.Value, value);
        if (result > 0)
            // parent.Value > value, therefore node must be added to
            // the left subtree.
            parent.Left = node;
        else
            // parent.Value < value, therefore node must be added to
            // the right subtree.
            parent.Right = node;
    }

    return node; // Return the node containing the value.
}

/*(^_^) The BinarySearchTree.Add(T value) operation is done (^_^).*/
```

BSTs – The *Add(T value) Recursive* (1/2)? *(...cont'd)*

```
// Add( ) - Adds a new node containing the specified value to the
//             BinarySearchTree<T> --- Non-recursive public wrapper.
public virtual BinaryTreeNode<T> Add(T value)
{
    return root = Add(root, value);
}

// Add( ) - Adds a new node containing the specified value to the
//             BinarySearchTree<T> --- Recursive implementation(1/2).
private BinaryTreeNode<T> Add(BinaryTreeNode<T> node, T value)
{
    if (node == null)
    {
        node = new BinaryTreeNode<T>(value);
        count++;
    }
    else
    {
        int result = comparer.Compare(node.Value, value);
        if (result > 0)
        {
            node.Left = Add(node.Left, value);
        }                                                // continued...
```

Recursive Implementation

BSTs – The *Add(T value) Recursive* (2/2)? *(…cont'd)*

```csharp
// Add( ) - Adds a new node containing the specified value to the
//           BinarySearchTree<T> --- Recursive implementation(2/2).

        else if (result < 0)
        {
            node.Right = Add(node.Right, value);
        }

        else
        {
            throw new ArgumentException("Duplicate node.",
                                         nameof(value));
        }
    }

    return node;
}

/*(^_^) The BinarySearchTree.Add(T value) operation is done (^_^).*/
```

BSTs – The *BinarySearchTree<T>* Class? *(…cont'd)*

```
// Binary Search Tree Data Structure - BinarySearchTree<T> Class.
```

?

```
// TODO: Write a C#  generic classes that implements the BST data
// structure (BinarySearchTree<T> and BinaryTreeNode<T>) UML class
// diagrams with a complete "Test Driver" to demonstrate using the
// different binary search tree traversal( ) methods.
```

DEMO

Object-Oriented Implementation of Binary-Search Trees in C# – The BinarySearchTree<T> Generic Class.

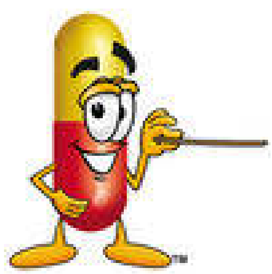
BSTs – The *BinarySearchTree<T>* Class?

OOP Implementation

Enhanced object-oriented modelling and implementation of the "*Binary Search Tree* " data structure – The BinarySearchTree<T> and BinaryTreeNode<T> generic classes.

BSTs – The *BinarySearchTree<T>* Class?

BinarySearchTree<T> Class Diagram (Enhanced Version)



BinarySearchTree	T
- root: BinaryTreeNode<T> = null - count: int = 0 - comparer: IComparer<T> = Comparer<T>.Default.	
+ BinarySearchTree ( ) + BinarySearchTree (comparer: IComparer<T>) + ~BinarySearchTree ( )	
+ Add (value: T) : BinaryTreeNode<T> + Remove (value: T) : bool + Clear () : void + Contains (value: T) : bool + Find (value: T) : BinaryTreeNode<T> + InorderTraversal () : void + PreorderTraversal () : void + PostorderTraversal () : void + ToString () : string + ToString (traversalMethod:TraversalMethod) : string	
+ Count: int {READONLY} + Inorder: IEnumerable<T> {READONLY} + Preorder: IEnumerable<T> {READONLY} + Postorder: IEnumerable<T> {READONLY}	

BSTs – The *BinarySearchTree<T>* (1/2)? *(...cont'd)*

