Trees

1. Introduction

• So far, ADTs have the following major operations: Insert, Delete, Retrieve

• Position oriented ADTs

  ADTs List: $i^{th}$ position
  Stack: first position
  Queue: first/last position

• Value Oriented ADTs

  ADT Sorted List: value of item x

• Tree supports the following ADTs

  Binary Tree: position oriented
  Binary Search Tree: value oriented
  Table: value oriented
  Priority Queue: value oriented

• Terminology

  A tree is a data structure that organizes data in hierarchical way

Example:

Note: all examples below will refer to the above tree
9. Binary Trees and Binary Search Trees

- Definition: A *tree* is a finite set of one or more nodes such that

  - There is a specially designated node called the *root*

  - The remaining nodes are partitioned into \( n \geq 0 \) disjoint sets \( T_1, T_2, \ldots, T_n \), where each of these sets is a tree. \( T_1, T_2, \ldots, T_n \) are called the *subtrees* of the root.

- Note: may consider the empty tree is a tree with 0 node

- Definition: The number of subtrees of a node is called its *degree*

  Example:

<table>
<thead>
<tr>
<th>Node</th>
<th>Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>M</td>
<td>0</td>
</tr>
</tbody>
</table>

- Definition: Nodes that have degree zero are called *leaf nodes* or *terminal nodes*.

  Example: L,F,M,H,N,O,J,K are leaf nodes

- Definition: Nodes that are not leaf nodes are called *interior nodes* or *nonterminal nodes*.

  Example: A,B,C,D,E,G,I are interior nodes

- Definition: The roots of the subtrees of a node \( X \), are called the *children* of \( X \). \( X \) is the *parent* of its children.

  Example:

<table>
<thead>
<tr>
<th>Node</th>
<th>Children</th>
<th>Node</th>
<th>Parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, C and D</td>
<td>G</td>
<td>C</td>
</tr>
<tr>
<td>G</td>
<td>M</td>
<td>N</td>
<td>I</td>
</tr>
</tbody>
</table>
• You may also use the notions of grandchildren, grand parent etc

• Definition: Children of the same parent are called *siblings*

Example:  

<table>
<thead>
<tr>
<th>Siblings</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B, C, D</td>
<td></td>
</tr>
<tr>
<td>I, J, K</td>
<td></td>
</tr>
<tr>
<td>N, O</td>
<td></td>
</tr>
</tbody>
</table>

• Definition: The *degree of a tree* is the maximum degree of the nodes in the tree

Example: Degree of the above tree is 3

• Definition: The *ancestors* of a node are all the nodes along the path from the root to that node

Example:  

<table>
<thead>
<tr>
<th>Node</th>
<th>Ancestors</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>A, C and G</td>
</tr>
<tr>
<td>E</td>
<td>B, A</td>
</tr>
</tbody>
</table>

• Definition: The *descendants* of a node are all nodes in all the subtrees of the node

Example:  

<table>
<thead>
<tr>
<th>Node</th>
<th>Descendants</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B, C, D, ..., O</td>
</tr>
<tr>
<td>B</td>
<td>E, L, F</td>
</tr>
<tr>
<td>D</td>
<td>I, J, K, N, O</td>
</tr>
<tr>
<td>G</td>
<td>M</td>
</tr>
<tr>
<td>N</td>
<td>none</td>
</tr>
</tbody>
</table>

• Property: There is only one path from the root of a tree to every other node
• Definition: The *level* of a node is defined recursively as follows:

The root of a tree is at level 1

If a node is at level k, then its children are at level k+1

Example:

<table>
<thead>
<tr>
<th>Node</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B, C, D</td>
<td>2</td>
</tr>
<tr>
<td>M</td>
<td>4</td>
</tr>
</tbody>
</table>

• Definition: The *height* (or *depth*) of a tree is defined to be the maximum level of any node in the tree

Example: The height of the above tree is 4

May consider the height of an empty tree is 0

2. ADT Binary Tree

• Definition: A *binary tree* is a finite set of nodes that is either empty or consists of a root and two disjoint trees called the left subtree and the right subtree.

