CPE 212 - Fundamentals of Software Engineering

Binary Trees

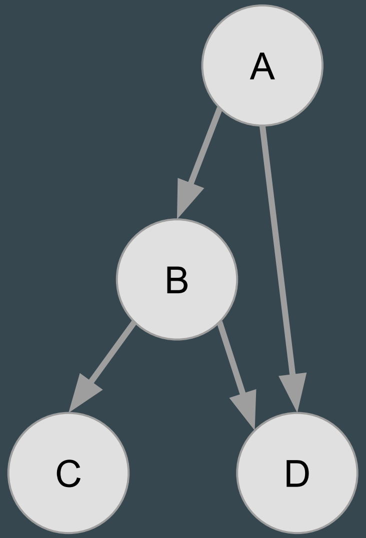
Outline

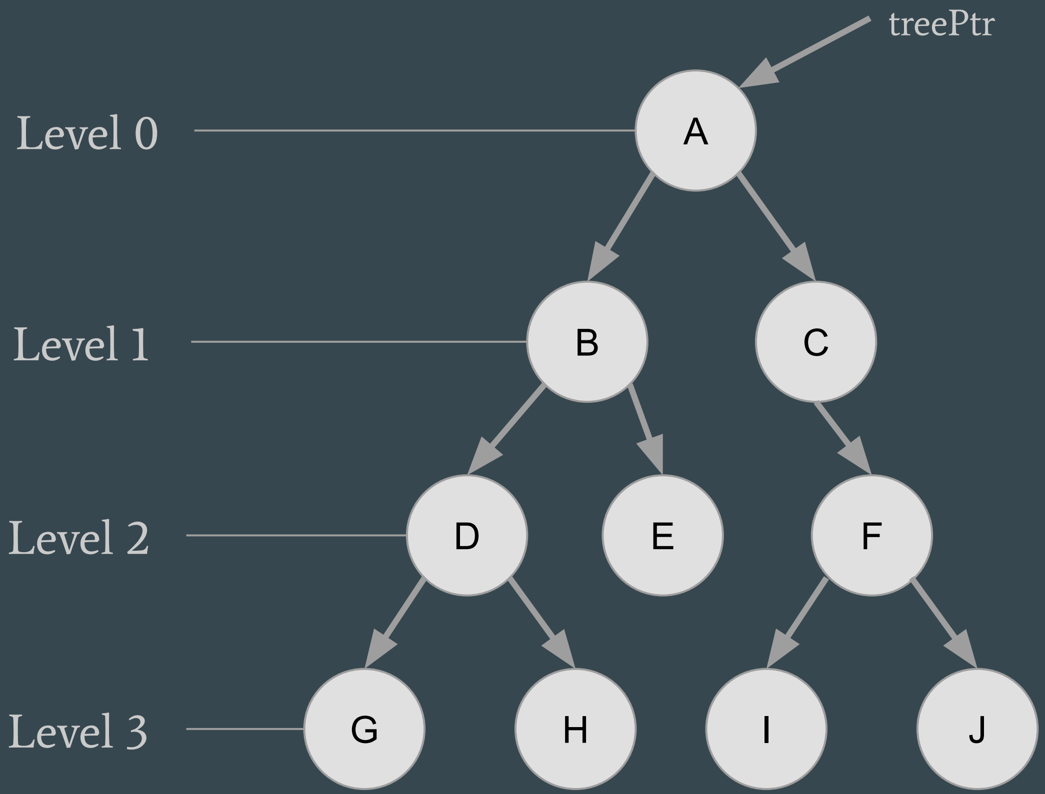
- Tree definition
- Tree examples
- Binary Search Tree Definition
- Example
- Copy Constructors
- Tree Traversal

Tree

- Binary Tree
  - A structure with a unique starting node (root), in which each node is capable of having two child nodes, and in which a unique path exists from the root to every other node
- Root
  - The top node of a tree structure
- Leaf Node
  - A tree node that has no children

Tree Example





Real-World Examples of Trees?

Real-World Tree Examples

- **C++ Class Inheritance Relationship**
  - Assuming no multiple inheritance
- UNIX File System
  - Root directory is called /
  - Root contains other directories and files
- Arithmetic Expressions
- Book

Binary Tree Concepts

- Level
  - The distance of a node from the root; the root is at level 0
- Height
  - The maximum level in a tree
- Maximum number of nodes at Level N is 2^N
  - Level 0: contains no more than $2^0 = 1$ node
  - Level 1: contains no more than $2^1 = 2$ nodes
  - Level 2: contains no more than $2^2 = 4$ nodes
  - Etc.

