Chapter 20

Binary Trees

Definition and Applications of Binary Trees

- A binary tree is a non-linear linked list where each node may point to at most two other nodes.
Definition and Applications of Binary Trees

- A binary tree is anchored at the top by a tree pointer, which is like the head pointer in a linked list.
- The tree pointer is called the root.
- The root node has pointers to two other nodes, which are called children, or child nodes.

Definition and Applications of Binary Trees

- Every node in the tree is reachable from the root node.
- The root node has no predecessor.
  - Every other node has only one predecessor.
  - The predecessor is called the parent node.
Definition and Applications of Binary Trees

- A node that has no children is called a leaf node.
- All pointers that do not point to a node are set to NULL.

Definition and Applications of Binary Trees

- Binary trees can be divided into subtrees. A subtree is an entire branch of the tree, from one particular node down.
Traversing the Tree

- There are three common methods for traversing (visiting each node in) a binary tree and processing the value of each node:
  - *Inorder*
  - *Preorder*
  - *Postorder*
- Each of these methods is best implemented as a recursive function

Inorder Traversal - *Idr*

1. The node’s left subtree is traversed
2. The node’s data is processed
3. The node’s right subtree is traversed
Preorder Traversal - \textbf{dlr}

1. The node’s data is processed
2. The node’s left subtree is traversed
3. The node’s right subtree is traversed

Postorder Traversal - \textbf{ird}

1. The node’s left subtree is traversed
2. The node’s right subtree is traversed
3. The node’s data is processed
Activities for Binary Trees

- Count the number of leaves in a tree
- Find the height of the tree
  - Height is the longest path from the root to any leaf
- Evaluate the tree (if it represents an expression tree)
- Is the tree **strictly binary**
  - Every node has exactly two children or none

Activities for Binary Trees

- Is tree **leftist**?
  - Every node that has only one child, has a left child, but no right child.
- Is every node equal or greater than both children?
  - Also called a **heap**
- Count the number of times a value appears in the tree
- Print the tree by level
  - Need to use a queue
Activities for Binary Trees

- **Is the tree balanced?**
  - Height of left subtree and height of right subtree differ by at most 1

- **Given the following traversals, construct the tree**
  - Is the tree unique?
  - **Preorder** – TOERRISHUMAN
  - **Inorder** – EORSIAMUNHRT

---

Activities for Binary Trees

- **Reflect** a binary tree.
  - Exchange left and right children throughout

- **Are two trees similar?**
  - Does each tree have same branching structures, but possibly different node values

- **Are two trees mirror images of each other?**
Definition and Applications of Binary Trees

- Binary trees are excellent data structures for searching large amounts of information.
- They are commonly used in database applications to organize key values that index database records.
- When a binary tree is used to facilitate a search, it is called a **binary search tree**.

Information is stored in binary search trees in a way that makes a binary search simple. For example, look at the figure below:

Values are stored in a binary tree so that a node's left child holds data whose value is less than the node's data, and the node's right child holds data whose value is greater than the node's data.
Definition and Applications of Binary Trees

- For any node in the tree:
  - The nodes in the left subtree contain values less than the node's value
  - The nodes in the right subtree contain values greater than the node's data
- When an application is searching a binary tree, it starts at the root node
  - If the root node does not hold the search value, the application branches either to the left or right child, depending on whether the search value is less than or greater than the value at the root node

This process continues until the value is found.

The figure below illustrates the search pattern for finding the value P in the binary tree.
Binary Search Tree Operations

- We will demonstrate binary tree operations by constructing a class name `IntBinaryTree`
- The basis of our binary tree node is the class named `TreeNode`
  - I use a nested class in order to have constructors that initialize the fields in the class
  - This is different from the text

---

IntBinaryTree.h

```cpp
#ifndef INTBINARYTREE_H
#define INTBINARYTREE_H

class IntBinaryTree
{
private:
    class TreeNode
    {
        public:
            int value;
            TreeNode *left;
            TreeNode *right;
            TreeNode ( int v, TreeNode *l = NULL, TreeNode *r = NULL )
            { value = v; left = l; right = r; }
        }; // TreeNode
    TreeNode *root;
}; // IntBinaryTree
```

---
IntBinaryTree.h

public:
    void insert( TreeNode *&, TreeNode *& );
    void destroySubTree( TreeNode * );
    void deleteNode( int, TreeNode *& );
    void makeDeleteion( TreeNode * & );
    void displayInOrder( TreeNode * ) const;
    void displayPreOrder( TreeNode * ) const;
    void displayPostOrder( TreeNode * ) const;

CS 1410 - SJAllan Chapter 20 21

IntBinaryTree.h

public:
    IntBinaryTree ( void ) { root = NULL; }
    ~IntBinaryTree ( void ) { destroySubTree( root ); }
    void insertNode( int );
    bool searchNode( TreeNode *&, int );
    void remove( int );
    void showNodesInOrder( void ) { displayInOrder( root ); }
    void showNodesPreOrder( void ) { displayPreOrder( root ); }
    void showNodesPostOrder( void ) { displayPostOrder( root ); }
}; // IntBinaryTree
#endif

CS 1410 - SJAllan Chapter 20 22
**insertNode()**

```cpp
void IntBinaryTree::insertNode( int num )
{
    TreeNode *nn;
    nn = new TreeNode( num );
    insert(root, nn );
} // IntBinaryTree::insertNode
```