```
// Example: Binary Tree Data Structure - BinarySearchTree<T> (1/2).
using System;

namespace Mohyeldin.DSA
{
  public sealed class BinarySearchTree<T>
          : ICollection<T>, IEnumerable<T>
  {
    public BinarySearchTree( ) {// TODO: ... }
    public BinarySearchTree(IComparer<T> comparer) {// TODO: ... }

    public BinaryTreeNode<T> Add(T value) {// TODO: ... }
    public bool Remove(T value) {// TODO: ... }
    public virtual void Clear() {// TODO: ... }

    public bool Contains(T value) { // TODO: ... }
    public BinaryTreeNode<T> Find(T value) { // TODO: ... }

    public void InorderTraversal() { // TODO: ... }
    public void PreorderTraversal() { // TODO: ... }
    public void PostorderTraversal() { // TODO: ... }

                                                    // continued...
```

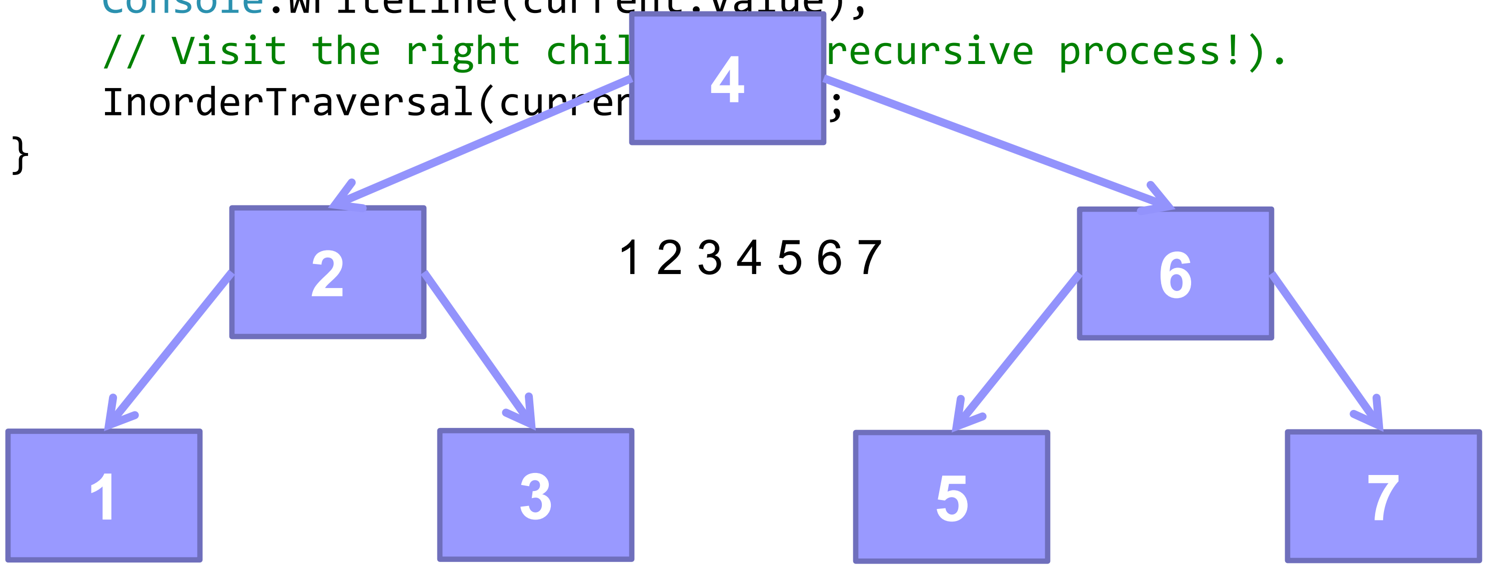
Value: T	
Left: BinaryTreeNode	**Right:** BinaryTreeNode

BST – The BinarySearchTree<T> (2/2)? *(...cont'd)*