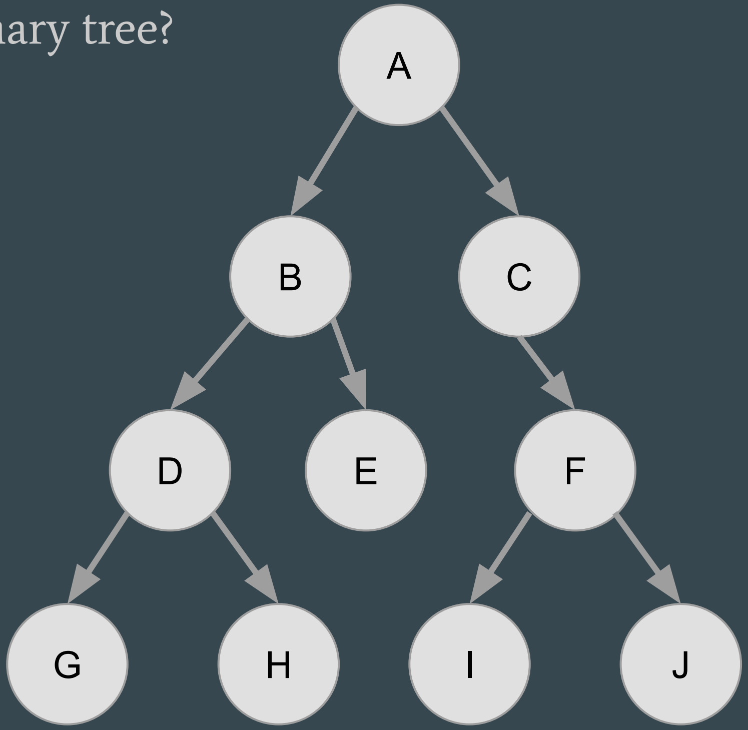
Example: Ten(10) Node Binary Tree

- What is the minimum height of a ten-node binary tree?
- At level N there can be at most 2^N nodes

	Running Sum
Level 0 => 1 node	1
Level 1 => 2 nodes	3
Level 2 => 4 nodes	7
Level 3 => 8 nodes	15

Answer minimum height = floor($\log_2 N$)