![Fig. 1](image1)

![Fig. 2](image2)

![Fig. 3 & Fig. 4](image3)

• In general, the degree of a binary tree is two

• Definition: a *full binary tree* is a binary tree where all interior nodes have 2 children and all terminal nodes are in the same level. Example: Figure 2.

• Definition: a *leftist (rightist)* tree is a binary tree where every interior node has only a left (right) subtree. Example: Figure 3 (Figure 4)
• Definition: A balanced binary tree is a binary tree in which the left and right subtrees of any node have heights that differ by at most 1. Example: Figure 1 and Figure 2

• Definition: a complete binary tree is a full binary tree except that some of the rightmost leaves may be missing. Example: Figure 1

• Note: A full binary tree is a complete binary tree. A complete binary tree is balanced tree

• Properties:

The maximum number of nodes on level $j$ of a binary tree is $2^{j-1}$, $j \geq 1$
The maximum number of nodes in a binary tree of height $k$ is $2^k - 1$, $k \geq 1$

• ADT binary tree operations

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createBinaryTree()</code></td>
<td>creates an empty binary tree</td>
</tr>
<tr>
<td><code>createBinaryTree(in rootItem:TreeItemType)</code></td>
<td>creates a one node binary tree</td>
</tr>
<tr>
<td><code>createBinaryTree(in rootItem:TreeItemType, inout leftTree:BinaryTree, inout rightTree:BinaryTree)</code></td>
<td>creates a binary tree whose root is rootItem with left subtree leftTree and right subtree rightTree</td>
</tr>
<tr>
<td><code>destroyBinaryTree()</code></td>
<td>destroys a binary tree</td>
</tr>
<tr>
<td><code>isEmpty() : boolean {query}</code></td>
<td>determines if the tree is empty</td>
</tr>
<tr>
<td><code>getRootData() : TreeItemType throw TreeException</code></td>
<td>returns the data item in the root. throws exception if the tree is empty</td>
</tr>
<tr>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
</tr>
<tr>
<td>+setRootData(in newItem:TreeItemType)</td>
<td>replaces the item in the root by newItem. If the tree is empty, create a root node whose data item is newItem. Throws exception if a new node cannot be created.</td>
</tr>
<tr>
<td>+attachLeft(in newItem:TreeItemType)</td>
<td>attaches a left (right) child containing the newItem to the root of a binary tree. throws exception if a new node cannot be created or if the tree is empty or a left subtree (right) already exists.</td>
</tr>
<tr>
<td>+attachRight(in newItem:TreeItemType)</td>
<td></td>
</tr>
<tr>
<td>+attachLeftSubtree(inout leftTree:BinaryTree)</td>
<td>attaches a copy of leftTree (rightTree) as the left (right) subtree of the root of a binary tree. throws exception if the tree is empty or a left subtree (right) already exists.</td>
</tr>
<tr>
<td>+attachRightSubtree(inout rightTree:BinaryTree)</td>
<td></td>
</tr>
<tr>
<td>+detachLeftSubtree(out leftTree:BinaryTree)</td>
<td>detaches the left (right) subtree of a binary tree’s root and retains it in leftTree (rightTree) throws exception if the tree is empty</td>
</tr>
<tr>
<td>+detachRightSubtree(out rightTree:BinaryTree)</td>
<td></td>
</tr>
<tr>
<td>+leftSubtree():BinaryTree</td>
<td>returns a copy of left (right) subtree without detaching the subtree. return an empty tree if the tree is empty.</td>
</tr>
<tr>
<td>+rightSubtree():BinaryTree</td>
<td></td>
</tr>
<tr>
<td>+preorderTraverse(in visit: FunctionType)</td>
<td>traverses a binary tree in preorder (inorder or postorder) and calls the function visit() once with each node</td>
</tr>
<tr>
<td>+inorderTraverse(in visit: FunctionType)</td>
<td></td>
</tr>
<tr>
<td>+postorderTraverse(in visit: FunctionType)</td>
<td></td>
</tr>
</tbody>
</table>
• Example: using the ADT operations to build a binary tree

T1.setRootData('B')
T1.attachLeft('D')
T1.attachRight('E')

T2.setRootData('C')
T2.attachLeft('F')
T2.attachRight('G')