```
// Example: Binary Tree Data Structure - BinarySearchTree<T> (2/2).
    public override string ToString() { // TODO: ... }
    public string ToString(TraversalMethod traversalMethod)
    { // TODO: ... }

    public virtual IEnumerator<T> GetEnumerator() { // TODO: ... }
    public virtual IEnumerator<T> GetEnumerator(TraversalMethod
                                 TraversalMethod) { // TODO: ... }
    IEnumerator IEnumerable.GetEnumerator() { // TODO: ... }

    public int Count {get; private set;} = 0;

    public IEnumerable<T> Inorder {get { // TODO: ... }}
    public IEnumerable<T> Preorder {get { // TODO: ... }}
    public IEnumerable<T> Postorder {get { // TODO: ... }}

      // TODO: Any other enhancements may be added here...
  }
}  /* (^_^) The BST Class Definition – Enhanced Version. (^_^) */
```

BTs – The *BinaryTree<T>* Class? (Note1) *(...cont'd)*

```
// InorderTraversal( ) - Performs an inorder traversal of a binary
   tree, given a "BinaryTreeNode<T>" as the root node.
public void InorderTraversal() {// Call: bTree.InorderTraversal();
     InorderTraversal(root);
}
private static void InorderTraversal(BinaryTreeNode<T> current)
{
     if (current != null)
     {
          // Visit the left child... (a recursive process!).
          InorderTraversal(current.Left);
          // Visit the node (Output the value of the current node).
          Console.WriteLine(current.Value);
          // Visit the right chil[illegible]recursive process!).
          InorderTraversal(curren[illegible]);
     }
}
```



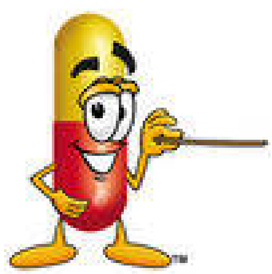
Nodes – The *Node<T>* Classes Hierarchy?

OOP Implementation

Enhanced object-oriented modelling and implementation of the "*Node* " class to fit the needs of all the *Node-Based Data Structures* , *e.g.*, *Trees*, *Graphs*, *Linked-Lists*, *etc.*

Nodes – The *Node<T>* Classes Hierarchy?

Nodes Hierarchy Class Diagram (Basic Version)



Node (T)

- neighbors: NodeList<T> = null
- value: T = default(T)

...

+ Neighbors: NodeList<T> {READWRITE}
+ Value: T {READWRITE}

...

BinaryTreeNode (T)

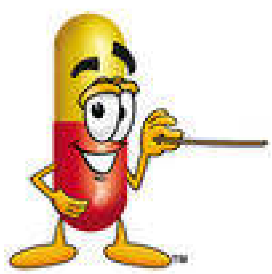
...

+ Left: BinaryTreeNode<T> {READWRITE} = base.Neighbors[0]
+ Right: BinaryTreeNode<T> {READWRITE} = base.Neighbors[1]

...

Nodes – The *Node<T>* Base Class?

Node<T>
Base Class
Diagram
(Basic Version)



Node (T)
- neighbors: NodeList<T> = null - value: T = default(T)
+ Node( ) + Node (value: T) + Node (value: T, neighbors: NodeList<T>) + ~Node ( )
+ Neighbors: NodeList<T> {READWRITE} + Value: T {READWRITE}

Nodes – The *Node<T>* Base Class? *(…cont'd)*

```
// Example: Node-Based Data Structures - Node<T>.
using System;


namespace Mohyeldin.DSA
{
    public class Node<T>
    {
        public Node () : base() { }

        public Node(T value) : this(value, null) { }

        public Node(T value, NodeList<T> neighbors)
        {
            this.Value = value;
            this.Neighbors = neighbors;
        }

        public T Value { get; set; } = default(T);