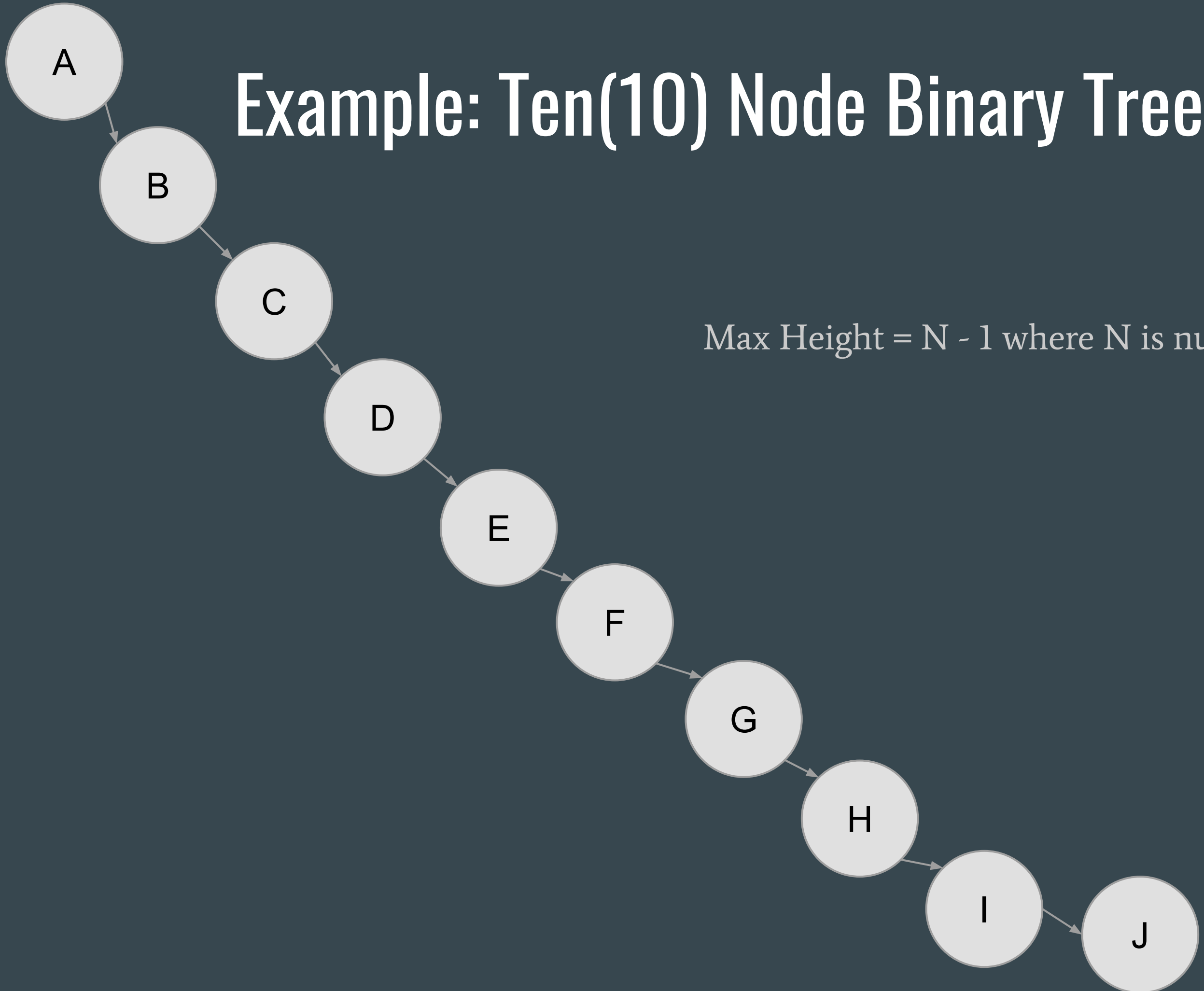
= floor(3.322) = 3



Example: Ten(10) Node Binary Tree

- What is the maximum height of a ten-node binary tree?
    - Think 10 node linked list?

Example: Ten(10) Node Binary Tree



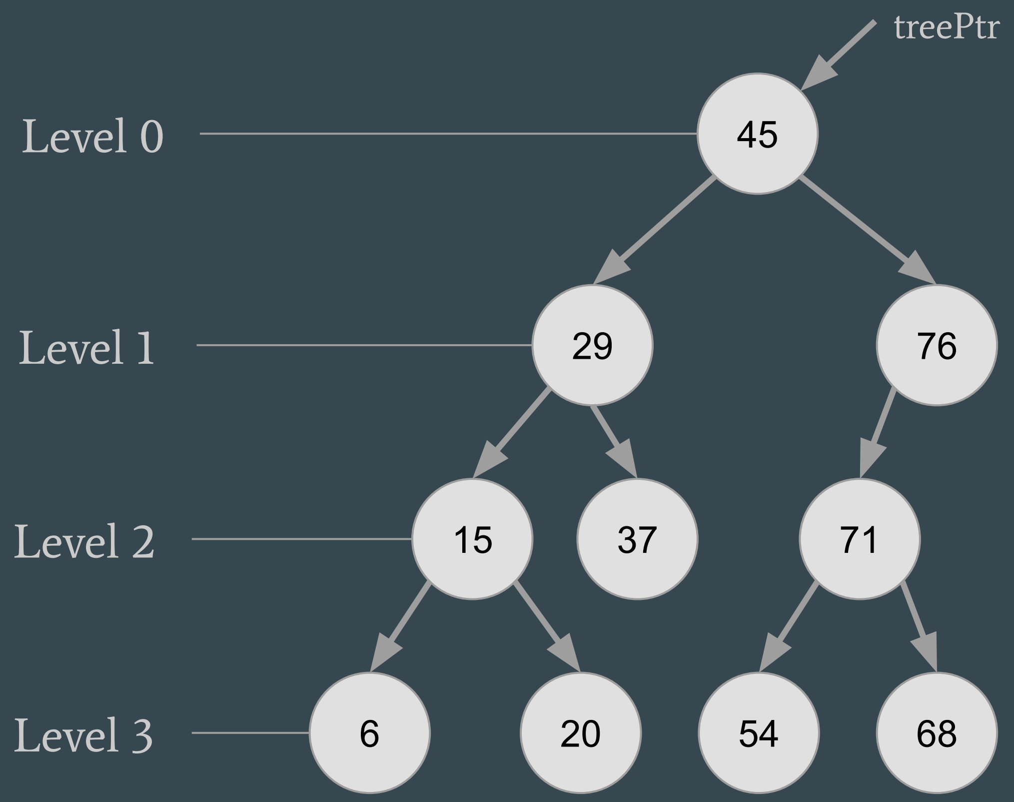
Max Height = N - 1 where N is number of Nodes

Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.

