CS372 Programming Assignment #5

(Individual Programming Project implementing a Binary Search Tree and Recursion)

WARNING: Programming assignment #5 will ***not*** be accepted ***more than 4 days late***.

Part 1: The Program

Write a program to implement storing a list of integers into a binary search tree.
The program will:

- Read integer data from a text file and store unique integers into a binary tree.

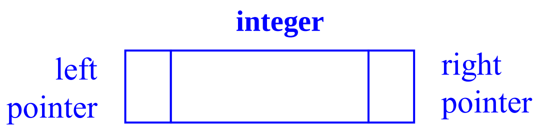
- After all data has been read, within a loop, let the user choose to:
  - Display all the integers in the binary tree
  - Add a value to the tree
  - Delete a value from the tree
  - Find an value within the binary tree, and display its subtree
  - Exit the program

You will create ***Abstract Data Type*** data definitions and functions for the binary tree.

Data Structure Overview

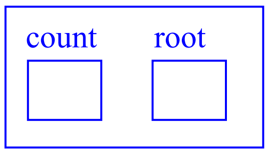
The individual **nodes** in the binary tree should each contain one integer and two pointers (one pointer pointing left and one pointer pointing right) only.

Sample Tree Node:



The binary search tree will be represented by a pointer to a structure that contains a field for the number of nodes in the binary search tree and a field for a ***pointer*** to a **node**, which will point to the root node of the binary search tree. Initially, the binary search tree pointer will be NULL. When the binary search tree is created, the structure will be dynamically allocated.

Sample BST structure:



You must implement this data structure from scratch (no use of templates from the STL allowed).

ADT Functions Overview

You will be required to implement the following functions to operate on your binary search tree list data structure:

> **CreateTree** – Allocate a binary search tree structure, and initialize the root **pointer in it** to NULL and the count to 0, to indicate an empty tree. Returns a pointer to the new binary search tree structure.
>
> **IsEmpty –** returns ***true*** if binary search tree is empty, ***false*** otherwise
>
> **CreateNode –** allocate and fills new node. Passes back pointer to new node, or NULL if node could not be allocated.
>
> **InsertNode** – inserts a new node into the correct location within a binary search tree.
>
> **FindNode** – searches for a value within a binary search tree. Passes back a flag indicating if the value was found and a pointer to a node. If the value was found, the pointer will point to the node containing the value. If the value was not found, the pointer will point to the last node searched before determining that the value could not be found (or NULL if the tree was empty).
>
> **DeleteNode** – deletes a node from the binary search tree.
>
> **InOrderDisplay** – neatly displays all integers in the list in sorted order, using a ***recursive***, in-order traversal.
>
> **FreeNodes** – recursively de-allocates all dynamic memory allocated to nodes in the binary search tree.
>
> **DestroyTree** – de-allocates all dynamic memory allocated for the binary search tree.

Only the functions listed above will be allowed to access the fields within the binary search tree data structure. You can call these functions from each other, when necessary. All other code must use these functions to access the binary search tree.

Implementation Details

1) Create an empty binary search tree.

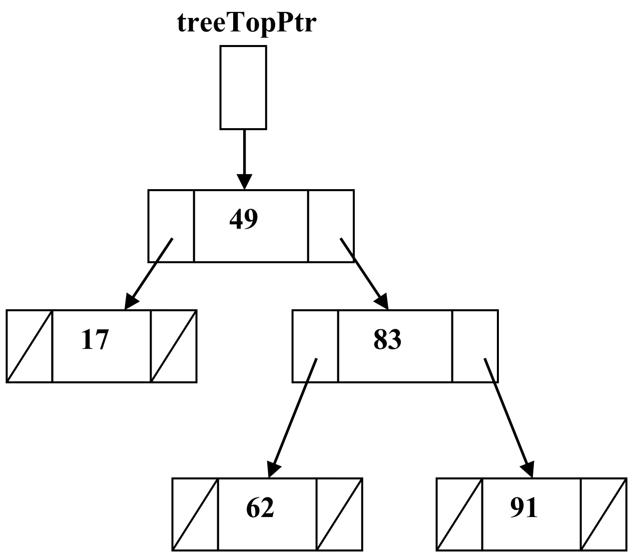
2) Read integer data from a text file

   - Prompt for a filename from the user. Loop until the user enters the name of a file that exists. Then open the file.
   - Confirm that the file is not empty. If it is, exit the file reading loop and proceed to step 3.
   - If there is data, read the data in the file. As you read each integer from the file:
     - Search for the integer in the existing tree.
     - If the integer is found, issue a message that it will be ignored (not inserted into the tree).