T3.setRootData('A')
T3.attachLeftSubtree(T1)
T3.attachRightSubTree(T2)

• Binary Tree Traversal

• There are 3 ways to traverse all the nodes of a binary tree and produce a listing. Note: Each node only visit once

• Let T be a binary tree with root r

• Inorder Traversal of T is
  ▪ an inorder traversal of all nodes in the left subtree of r
  ▪ r
  ▪ an inorder traversal of all nodes in the right subtree of r

• Preorder Traversal of T is
  ▪ r
  ▪ a preorder traversal of all nodes in the left subtree of r
  ▪ a preorder traversal of all nodes in the right subtree of r

• Postorder Traversal of T is
  ▪ a postorder traversal of all nodes in the left subtree of r
  ▪ a postorder traversal of all nodes in the right subtree of r
  ▪ r
• Example:

<table>
<thead>
<tr>
<th>Tree 1</th>
<th>Tree 2</th>
<th>Tree 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inorder</td>
<td>GDBEHACF</td>
<td>CFEBGHDA</td>
</tr>
<tr>
<td>Preorder</td>
<td>ABDGEHCF</td>
<td>ABCEFDGH</td>
</tr>
<tr>
<td>Postorder</td>
<td>GDHEBFCA</td>
<td>FECHGDBA</td>
</tr>
</tbody>
</table>

3. Array-based representation of a binary tree

• Use array of nodes to represent a tree

```cpp
const int MAX_NODES = 100; // maximum number of nodes
typedef string TreeItemType; // example : itemType is string
class TreeNode // a node in the tree
{
    private:
        TreeNode();
        TreeNode(const TreeItemType& nodeItem, int left, int right);
        TreeItemType item; // data portion
        int leftChild; // index to left child
        int rightChild; // index to right child

        // friend class - can access private parts
        friend class BinaryTree;
}; // end class TreeNode
```
• Example: TreeNode[MAX_NODES] tree; // array of nodes
  int root; // index of root
  int free; // index of free list

Initially:

root = -1
free = 0 // use rightChild to point to next free node

<table>
<thead>
<tr>
<th>Index</th>
<th>item</th>
<th>leftChild</th>
<th>rightChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-2</td>
<td>-</td>
<td>-</td>
<td>N-1</td>
</tr>
<tr>
<td>N-1</td>
<td>-</td>
<td>-</td>
<td>-1</td>
</tr>
</tbody>
</table>

After building a tree

Root = 0
Free = 6

Index | item | leftChild | rightChild |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-2</td>
<td>-</td>
<td>-</td>
<td>N-1</td>
</tr>
<tr>
<td>N-1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
After deleting nodes C and F

Root = 0
Free = 2

<table>
<thead>
<tr>
<th>Index</th>
<th>item</th>
<th>leftChild</th>
<th>rightChild</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>7</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N-2</td>
<td>-</td>
<td>-</td>
<td>N-1</td>
</tr>
<tr>
<td>N-1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- Alternatively, you may eliminate leftChild and rightChild fields, root and free.

The root of a tree always start at 0. Children of item at index i (if they exist) are 2*i+1 for left child and 2*i+2 for right child.

Example:

<table>
<thead>
<tr>
<th>Index</th>
<th>Item</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>E</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

This is good for complete tree!
4. Pointer-based representation of a binary tree

- Use pointers to link the nodes in a tree. TreeNode.h is:

```cpp
typedef string TreeItemType; // should specify the TreeItemType here!

class TreeNode // node in the tree
{
    private:
        TreeNode() {};
        TreeNode( const TreeItemType& nodeItem, TreeNode *left = NULL,
            TreeNode *right = NULL) : item(nodeItem), leftChildPtr(left),
            rightChildPtr(right) {}
        TreeItemType item;        // data portion
        TreeNode *leftChildPtr;   // pointer to left child
        TreeNode *rightChildPtr;  // pointer to right child
    friend class BinaryTree;
}; // end TreeNode class
```

- A TreeException class can be defined easily. TreeException.h is:

```cpp
#include <stdexcept>
#include <string>
using namespace std;

class TreeException : public runtime_error
{
    public:
        TreeException(const string &message =""):
            runtime_error(message.c_str()) {}
}; // end TreeException
```
9. Binary Trees and Binary Search Trees