Binary Search Tree



Big-O Notation

- A notation that expresses computing time (complexity) as the term in a function that increases most rapidly relative to the size of the problem

Binary Search Tree Efficiency

- If we wish to determine if a particular value is in a binary search tree, what is the maximum number of comparisons we must make?

$$O(\log_2 N)$$

$$x = \log_2 n \iff 2^x = n.$$

- How does it compare to a sorted linked list?

$$O(N)$$

N = 32

Binary Search = ?
Sorted Linked List = ?

Binary Search Tree Efficiency

- If we wish to determine if a particular value is in a binary search tree, what is the maximum number of comparisons we must make?

$$O(\log_2 N)$$

$$x = \log_2 n \iff 2^x = n.$$

- How does it compare to a sorted linked list?

$$O(N)$$

N = 32

Binary Search = 32
Sorted Linked List = 5

More than 6x more efficient!

Traversal of a Binary Tree

1. Inorder Traversal
2. Postorder Traversal
3. Preorder Traversal

Inorder, Postorder, Preorder refer to when a particular node is visited relative to its left and right subtrees

Traversal of a Binary Tree

1. Inorder Traversal

A systematic way of visiting all nodes in a binary tree that visits the nodes in the left subtree of a node, then visits the node, and then visits the nodes in the right subtree of the node

Traversal of a Binary Tree

1. Inorder Traversal

A systematic way of visiting all nodes in a binary tree that visits the nodes in the left subtree of a node, then visits the node, and then visits the nodes in the right subtree of the node

Gives the nodes in nondecreasing order

Traversal of a Binary Tree

2. Postorder Traversal

A systematic way of visiting all nodes in a binary tree that visits the nodes in the left subtree of a node, then visits the nodes in the right subtree of the node, and then visits the node

Used to delete the tree.

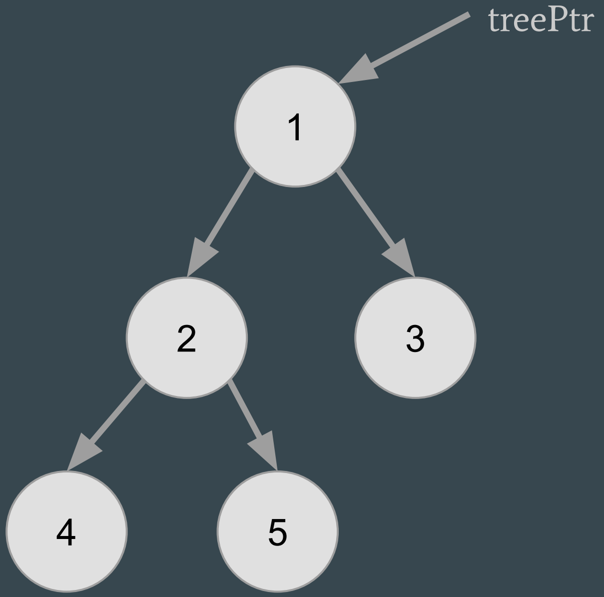
Traversal of a Binary Tree

3. Preorder Traversal

A systematic way of visiting all nodes in a binary tree that visits the node, then visits the nodes in the left subtree of the node, and then visits the nodes in the right subtree of the node

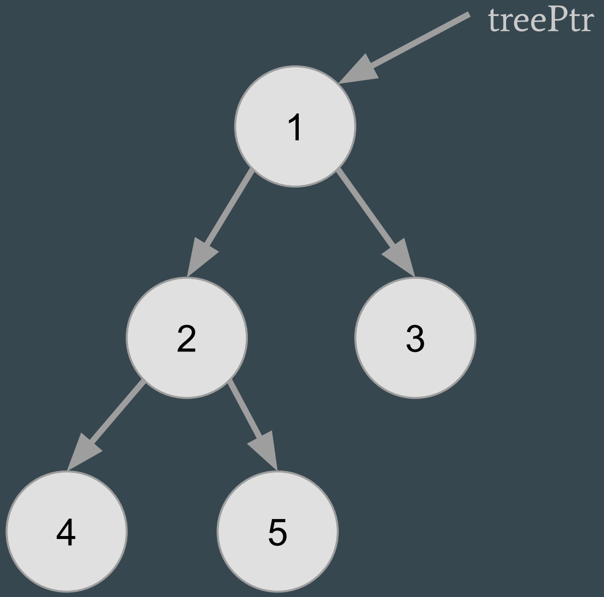
Used to create a copy of the tree.