        protected NodeList<T> Neighbors { get; set; } = default(T);
    }
} /* (^_^) The Base Node Class Definition - Basic Version. (^_^) */
```

Value: T
Neighbors: NodeList

Nodes – The *NodeList<T>* Collection?

NodeList<T>
Collection Class Diagram
(Basic Version)

T

Collection : System.Object

- \- count: int
- \- items: T[ ]

- \+ Collection( )
- \+ Collection (list: Ilist<T>)
- \+ ~Collection ( )

- \+ Add(value: T) void
- ...

- \+ Count: int {READONLY}
- \+ Item[int index]: T {READWRITE}
- \+ Items: Ilist<T> {READONLY}

T

NodeList : Collection<Node<T>>

- \+ NodeList( )
- \+ NodeList (initialSize: int)
- \+ ~Node ( )

- \+ Find (value: T) Node<T>

- ...

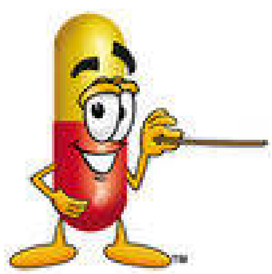
Nodes – The *NodeList<T>* Collection? *(...cont'd)*

```
// Example: Node-Based Data Structures - NodeList<T>.
using System;
using System.Collections.ObjectModel;

namespace Mohyeldin.DSA
{
    public class NodeList<T> : Collection<Node<T>>
    {
        public NodeList() : base() { }
        public NodeList(int initialSize)
        {   // Add the specified number of items.
            for (int i = 0; i < initialSize; i++)
                base.Items.Add(default(Node<T>));
        }
        public Node<T> FindByValue(T value)
        {   // Search the list for the value.
            foreach (Node<T> node in Items)
                if (node.Value.Equals(value))
                    return node; // Match is found; return the node.
            return null;       // No match is found; return null.
        }
    }
} /* (^_^) The NodeList Class Definition - Basic Version. (^_^) */

```

Collection.Items: Node<T>[ ]

Nodes – The BinaryTreeNode<T> Class?

BinaryTreeNode<T>
Class
Diagram
(Basic Version)



T

Node

- neighbors: NodeList<T> = null
- value: T = default(T)

+ Node( )
+ Node (value: T)
+ Node (value: T, neighbors: NodeList<T>)
+ ~Node ( )

+ Neighbors: NodeList<T> {READWRITE}
+ Value: T {READWRITE}

T

BinaryTreeNode : Node<T>

...

+ BinaryTreeNode( )
+ BinaryTreeNode (value: T)
+ BinaryTreeNode (value: T, left: BinaryTreeNode<T>, right: BinaryTreeNode<T>)
+ ~BinaryTreeNode ( )

+ Left: BinaryTreeNode<T> {READWRITE} = base.Neighbors[0]
+ Right: BinaryTreeNode<T> {READWRITE} = base.Neighbors[1]

Nodes – The *BinaryTreeNode<T>* ? (1/3) *(…cont'd)*

Value: T	
Left: BinaryTreeNode	Right: BinaryTreeNode

```
// Example: Node-Based Data Structures - BinaryTreeNode<T> (1/3).
using System;


namespace Mohyeldin.DSA
{
    public sealed class BinaryTreeNode<T> : Node<T>
    {
        public BinaryTreeNode() : base()
        { // TODO: ... }

        public BinaryTreeNode(T value) : base(value, null)
        {// TODO: ... }

        public BinaryTreeNode(T value, BinaryTreeNode<T> left,
                                       BinaryTreeNode<T> right)
        {
            base.Value = value;
            NodeList<T> children = new NodeList<T>(2);
            children[0] = left;
            children[1] = right;
            base.Neighbors = children;
        }                                              // continued...
```

Nodes – The *BinaryTreeNode<T>* ? (2/3) *(…cont'd)*

```
// Example: Node-Based Data Structures - BinaryTreeNode<T> (2/3).
        public BinaryTreeNode<T> Left
        {
            get
            {
                if (base.Neighbors == null)
                    return null;
                else
                    return (BinaryTreeNode<T>) base.Neighbors[0];
            }
            set
            {
                if (base.Neighbors == null)
                    base.Neighbors = new NodeList<T>(2);

                base.Neighbors[0] = value;
            }
        }

         // TODO: Any other enhancements may be added here...
    }
}
                                                       // continued...

```

Nodes – The *BinaryTreeNode<T>* ? (3/3) *(…cont'd)*