- Definition file: BinaryTree.h is

```cpp
// ********************************************************
// Header file BinaryTree.h for the ADT binary tree.
// ********************************************************
#include "TreeException.h"
#include "TreeNode.h" // contains definitions for TreeNode and TreeItemType

typedef void (*FunctionType)(TreeItemType& anItem);

class BinaryTree
{
  public:
    // constructors and destructor:
    BinaryTree();
    BinaryTree(const TreeItemType& rootItem);
    BinaryTree(const TreeItemType& rootItem, BinaryTree& leftTree,
               BinaryTree& rightTree);
    BinaryTree(const BinaryTree& tree);
    virtual ~BinaryTree();

    // binary tree operations:
    virtual bool isEmpty() const;
    virtual TreeItemType getRootData() const throw(TreeException);
    virtual void setRootData(const TreeItemType& newItem) throw(TreeException);
    virtual void attachLeft(const TreeItemType& newItem) throw(TreeException);
    virtual void attachRight(const TreeItemType& newItem) throw(TreeException);
    virtual void attachLeftSubtree(BinaryTree& leftTree) throw(TreeException);
    virtual void attachRightSubtree(BinaryTree& rightTree) throw(TreeException);
    virtual void detachLeftSubtree(BinaryTree& leftTree) throw(TreeException);
    virtual void detachRightSubtree(BinaryTree& rightTree) throw(TreeException);
    virtual BinaryTree leftSubtree() const;
    virtual BinaryTree rightSubtree() const;
    virtual void preorderTraverse(FunctionType visit);
    virtual void inorderTraverse(FunctionType visit);
    virtual void postorderTraverse(FunctionType visit);
    virtual BinaryTree& operator=(const BinaryTree& rhs);  // overloaded operator:
```
protected:

// protect TreeNode from objects, but OK for derived classes
// to access tree node pointers (derived classes are not friend of
// node class)

BinaryTree(TreeNode *nodePtr); // constructor

void copyTree(TreeNode *treePtr, TreeNode* & newTreePtr) const;
// Copies the tree rooted at treePtr into a tree rooted at newTreePtr.
// Throws TreeException if a copy of the tree cannot be allocated.

void destroyTree(TreeNode * &treePtr); // Deallocates memory for a tree.

// The next four functions retrieve and set the values of the private data
// member root and child pointers. This is not used in here.
TreeNode *rootPtr() const;
void setRootPtr(TreeNode *newRoot);
void getChildPtrs(TreeNode *nodePtr, TreeNode * &leftChildPtr,
                 TreeNode * &rightChildPtr) const;
void setChildPtrs(TreeNode *nodePtr, TreeNode *leftChildPtr,
                  TreeNode *rightChildPtr);

// protected functions for recursive traversals
void preorder(TreeNode *treePtr, FunctionType visit);
void inorder(TreeNode *treePtr, FunctionType visit);
void postorder(TreeNode *treePtr, FunctionType visit);

private:
    TreeNode *root; // pointer to root of tree
}; // End of header file.
• Discussion:

overloaded assignment operator

to create a copy of existing tree (should compare with copy constructor!)

protected functions

if you do not want the instances (or objects) of a class to access the internal data structure directly, use “protected” functions (these functions are available for derived class)

Example: the protected constructor in binary tree class

function pointer

function can be passed as a parameter into another function using function pointer

Example: visit() function as parameter in preorder(), inorder() and postorder()

• Implementation File: BinaryTree.C

    // ********************************************************
    // Implementation file BinaryTree.cpp for the ADT binary tree.
    // ********************************************************
    #include "BinaryTree.h" // header file
    #include <cassert> // for assert()

    BinaryTree::BinaryTree() : root(NULL)
    {
    } // end default constructor
BinaryTree::BinaryTree(const TreeItemType& rootItem)
{
    root = new TreeNode(rootItem, NULL, NULL);
    assert(root != NULL);
} // end constructor