Binary Tree Traversal Example



Inorder (Left, Root, Right)

Binary Tree Traversal Example



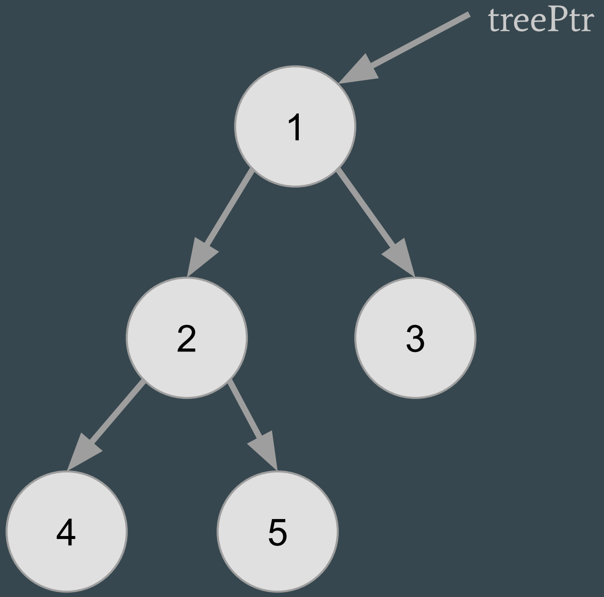
Inorder (Left, Root, Right)

4 2 5 1 3

Inorder Traversal

```
/* Given a binary tree, print its nodes in inorder*/
void printInorder(struct Node* node)
{
   if (node == NULL)
       return;
    /* first recur on left child */
   printInorder(node->left);
    /* then print the data of node */
   cout << node->data << " ";
    /* now recur on right child */
   printInorder(node->right);
}
```

Binary Tree Traversal Example

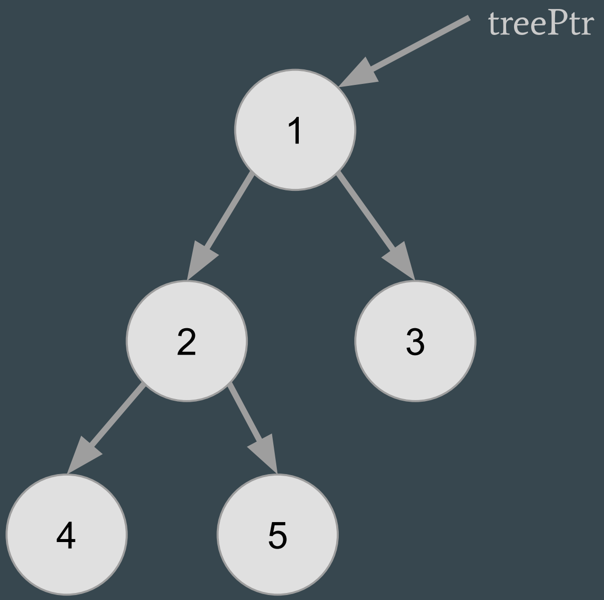


Inorder (Left, Root, Right)

4 2 5 1 3

Preorder (Root, Left, Right)

Binary Tree Traversal Example

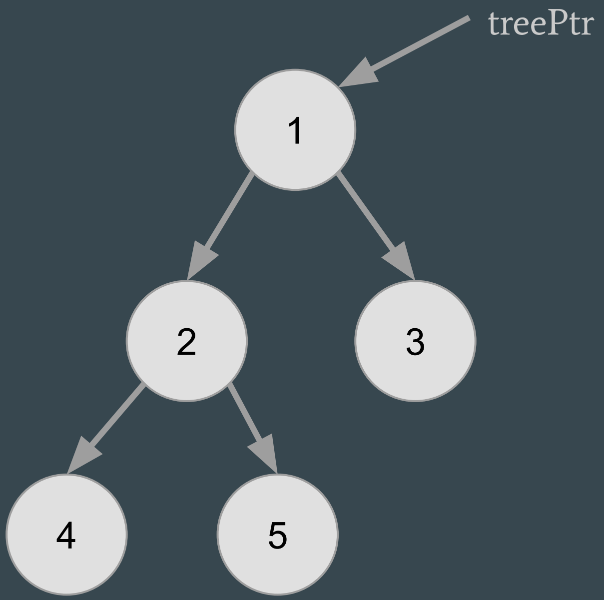



```
Inorder (Left, Root, Right)

4 2 5 1 3

Preorder (Root, Left, Right)

1 2 4 5 3
```

