ECE 71/191T – Data Structures and Algorithms

Dr. Gregory R. Kriehn, Fresno State
C++ Homework Assignment: Chapter 19

Code Due By: Midnight on Mon, Apr 24
Write-up Due By: Class on Tue, Apr 25

HOMEWORK #35 – Binary Trees

Reconstruct the binary **`nodeType`** template class and the **`binaryTreeType`** template class from Malik. **`binaryTreeType`** contains the following members:

```
Public:
  const binaryTreeType<elemType>& operator=
       (const binaryTreeType<elemType>&);
  // Overload Assignment Operator

  bool isEmpty() cosnt;
  // Checks for empty binary tree

  void inorderTraversal() const;
  // inorder traversal of binary tree

  void preorderTraversal() const;
  // preorder traversal of binary tree

  void postorderTraversal() const;
  // postorder traversal of binary tree

  int treeHeight() const;
  // Returns tree height of binary tree

  int treeNodeCount()const;
  // Returns number of nodes of binary tree

  int treeLeavesCount() const;
  // Returns number of leaves of binary tree

  void destroyTree();
  // Deallocates memory occupied by binary tree

  virtual bool search(const elemType &searchItem) const = 0;
  // Function to determine if searchItem is in binary tree

  virtual void insert(const elemType &insertItem) = 0;
  // Function to insert insertItem in the binary tree
```

```
  virtual void deleteNode(const elemType &deleteItem) = 0;
  // Function to delete deleteItem from binary Tree

  binaryTreeType(const binaryTreeType<elemType> &otherTree);
  // Copy constructor

  binaryTreeType();
  // Default constructor

  ~binaryTreeType();
  // Destructor

Protected:
  nodeType<elemType> *root;


Private:
  void copyTree(nodeType<elemType>* &copiedTreeRoot,
            nodeType<elemType>* otherTreeRoot);
  // Makes a copy of the binary tree to which otherTreeRoot
  // points

  void destroy(nodetype<elemType>* &p);
  // Function to destroy binary tree to which p points

  void inorder(nodeType<elemType> *p) const;
  // Function to do an inorder traversal of binary tree
  // to which p points

  void preorder(nodetype<elemType> *p) const;
  // Function to do a preorder traversal of binary tree
  // to which p points

  void postorder(Nodetype<elemType> *p) const;
  // Function to do a postorder traversal of binary tree
  // to which p points

  int height(nodetype<elemType> *p) const;
  // Function to determine the height of the binary tree
  // to which p points

  elemType max(elemType x, elemType y) const;
  // Function to determine the larger of x and y

  int nodeCount(nodeType<elemType> *p) const;
  // Function to determine the number of nodes in binary tree
  // to which p points
```

```
int leavesCount(nodetype<elemType> *p) cosnt;
// Function to determine the number of leaves in binary
// tree to which p points
```

NOTE: The **`max()`** function has been changed so that it will work generically with any data type.

Next, create a **`bSearchTreeType`** template class that defines the abstract members in **`binaryTreeType`**:

```
Public:
  bool search(const elemType &searchItem) const;
  // Function to determine if serachItem is in binary search
  // tree

  void insert(const elemType &insertItem);
  // Function to insert insertItem in binary search tree

  void deleteNode(const elemType &deleteItem);
  // Function to delete deleteItem from the binary search
  // tree

Private:
  void deleteFromtree(nodeType<elemType>* &p);
  // Function to delete the node to which p points from
  // the binary search tree
```

Fully debug and verify the operation of your abstract and derived classes before proceeding further.

Next write a program to perform the following:

1. Build a binary search tree $\mathbf{T_1}$.
2. Perform a postorder traversal of $\mathbf{T_1}$, and while performing the traversal, insert the nodes into a second binary search tree $\mathbf{T_2}$.
3. Perform a preorder traversal of $\mathbf{T_2}$, and while performing the traversal, insert the nodes into a third binary search tree $\mathbf{T_3}$.
4. Perform an inorder traversal of $\mathbf{T_3}$.
5. Output the heights and number of leaves in each of the three binary search trees.

To help you with your program, you may want to augment your **`bSearchTreeType`** template class with the following member functions:

```
Public:
  void insertPostorder(bSearchTreeType<elemType> &otherTree);
  // Calls insertPost() to perform a postorder traversal of a
```

```
    // binary search tree and copies the nodes into a new tree

    void insertPreorder(bSearchTreeType<elemType> &otherTree);
    // Calls insertPre() to perform a preorder traversal of a
    // binary search tree and copies the nodes into a new tree

  Private:
    void insertPost(bSearchTreeType<elemType> &otherTree,
         binaryTreeNode<elemType> *p);
    // Performs the postorder traversal of otherTree and
    // inserts the nodes into a new tree

    void insertPre(bSearchTreeType<elemType> &otherTree,
         binaryTreeNode<elemType> *p);
    // Performs the preorder traversal of otherTree and
    // inserts the nodes into a new tree
```

Test your program with the following data:

```
Enter numbers ending with -999
68 43 10 56 77 82 61 82 33 56 72 66 99 88 12 6 7 21 -999

The item to be inserted is already in the list -- Duplicate
82 is not allowed.
The item to be inserted is already in the list -- Duplicate
56 is not allowed.

Tree 1 Nodes in Postorder:
   7 6 21 12 33 10 66 61 56 43 72 88 99 82 77 68

Tree 2 Nodes in Preorder:
   7 6 21 12 10 33 66 61 56 43 72 68 88 82 77 99

Tree 3 Nodes in Inorder:
   6 7 10 12 21 33 43 56 61 66 68 72 77 82 88 99

Tree 1 Height: 6
Tree 1 Leaves: 5

Tree 2 Height: 8
Tree 2 Leaves: 6

Tree 3 Height: 8
Tree 3 Leaves: 6
```