BinaryTree::BinaryTree(const TreeItemType& rootItem,
                        BinaryTree& leftTree, BinaryTree& rightTree)
{
    root = new TreeNode(rootItem, NULL, NULL);
    assert(root != NULL);

    attachLeftSubtree(leftTree);
    attachRightSubtree(rightTree);
} // end constructor

BinaryTree::BinaryTree(const BinaryTree& tree)
{
    copyTree(tree.root, root);
} // end copy constructor

BinaryTree::BinaryTree(TreeNode *nodePtr): root(nodePtr)
{
} // end protected constructor

BinaryTree::~BinaryTree()
{
    destroyTree(root);
} // end destructor

bool BinaryTree::isEmpty() const
{
    return (root == NULL);
} // end isEmpty

TreeItemType BinaryTree::getRootData() const throw(TreeException)
{
    if (isEmpty())
        throw TreeException("TreeException: Empty tree");
    return root->item;
} // end rootData
void BinaryTree::setRootData(const TreeItemType& newItem) throw(TreeException) {
    if (!isEmpty())
        root->item = newItem;
    else
    {  root = new TreeNode(newItem, NULL, NULL);
        if (root == NULL)
            throw TreeException("TreeException: Cannot allocate memory");
    } // end if
} // end setRootData

void BinaryTree::attachLeft(const TreeItemType& newItem) throw(TreeException) {
    if (isEmpty())
        throw TreeException("TreeException: Empty tree");
    else if (root->leftChildPtr != NULL)
        throw TreeException("TreeException: Cannot overwrite left subtree");
    else // Assertion: nonempty tree; no left child
    {  root->leftChildPtr = new TreeNode(newItem, NULL, NULL);
        if (root->leftChildPtr == NULL)
            throw TreeException("TreeException: Cannot allocate memory");
    } // end if
} // end attachLeft

void BinaryTree::attachRight(const TreeItemType& newItem) throw(TreeException) {
    // similar to attachLeft()
} // end attachRight

void BinaryTree::attachLeftSubtree(BinaryTree& leftTree) throw(TreeException) {
    if (isEmpty())
        throw TreeException("TreeException: Empty tree");
    else if (root->leftChildPtr != NULL)
        throw TreeException("TreeException: Cannot overwrite left subtree");
    else // Assertion: nonempty tree; no left child
    {  root->leftChildPtr = leftTree.root;
        leftTree.root = NULL;
    }
} // end attachLeftSubtree
void BinaryTree::attachRightSubtree(BinaryTree& rightTree) throw(TreeException)
{
    // similar to attachLeftSubtree
} // end attachRightSubtree

void BinaryTree::detachLeftSubtree(BinaryTree& leftTree) throw(TreeException)
{
    if (isEmpty())
        throw TreeException("TreeException: Empty tree");
    else
    {
        leftTree = BinaryTree(root->leftChildPtr);
        root->leftChildPtr = NULL;
    } // end if
} // end detachLeftSubtree

void BinaryTree::detachRightSubtree(BinaryTree& rightTree) throw(TreeException)
{
    // similar to detachLeftSubtree
} // end detachRightSubtree

BinaryTree BinaryTree::leftSubtree() const
{
    TreeNode *subTreePtr;
    if (isEmpty())
        return BinaryTree();
    else
    {
        copyTree(root->leftChildPtr, subTreePtr);
        return BinaryTree(subTreePtr);
    } // end if
} // end leftSubtree

BinaryTree BinaryTree::rightSubtree() const
{
    // similar to leftSubtree
} // end rightSubtree

void BinaryTree::preorderTraverse(FunctionType visit)
{
    preorder(root, visit);
} // end preorderTraverse
void BinaryTree::inorderTraverse(FunctionType visit) {
    inorder(root, visit);
} // end inorderTraverse

void BinaryTree::postorderTraverse(FunctionType visit) {
    postorder(root, visit);
} // end postorderTraverse