Preorder Traversal

```
/* Given a binary tree, print its nodes in preorder*/
void printPreorder(struct Node* node)
{
   if (node == NULL)
       return;
    /* first print data of node */
   cout << node->data << " ";
    /* then recur on left sutree */
   printPreorder(node->left);
    /* now recur on right subtree */
   printPreorder(node->right);
}
```

Binary Tree Traversal Example



Inorder (Left, Root, Right)

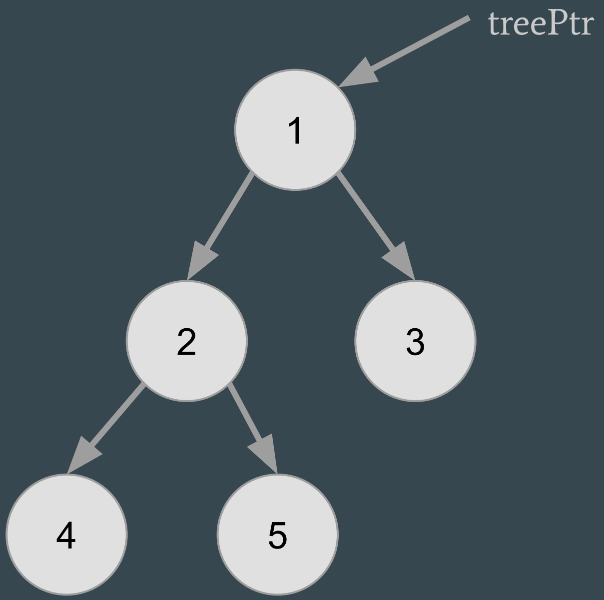
4 2 5 1 3

Preorder (Root, Left, Right)

1 2 4 5 3

Postorder (Left, Right, Root)

Binary Tree Traversal Example



Inorder (Left, Root, Right)

4 2 5 1 3

Preorder (Root, Left, Right)

1 2 4 5 3

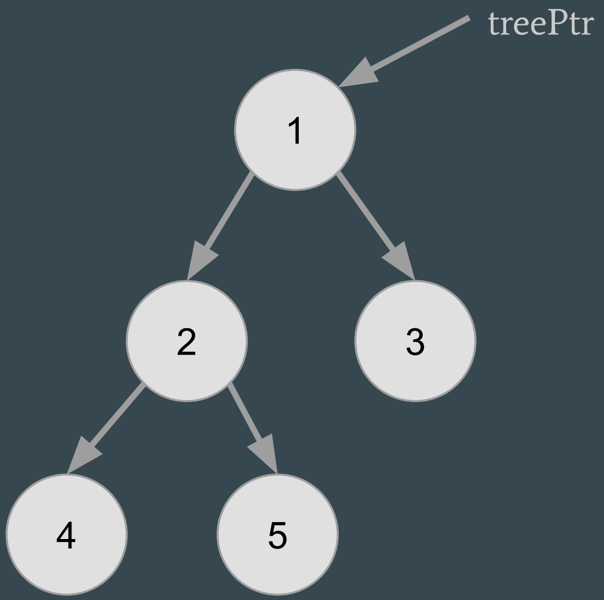
Postorder (Left, Right, Root)

4 5 2 3 1

Postorder Traversal

```
/* Given a binary tree, print its nodes according to the
"bottom-up" postorder traversal. */
void printPostorder(struct Node* node)
{
   if (node == NULL)
       return;
    // first recur on left subtree
   printPostorder(node->left);
    // then recur on right subtree
   printPostorder(node->right);
    // now deal with the node
   cout << node->data << " ";
}
```

Binary Tree Traversal Example



Inorder (Left, Root, Right)

4 2 5 1 3

Preorder (Root, Left, Right)

1 2 4 5 3

Postorder (Left, Right, Root)

4 5 2 3 1

Breadth First or Level Order

1 2 3 4 5

Binary Search Tree ADT

Structure Specification

- Placement of each node satisfies the binary search tree property
    - Key value of a node is greater than that in any node of its left subtree and less than that in any node of its right subtree

Basic Operations

- IsFull: determines whether tree is full
- IsEmpty: determines whether tree is empty
- LengthIs: determines number of elements in tree
- MakeEmpty: initializes tree to empty state

Binary Search Tree ADT

Basic Operations Continued

- InsertItem: adds specified item to tree
- DeleteItem: deletes a specified item from tree
- Print: prints tree contents to specified output file
- Etc.