```
// Example: Node-Based Data Structures - BinaryTreeNode<T> (3/3).
        public BinaryTreeNode<T> Right
        {
            get
            {
                if (base.Neighbors == null)
                    return null;
                else
                    return (BinaryTreeNode<T>)base.Neighbors[1];
            }
            set
            {
                if (base.Neighbors == null)
                    base.Neighbors = new NodeList<T>(2);

                base.Neighbors[1] = value;
            }
        }

      // TODO: Any other enhancements may be added here...
    }
}
/* (^_^)The BinaryTreeNode Class Definition - Basic Version. (^_^)*/
```

DEMO

Object-Oriented Implementation of Binary-Trees and Binary-Search Trees in C# – Using The Node<T> Generic Class.

BSTs – The *Time* Analysis?

- *Examining the efficiency of common operations, on a binary search tree consisting of n nodes*?

BSTs – The *Performance* Summary?

- *Performance Analysis:*
  - *Binary trees* offer tremendous power. flexibility, and efficiency when used with database management programs because the information for these data-bases must reside on disk and because access times are important Because a balanced *binary tree has as a worst cases, $\log_2 n$ comparisons in searching it performs for better than a linked list, which must rely on a sequential search.*
- *Time Analysis*:
  - Worst Case *O(n) for* Indexing, Search, Insertion & Deletion.
  - Average Case for Indexing $O(\log_2 n)$, Search $O(\log_2 n)$, Insertion $O(\log_2 n)$, and Deletion $O(\log_2 n)$
    - Where, *n* is the number of items being sorted.

Linked Lists – *Lab Assignments*

Lab Assignments

Linked Lists – *Lab Assignments*

- Structured Programming Implementation:

    1. Implement the operations of the ***BST*** data structure using a group of static class methods. The nodes can be implemented as a POD structure that has a generic information field of type ***T***, and two link fields to represent the left and right children of the node. Finally, write an appropriate test driver for each of the tree operations.

- Object-Oriented Programming Implementation:

    2. Create a ***BinarySearchTree<T>*** and ***BinaryTreeNode<T>*** generic classes that implement the ***BST*** data structure with an appropriate test driver. The nodes can hold items of any specified data type ***T*** that implements the ***IComparable<T>*** *interface*.
        - ***Bonus:*** Create a ***Node<T>*** generic class that represents the base concept of a node for a linked list, tree or graph; a node that contains a data item and has an arbitrary number of neighbors. Then, use it to derive the ***BinaryTreeNode<T>*** generic class that represents a node in a binary tree. Finally, rework the previous assignment using these node classes.

SUMMARY – Binary Trees

- Tree → is a data structure that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of linked nodes.
- Binary Tree (*BT*) → is a tree data structure in which each leaf (*i.e.,* node) has at most two children.
- *Binary Search Tree* (*BST*) → is a special kind of binary tree that exhibit the following property: for any node ***n***, every descendant node's value in the left subtree of ***n*** is less than the value of ***n***, and every descendant node's value in the right subtree is greater than the value of ***n***.
- Time Analysis → Binary Search Trees
  - Insertion/Deletion → O(n) worst case, O($\log_2$n) average case.
  - Traversal → O(n) worst case, O($\log_2$n) average case.
  - Searching → O(n) worst case, O($\log_2$n) average case.
- Space Analysis → Binary Search Trees
  - Wasted Space → O(n) worst case, O(n) average case.

Please consider the environment before printing this material.

Now, Let's go to the DSA programming Lab ☺

Lecture #7: Binary Trees

A Tree-Like Data Structure ☺

SUMMARY – Q & A

Feedbacks: email to ameldin@gmail.com. We value your feedback!
(*Please include the following prefix in the subject field:* [DSA-C#])