BinaryTree& BinaryTree::operator=(const BinaryTree& rhs) {
    if (this != &rhs) {
        destroyTree(root); // deallocate left-hand side
        copyTree(rhs.root, root); // copy right-hand side
    } // end if
    return *this;
} // end operator=

void BinaryTree::copyTree(TreeNode *treePtr, TreeNode *& newTreePtr) const {
    // preorder traversal
    if (treePtr != NULL) {
        // copy node
        newTreePtr = new TreeNode(treePtr->item, NULL, NULL);
        if (newTreePtr == NULL) throw TreeException("TreeException: Cannot allocate memory");

        copyTree(treePtr->leftChildPtr, newTreePtr->leftChildPtr);
        copyTree(treePtr->rightChildPtr, newTreePtr->rightChildPtr);
    } else
        newTreePtr = NULL; // copy empty tree
} // end copyTree
void BinaryTree::destroyTree(TreeNode *& treePtr) {
    // postorder traversal
    if (treePtr != NULL) {
        destroyTree(treePtr->leftChildPtr);
        destroyTree(treePtr->rightChildPtr);
        delete treePtr;
        treePtr = NULL;
    } // end if
} // end destroyTree

TreeNode *BinaryTree::rootPtr() const {
    return root;
} // end rootPtr

void BinaryTree::setRootPtr(TreeNode *newRoot) {
    root = newRoot;
} // end setRoot

void BinaryTree::getChildPtrs(TreeNode *nodePtr, TreeNode *& leftPtr, TreeNode *& rightPtr) const {
    leftPtr = nodePtr->leftChildPtr;
    rightPtr = nodePtr->rightChildPtr;
} // end getChildPtrs

void BinaryTree::setChildPtrs(TreeNode *nodePtr, TreeNode *leftPtr, TreeNode *rightPtr) {
    nodePtr->leftChildPtr = leftPtr;
    nodePtr->rightChildPtr = rightPtr;
} // end setChildPtrs
void BinaryTree::preorder(TreeNode *treePtr, FunctionType visit)
{
    if (treePtr != NULL)
    {
        visit(treePtr->item);
        preorder(treePtr->leftChildPtr, visit);
        preorder(treePtr->rightChildPtr, visit);
    } // end if
} // end preorder

void BinaryTree::inorder(TreeNode *treePtr, FunctionType visit)
{
    if (treePtr != NULL)
    {
        inorder(treePtr->leftChildPtr, visit);
        visit(treePtr->item);
        inorder(treePtr->rightChildPtr, visit);
    } // end if
} // end inorder

void BinaryTree::postorder(TreeNode *treePtr, FunctionType visit)
{
    if (treePtr != NULL)
    {
        postorder(treePtr->leftChildPtr, visit);
        postorder(treePtr->rightChildPtr, visit);
        visit(treePtr->item);
    } // end if
} // end postorder

// End of implementation file.

• Sample Usage :

libra% more useBinary.C
#include "BinaryTree.h" // binary tree operations
#include <iostream>
using namespace std;

void display(TreeItemType& anItem)  // set type to “int”
{ cout << " " << anItem; }

```c++
int main()
{
    BinaryTree tree1, tree2, left; // empty trees
    BinaryTree tree3(70); // tree with only a root 70

    tree1.setRootData(40); tree1.attachLeft(30); tree1.attachRight(50);
    tree2.setRootData(20); tree2.attachLeft(10); tree2.attachRightSubtree(tree1);

    BinaryTree binTree(60, tree2, tree3);

    binTree.inorderTraverse(display); cout << "\n--------------------\n";
    binTree.leftSubtree().inorderTraverse(display); cout << "\n--------------------\n";

    binTree.detachLeftSubtree(left);
    left.inorderTraverse(display);
    cout << "\n--------------------\n";

    binTree.inorderTraverse(display); cout << "\n--------------------\n";
    return 0;
} // end main
```

```
libra% a.out
10 20 30 40 50 60 70
--------------------
10 20 30 40 50
--------------------
10 20 30 40 50
--------------------
60 70
--------------------
libra%
```
5. ADT Binary Search Tree

- Problem: Binary tree does not support “search” operation efficiently

- Definition: a binary search tree is a binary tree that satisfies the following 3 properties
  - Each node has record with unique key (or search key)
  - All keys in left subtree < key in the root < All keys in right subtree
  - Both left subtree and right subtree are binary search trees

- Examples:

```
       60
      /  \
    70   30
   /   /  \
  65  90   5  40
```

```
       20
      /   \
    15    25
   /     /  \
  12    5   40
```

```
       20
      /   \
    15    25
   /     /  \
  12    5   40
```