Binary Search Tree ADT

tree.h

```
/*
** Binary Search Tree implementation in C++
*/
#include <iostream>
#include <new>
#include <fstream>

using namespace std;

struct Node {
   int data;
   Node* left;
   Node* right;
};

class BST {

   private:
       Node* root;

       Node* makeEmpty(Node* t);
       Node* insert(int x, Node* t);
       Node* remove(int x, Node* t);
       Node* find(int x, Node* t);
       Node* findMin(Node* t);
       Node* findMax(Node* t);
       void inorder(Node* t);
       int countNodes(Node* t);
```

Binary Search Tree ADT

tree.h - continued

```
    public:
        BST();
        ~BST();

        void insert(int x);
        void remove(int x);
        void display();

        void search(int x);
        int lengthIs();

        bool makeEmpty();
        bool isEmpty();
        bool isFull();
};
```

Binary Search Tree ADT

`IsEmpty()`

- Root pointer will point to NULL if BST is empty

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

Binary Search Tree ADT

`IsFull()`

- Similar to stack and queue
- Attempts to create a new node and if it is unable then it catches the exception thrown and returns true.

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

Binary Search Tree ADT

`LengthIs()`

- Consider the contributions of a single node and its subtrees first
- If node is a leaf, there are no subtrees so return 1 for the node itself
- Must know # of nodes in left subtree and # of nodes in right subtree
- Problem structure suggests recursion, so try to define a recursive algorithm

```
// LengthIs algorithm below
If (nodeptr->left == NULL) && (nodeptr->right == NULL)   // Node is
    a leaf
      return 1
Else
      return ( (# nodes in left subtree) + 1 + (# nodes in right
    subtree) )
```

Binary Search Tree ADT

`LengthIs()`

- Remember, Client does not have access to the ROOT pointer since it is private ==> use a helper function which does have access

```
// CountNodes Draft algorithm below
If (nodeptr->left == NULL) && (nodeptr->right == NULL)
    // Node is a leaf
      return 1
Else
      return ( (# nodes in left subtree) + 1 + (# nodes in
    right subtree) )
```

Binary Search Tree ADT

`CountNodes()`

```
CountNodes() Algorithm
// Base Case #1
if tree is NULL
    return 0
// Base case #2
else if (left_tree is NULL) AND (right_tree is NULL)
    return 1
else if (left_tree is NULL)
    return  CountNodes(right_tree) + 1
else if (right_tree is NULL)
    return  CountNodes(left_tree) + 1
else
    return  CountNodes(left_tree) + CountNodes(right_tree) + 1

// Equivalent CountNodes() Algorithm
//  Base case
if tree is NULL
    return 0
else
    return CountNodes(left_tree) + CountNodes(right_tree) + 1
```

Binary Search Tree ADT

`lengthIs()`

```
int BST::lengthIs() {
   return countNodes(root);
}


int BST::countNodes(Node* t) {
   if (t == NULL) {
       return 0;
   } else {
       return ( countNodes(t->left) + countNodes(t->right) + 1);
   }
}
```

Binary Search Tree ADT

`search() and find()`

```
void BST::search(int x) {
    root = find(x, root);
}
```

```
Node* BST::find(int x, Node* t) {
    if(t == NULL)
        return NULL;
    else if(x < t->data)
        return find(x, t->left);
    else if(x > t->data)
        return find(x, t->right);
    else
        return t;
}
```

Binary Search Tree ADT

`insert()`

```
// Public
void BST::insert(int x) {
   root = insert(x, root);
}

// Private
Node* BST::insert(int x, Node* t)
{
   if(t == NULL)
   {
      t = new Node;
      t->data = x;
      t->left = t->right = NULL;
   }
   else if(x < t->data)
      t->left = insert(x, t->left);
   else if(x > t->data)
      t->right = insert(x, t->right);
   return t;
}
```