- If the integer is not yet in the tree:
  - Allocate memory for the new node and validate that memory was allocated.
    - If there is a memory allocation error, the program should issue an error message, stop reading the data from the file, and proceed to step 3.
  - Place the integer data into the new node and initialize the pointers.
  - Place the new node into the correct location within the binary search tree.

NOTE: Your input data file will be a list of positive, non-zero integers, separated by whitespace. The file may contain duplicate values, but only unique values will be used. It is important that the integers ***NOT*** be in any kind of sorted order within the file.

Diagram of Sample Binary Search Tree
(created from integer file containing values 49, 83, 91, 17, and 62, in that order):



3) Display the total number of integers in the binary search tree.

   Then display a menu to the user, with the following choices:

   a) Show all integers in the binary search tree

      - Use a ***recursive*** **in-order** tree traversal to display all the integers associated with each node in the tree in sorted order.
      - Format the display so that the numbers are in neat columns, with 10 integers displayed per line.

        For formatting purposes, you can assume that all integers in the data file will be 4 digits or less.

Sample display:
```
Values stored in entire binary search tree are:
    56    82    99   169   290   301   472   777   778   780
   801   823   845   856   888   999   999  1004  1555  1690
  2222  5678  6789
```

or if the tree is empty, simply display:
```
Binary search tree is empty.
```

b) Add an integer to the tree.

- Prompt for an integer, and determine if the integer is in the tree
  - If integer is found, issue a message saying it cannot be added.
  - If integer **is** not found, add it to the tree.

c) Delete an integer from the tree.

- Prompt for an integer, and determine if the integer is in the tree
  - If integer is **not** found, issue a message saying it cannot be found.
  - If integer **is** found, remove the integer from the tree and deallocate the node memory.

d) Find an integer within the tree, and display its subtree

- Prompt for an integer, and determine if the integer is in the tree
  - If integer is **not** found, issue a message saying so
  - If integer **is** found, display the subtree whose root is the integer.

Example:
```
Enter value to find:  83

Values stored subtree with root 83 are:
    62    83    91
```

e) Exit the program

Loop around the menu until the user chooses to exit the program.

4) Before exiting, free up all dynamic memory allocated for the binary tree.

The program must be modular (i.e. broken down into logical functions) and use correct parameter passing. Use of global variables will NOT be allowed. And object-oriented programming (classes, objects, templates, STL, etc) may NOT be used.

And remember, your code should follow all required coding standards for formatting and documentation from Content section 1.8.

Part 2: Efficiency Analysis

Analyze your program as follows:

a) Analyze the efficiency of each function (except **main**) using:
n = number of integers in all the text files read

Calculate both the ***average*** case and ***worst*** case efficiency of each function within your program. Explain your logic for calculating the efficiencies.

b) Using the efficiencies for each function from (a) above, determine the ***average*** and ***worst*** case efficiencies of the **main** function. And use them to determine the big-O efficiency for the entire program. Explain your logic.

Part 3: Items Due

By **midnight Tuesday of week 8**:
Your **program source code** and **program analysis**
Also include any data files you used to test your program

WARNING:
Programs that do not compile, are not modular, use ANY global variables, or are submitted more than 4 days late, will NOT be accepted.

Submit your program code and analysis, along with any test data files to the ***Programming Assignment 5*** folder in the ***Dropbox*** area of the online course. All necessary files should be attached to **one** dropbox submission.

Before submitting your program and data files, name them as follows:
`Lastname-assn5-prog.cpp`
`Lastname-assn5-analysis.docx`
`Lastname-assn5-testdata.txt`

If you have multiple test data files, append a letter on the end of each filename.

Examples: **`Smith-assn5-prog.cpp`**
`Smith-assn5-analysis.docx`
`Smith-assn5-testdataA.txt`
`Smith-assn5-testdataB.txt`

Part 4: Grading

The grading rubric that will be used for this assignment is included on the next pages.

CS 372 Advanced Programming and Algorithms
Individual Programming Assignment #5 Grading Rubric