Not a BST
Binary search tree operations (+ several operations in binary tree ADT)

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+createSearchTree()</td>
<td>creates an empty binary search tree</td>
</tr>
<tr>
<td>+destroySearchTree()</td>
<td>destroys a binary search tree</td>
</tr>
<tr>
<td>+isEmpty() : boolean {query}</td>
<td>determines if the tree is empty</td>
</tr>
<tr>
<td>+searchTreeInsert( in newItem:TreeItemType) throw TreeException</td>
<td>Inserts newItem into the tree throws exception if the insertion is not success</td>
</tr>
<tr>
<td>+searchTreeDelete( in searchKey:KeyType) throw TreeException</td>
<td>Delete from the tree the item whose key equals searchKey. throws exception if the operation is not success</td>
</tr>
<tr>
<td>+searchTreeRetrieve( in searchKey:KeyType, out treeItem:TreeItemType) throw TreeException</td>
<td>Retrieve into treeItem the item in the tree whose key equals searchKey. throws exception if the operation is not success</td>
</tr>
<tr>
<td>+preorderTraverse( in visit: FunctionType)</td>
<td>traverses a binary search tree in preorder (inorder or postorder) and calls the function visit() once with each node</td>
</tr>
<tr>
<td>+inorderTraverse( in visit: FunctionType)</td>
<td></td>
</tr>
<tr>
<td>+postorderTraverse( in visit: FunctionType)</td>
<td></td>
</tr>
</tbody>
</table>

A possible design for search key class & record class in binary search tree

typedef string KeyType; // A class for searchKey. Assume it is “string”
class KeyedItem
{
    public:
        KeyedItem() {};
        KeyedItem(const KeyType& keyValue): searchKey(keyValue) {} 
        KeyType getKey() const { return searchKey; }
    
    private:
        KeyType searchKey;
}; // end class
// A class for a personal record with name, idNum, phoneNumber etc
// The searchKey is name (derived this class from class KeyedItem)

class Person : public KeyedItem
{
    public:
        Person() {}
        Person(const string &name, const string &id, const string &phone)
            : KeyedItem(name), idNum(id), phoneNumber(phone) {}
    private:
        string idNum;
        string phoneNumber;
        //... and other data about the person
}; // end class

• Assume a treeNode has a leftChildPtr, a rightChildPtr and an item (or a record)

• Outline of general search algorithm
  // The idea will be used to develop insertion(), deletion() and retrieval().

search(in T:BinarySearchTree, in searchKey:KeyType)
{
    if (T is empty)  Not found!
    else if (searchKey == key of the root)  Found!
    else if (searchKey < key of the root)
        search(Left subtree of T, searchKey);
    else
        search(Right subtree of T, searchKey);
}

• Outline of insertion algorithm

  • Need to make sure that the tree is still a binary search tree after insertion!
  • Starting from root, use search() strategy to locate the correct position
  • If the item already in the tree, the item is not inserted.
  • If the item is not in the tree, search() stops when the treePtr is NULL.
    Which is the proper location for the new node. Let treePtr = new node.
insertItem(inout treePtr:TreeNodePtr, in newItem:TreeItemType) {
    if (treePtr is NULL) // insert into this location
    {
        create a new node, let treePtr point to it,
        copyNewItem into new node, set pointers in new node to NULL,
    }
    else if (treePtr->item.getKey() == newItem.getKey())
    {
        // the item already in the tree, do not insert item..
    }
    else if (treePtr->item.getKey() < newItem.getKey())
    {
        insertItem(treePtr->rightChildPtr, newItem)
    }
    else
    {
        insertItem(treePtr->leftChildPtr, newItem)
    }
}
• Example: May get different binary search trees if input orderings of data are different

(1 2 3 4)   (4 3 2 1)   (3 4 1 2)

1 2 3 4
2
3
4

• Outline of deletion Algorithm

• Need to make sure that the tree is still a binary search tree after deletion!
• First, locate the desired node.
• Assume treePtr points to the desired node.