Binary Search Tree ADT

`remove()`

```
Node* BST::remove(int x, Node* t) {
    Node* temp;
    // Base case
    if(t == NULL)
        return NULL;
    // If the data to be deleted is smaller then it is in the left subtree
    if(x < t->data)
        t->left = remove(x, t->left);
    // If the data to be deleted is greater then it is in the right subtree
    else if(x > t->data)
        t->right = remove(x, t->right);
    // If there are two children then get the smallest in the right subtree
    else if(t->left && t->right)
    {
        temp = findMin(t->right);
        t->data = temp->data;
        t->right = remove(t->data, t->right);
    }
    // if the data is the same as the t->data then this node needs to be deleted
    else
    {
        temp = t;
        if(t->left == NULL)
            t = t->right;
        else if(t->right == NULL)
            t = t->left;
        delete temp;
    }

    return t;
}
```

Binary Search Tree ADT

`findMin(Node* t)`

- Given a non-empty binary search tree, return the node with minimum key value found in that tree. Note that the entire tree does not need to be searched.

```
Node* BST::findMin(Node* t)
{
    if(t == NULL)
        return NULL;
    else if(t->left == NULL)
        return t;
    else
        return findMin(t->left);
}
```

Binary Search Tree ADT

`findMax(Node* t)`

- Given a non-empty binary search tree, return the node with maximum key value found in that tree. Note that the entire tree does not need to be searched.

```
Node* BST::findMax(Node* t) {
   if(t == NULL)
       return NULL;
   else if(t->right == NULL)
       return t;
   else
       return findMax(t->right);
}
```

Copy Constructor

- Shallow copying is normally used when
  - Passing parameters by value
  - Initializing a variable in a declaration
  - Returning an object as the return value of a function
  - Implementing the assignment operator
- If a copy constructor is present, it is used implicitly when
  - Passing parameters by value
  - Initializing a variable in a declaration
  - Returning an object as the return value of a function
- For deep copy with the assignment operator, you must
  - Write your own member function to perform the deep copy
  - Overload the assignment operator

Copy Constructor

```
void CopyTree(TreeNode*& copy, const TreeNode* originalTree);

TreeType::TreeType(const TreeType& originalTree)
// Calls recursive function CopyTree to copy originalTree
//  into root.
{
 CopyTree(root, originalTree.root);
}

void TreeType::operator=(const TreeType& originalTree)
// Calls recursive function CopyTree to copy originalTree
// into root.
{
 if (&originalTree == this)
   return;             // Ignore assigning self to self

 Destroy(root);      // Deallocate existing tree nodes
 CopyTree(root, originalTree.root);
}
void CopyTree(TreeNode*& copy, const TreeNode* originalTree)
// Post: copy is the root of a tree that is a duplicate
//       of originalTree.
{
 if (originalTree == NULL)
   copy = NULL;
 else
 {
   copy = new TreeNode;
   copy->info = originalTree->info;
   CopyTree(copy->left, originalTree->left);
   CopyTree(copy->right, originalTree->right);
 }
}
```

Double Pointers

```
void pass_by_value(int* p)
{
   //Allocate memory for int and store the address in p
   p = new int;
}

void pass_by_reference(int*& p)
{
   p = new int;
}

int main()
{
   int* p1 = NULL;
   int* p2 = NULL;

   pass_by_value(p1); //p1 will still be NULL after this call
   pass_by_reference(p2); //p2's value is changed to point to the newly allocate
memory

   return 0;
}
```

Balanced Binary Tree

- Order in which nodes are inserted determines the shape of the tree
- If data is already sorted before insertion in the tree, the tree shape will be skewed
- Randomly ordered data will keep the tree short and "bushy"
- A logarithmic height tree keeps the searching efficient

Relative Efficiency of BST

Operation	BST	Array-Based Linear List	Linked List
Class Constructor	O(1)	O(1)	O(1)
Destructor	O(N)	O(1) or O(N)	O(N)
MakeEmpty	O(N)	O(1) or O(N)	O(N)
LengthIs	O(N)	O(1)	O(1)
IsFull	O(1)	O(1)	O(1)
IsEmpty	O(1)	O(1)	O(1)
RetrieveItem	$O(\log_2 N)$	$O(\log_2 N)$	O(N)
InsertItem	$O(\log_2 N)$	O(N)	O(N)
DeleteItem	$O(\log_2 N)$	O(N)	O(N)