**insert()**

```cpp
void IntBinaryTree::insert( TreeNode *& np, TreeNode &nn )
{
    if ( !np )
        np = nn;
    else if ( nn->value < np->value )
        insert( np->left, nn );
    else
        insert( np->right, nn );
} // IntBinaryTree::insert
```
Inserting into a Binary Tree

```c++
#include "IntBinaryTree.h"

int main ( void )
{
    IntBinaryTree tree;
    cout << "Inserting nodes. ";
    tree.insertNode( 5 );
    tree.insertNode( 8 );
    tree.insertNode( 3 );
    tree.insertNode( 12 );
    tree.insertNode( 9 );
    cout << "Done.\n";
    return 0;
} // main
```

Binary Tree that is Built

- The figure below shows the structure of the binary tree built by the program

Note: The shape of the tree is determined by the order in which the values are inserted. The root node in the diagram above holds the value 5 because that was the first value inserted.
IntBinaryTree::displayInOrder

```cpp
void IntBinaryTree::displayInOrder ( TreeNode *tn )
{
    if ( tn )
    {
        displayInOrder( tn->left );
        cout << tn->value << endl;
        displayInOrder( tn->right );
    } // if
} // IntBinaryTree::displayInOrder
```

IntBinaryTree::displayPreOrder

```cpp
void IntBinaryTree::displayPreOrder ( TreeNode *tn )
{
    if ( tn )
    {
        cout << tn->value << endl;
        displayPreOrder( tn->left );
        displayPreOrder( tn->right );
    } // if
} // IntBinaryTree::displayPreOrder
```
IntBinaryTree::displayPostOrder

void IntBinaryTree::displayPostOrder ( TreeNode *tn )
{ if ( tn )
{ displayPostOrder( tn->left );
  displayPostOrder( tn->right );
  cout << tn->value << endl;
} // if
} // IntBinaryTree::displayPostOrder

Inserting and Printing Binary Trees

int main ( void )
{ IntBinaryTree tree;
  cout << "Inserting nodes.\n";
tree.insertNode( 5 );
tree.insertNode( 8 );
tree.insertNode( 3 );
tree.insertNode( 12 );
tree.insertNode( 9 );
cout << "Inorder traversal:\n";
tree.showNodesInOrder();
cout << "\nPreorder traversal:\n";
tree.showNodesPreOrder();
cout << "\nPostorder traversal:\n";
tree.showNodesPostOrder();
return 0;
} // main
Output

Inserting nodes.
Inorder traversal:
3
5
8
9
12

Postorder traversal:
3
9
12
8
5

Preorder traversal:
5
3
8
12
9

IntBinaryTree::searchNode()

bool IntBinaryTree::searchNode ( TreeNode * np, int v )
{ if ( !np )
    return false;
else if ( np->value == v )
    return true;
else if ( np->value < v )
    return searchNode( np->right, v );
else
    return searchNode( np->left, v );
} // IntBinaryTree::searchNode
Deleting a Node

- Deleting a leaf node is easy
- We simply find its parent and set the child pointer that links to it to NULL, and then free the node's memory
- But what if we want to delete a node that has child nodes?
  - We must delete the node while at the same time preserving the subtrees that the node links to

There are two possible situations when we are deleting a non-leaf node:
- The node has one child, or
- The node has two children
Deleting a Node

Deleting a node with one subtree

The figure shows how we link the node’s subtree with its parent.
Deleting a Node

The problem is not as easily solved, however, when the node we are about to delete has two subtrees.

We cannot attach both of the node's subtrees to its parent, so there must be an alternative solution.

One way is to attach the node's right subtree to the parent, and then find a position in the right subtree to attach the left subtree. The result is shown as follows:
Remove Node With No Children - (-4)

Delete Node With One Child – (18)
Node With Two Children – (12)

Find the minimum value in the right subtree

Deleting a Node
void IntBinaryTree::deleteAux ( TreeNode * &treePtr, int num )
{
    if ( num < treePtr->value )
        deleteAux( treePtr->left, num );
    else if ( num > treePtr->value )
        deleteAux( treePtr->right, num );
    else
        makeDeletion( treePtr );
} // IntBinaryTree::deleteNode

void IntBinaryTree::makeDeletion ( TreeNode * &treePtr )
{
    TreeNode *tempNodePtr;
    if ( !treePtr )
        cout << "Cannot delete empty node.\n";
    else if ( !treePtr->right )
    {
        tempNodePtr = treePtr;
        treePtr = treePtr->left; // Reattach the left child
        delete tempNodePtr;
    } // else if
    else if ( !treePtr->left )
    {
        tempNodePtr = nodePtr;
        treePtr = treePtr->right; // Reattach the right child
        delete tempNodePtr;
    } // else if
IntBinaryTree::makeDeletion

// If the node has two children.
else
    { // Move one node the right.
        tempNodePtr = treePtr->right;
        // Go to the end left node.
        while ( tempNodePtr->left )
            tempNodePtr = tempNodePtr->left;
        // Reattach the left subtree.
        tempNodePtr->left = treePtr->left;
        tempNodePtr = treePtr;
        // Reattach the right subtree.
        treePtr = treePtr->right;
        delete tempNodePtr;
    } // else
} // makeDeletion

Template Considerations for Binary Trees

- When designing your template, remember that any data types stored in the binary tree must support the <, >, and == operators
- If you use the tree to store class objects, these operators must be overridden
Activities for BST’s

- Is the tree in BST order?
- Delete the smallest element in a BST
- Alter the BST so that parent pointers are stored at each node
  - Show insert/delete with parent pointers