Rating Rating Category	Exemplary	Proficient	Basic (needs work)	Not Demonstrated
Documentation	Documentation clearly explains what code does. Includes complete and accurate **file** and **function** headers, as detailed in the CS372 coding standards, along with additional in-line comments at appropriate locations.	Documentation includes all required headers, but lacks clarity or details in some places, OR violated minor details of the CS372 coding standards for comments, OR code is completely over commented.	Documentation is incomplete and/or incorrect and/or formatted incorrectly. May have violated significant details of the CS372 coding standards for comments or did not document the underlying program intent.	Only a few (or no) comments in program.
Data Storage	Constants used for fixed values. Correct data types for all variables including binary search tree nodes and tree structure. All definitions within the correct program scope. Followed CS372 coding standards for data storage.	A few minor errors in constant usage, data declaration scope, data typing or assignment OR violated some minor details of the CS372 coding standards.	One or more other major errors in constant usage, declaration scope, data typing, or assignment AND/OR violated of significant details of the CS372 coding standards for data storage.	Did not follow any of the CS372 coding standards for data storage.
File Usage	Prompts user for filename. Loops until entered filename exists. Opens file correctly. Reads integers from input file correctly. Recognizes end of file correctly, whether the file contains a newline on the last line or not. Closes file immediately after use.	Does not prompt for filename (hardcoded) OR does not check file exists, OR problems recognizing valid filenames, OR opens file incorrectly, OR reads integers incorrectly OR does not recognize end of file OR does not close file immediately after use.	Multiple problems with input file usage (from list under Proficient column).	Input data file not used.
Program Processing: Tree & Node Creation	Correctly creates empty tree. Correctly allocates new nodes and places data into them. Verifies memory allocation before using it. There are no memory leaks.	Minor problem tree or node creation.	Major (or multiple minor) problems with tree or node creation.	Both tree and node creation fail.
Program Processing: Node Insertion	Inserts each integer into the correct location within tree.	Minor problem with node insertion.	Major (or multiple minor) problems with node insertion.	All integers placed in wrong locations.
Program Processing: Searching	Correctly finds specified values. Issues error message if not found.	Minor problem with searching.	Major (or multiple minor) problems with searching.	No values correctly found.
Program Processing: Node Deletion	Correctly deletes integer from tree and de-allocates node memory.	Minor problem with deleting.	Major (or multiple minor) problems with deleting.	No values correctly deleted.
Program Processing: Tree Traversal	Correctly implements recursive in-order tree traversal to display sorted integers in tree/subtree.	Tree traversal has minor errors.	Tree traversal has multiple minor errors or major errors, or not implemented recursively.	Binary search tree not traversed at all.

Program Processing: Memory De-allocation	Correctly de-allocates memory for all nodes in binary tree on exit.	Memory de-allocation has minor errors.	Memory de-allocation has major errors.	Memory not de-allocated before exiting program.
Program Processing: menu/loop	Loops around menu until user chooses to exit.	Minor error in menu/loop implementation.	Multiple minor or major error(s) in menu/loop implementation.	No loop. Program only executes once.
Modularity (functional breakdown)	Program is efficient, modular in design, and logically organized. Each module performs ONE well-defined task and is defined, prototyped, and called correctly. main function minimal – does little more than call other functions. All required ADT functions were implemented correctly, and no other functions directly accessed the binary search tree.	Program is mostly efficient, modular, and logically organized. Program may include one unnecessary function, a function that performs too many tasks, and/or a function that is incorrectly defined or called. A required function may have been defined incorrectly, or another function may have directly accessed the binary search tree.	Program is only somewhat modular (missing necessary functional breakdown), AND/OR multiple modules perform too many tasks, AND/OR multiple parts of the program are not efficient or logically organized, OR multiple problems with module definitions/calls or incorrectly accessing the tree.	Program contains very little modularity (too many tasks performed from main function) but does contain at least two prototyped functions, AND/OR contains an adequate number of functions, but they are not implemented efficiently.
C++ Constructs / Parameter Passing/ Readability / Miscellaneous	Demonstrates understanding of program, control, and file structures. Appropriate use of language with only necessary parameter passing. Code is well organized, easy to follow, and adheres to all CS372 coding standards for code usage (e.g. no use of **break/return** to exit loop).	A parameter or two was passed incorrectly (value vs. reference), OR an unnecessary parameter(s) was passed, OR there was minor improper control or data structure usage, AND/OR had occasional spacing, indentation, and/or other minor organizational issues.	Many problems with control/data structure usage or parameter definitions and usage, AND/OR one other major error or coding standard violation, AND/OR substantial spacing, indentation, and/or other organizational issues.	Demonstrates minimal understanding of control/data structures and/or proper parameter passing AND/OR has major coding standard violations, AND/OR code is poorly organized and difficult to read AND/OR has other major construct/readability issues.
Analysis of Code Efficiency	Correctly analyzes the computational efficiencies of each function. Correctly determines the big-O efficiency for the entire program.	One or two minor errors, but the big-O efficiency for the entire program is logical using the function efficiencies given.	Many errors in calculating the computational efficiencies for the functions and the overall program efficiency.	Analysis not submitted.
Unacceptable	Does not compile OR is not modular OR uses one or more global variables OR over 4 days late.			