• There are 3 cases for desired node:

  It is a leaf → easy case, set treePtr to NULL

  It has only one child → set treePtr to the valid child node

  It has two children →

  • find the location of successor item, i.e. the node with smallest key which is larger than deleted key.
  • The successor item is in the left most leaf (node X) of right subtree
  • copy successor item to the desired node’s item
  • delete node X (either case I or case II)
9. Binary Trees and Binary Search Trees

• // This function locates the node to be deleted. It is pointed by treePtr.
  deleteItem(inout treePtr:TreeNodePtr, in searchKey:KeyType)
  {
    if (treePtr == NULL) Do not find the desired node!
    else if (treePtr->item.getKey() == searchKey)
    {
      deleteNodeItem(treePtr)
    }
    else if (searchKey < TreePtr->item.getKey())
    {
      deleteItem(treePtr->leftChildPtr, searchKey)
    }
    else
    {
      deleteItem(treePtr->rightChildPtr, searchKey)
    }
  }

  // This function deletes the node where NodePtr points
  deleteNodeItem(inout nodePtr:TreeNodePtr)
  {
    // Case I : delete a leaf
    if ( nodePtr->leftChildPtr == NULL AND
         nodePtr->rightChildPtr == NULL)
    {
      delete nodePtr; nodePtr = NULL;
    }
    // Case II : only 1 subtree
    else if ( nodePtr->leftChildPtr == NULL OR
              nodePtr->rightChildPtr == NULL)
    {
      delPtr = nodePtr;
      if ( nodePtr->leftChildPtr == NULL) // Left subtree is empty
        nodePtr = nodePtr->rightChildPtr;
      else
        nodePtr = nodePtr->leftChildPtr
      delete delPtr;
    }
  }
// Case III : have 2 subtrees
else
{
    // find the successor item; replace the desired item with it
    processLeftmost(nodePtr -> rightChildPtr, successorItem)
    nodePtr->item = successorItem
}

// delete the smallest item in the tree with root NodePtr, return the item
processLeftmost(inout nodePtr:TreeNodePtr, out treeItem:TreeItemType)
{
    if (nodePtr->leftChildPtr == NULL) // this is the successor item
    {
        treeItem = nodePtr->Item
        delPtr = nodePtr

        // no left subtree, may have right subtree
        // replace the left most node by right subtree
        nodePtr = nodePtr->rightChildPtr
        delete delPtr
    } else {
        processLeftmost(NodePtr->leftChildPtr, treeItem)
    }
}

• Outline of retrieval Algorithm

• Same as search algorithm

• See text book for a pointer-based implementation of Binary Search Tree ADT
• Inorder listing of a binary search tree

• Theorem: The inorder traversal of binary search tree will visit its nodes in sorted search key order

• Example:

```
40
/   \
30   50
/ \   / \n37 45 60
/ \   / \
35 47 55
```

Result:
30 35 37 45 47 50 55 60

Proof: Easy! use induction by the number of nodes

• Assume a binary search tree with N nodes, the maximum height is N, and minimum height is \( \log_2(N+1) \)

• Worst case running time of binary search tree operation, let \( h = \) height of binary tree

<table>
<thead>
<tr>
<th>Operations</th>
<th>Running Time</th>
<th>Worst Case Running Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieval</td>
<td>O(h)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Insertion</td>
<td>O(h)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Deletion</td>
<td>O(h)</td>
<td>O(N)</td>
</tr>
<tr>
<td>Traversal</td>
<td>O(N)</td>
<td>O(N)</td>
</tr>
</tbody>
</table>

Note: In CSC510, you will study how to maintain a balanced binary search tree, i.e. minimum height binary search tree (only need to consider insertion and deletion operations) \( \rightarrow \) the height of the tree is O(log N) \( \rightarrow \) Worst case running time of above operations (excluding Traversal) are O(log N)
6. ADT General Tree

- One way to represent a general tree, a tree node has
  ```
  treeItemType Item;
  ptrType leftChild;  // point to left child
  ptrType rightSibling;  // point to right child
  ```

- Example:

```
  A
  /   \
 B     C
  \
 D
  \
  E     F
  \
  G     H
  \
  I
```

```
  A
  /   \
 B     C
  \   /\  \
 E   F   G
  \   /\  \
 H   I
```