CPSC 2430 Data Structures

Winter Quarter 2024

Assignment 3 – BST Operations Implementation **Due: 10:00pm, Jan 31, 2023**

Following Lab 3, in this assignment we continue the implementation of ShelterBST basic operations. You will also split your code into a header (.h) and multiple code (.cpp) files.

```
/* SheterBST.cpp Class definitions */
ShelterBST::ShelterBST(){
  //define your ShelterBST constructor here
}
ShelterBST::TreeNode* ShelterBST::insert(TreeNode* root, Pet* pet){
  //define your private insert method here, ORDER BY NAME this time
}
// define the rest of your private and public methods
```

```
/* assignment3.cpp User interface to test all 15 functions, for example: */

int main() {
      // Welcome to my implementation of ShelterBST
      // Please choose the operation you want:
      // 1. Insert a node
      // 2. Search a value
      // 3. Delete a node
      // …
      // 0. Quit the system
}
```

Your ShelterBST class should implement the following functions. However, this time, you will sort your ShelterBST **by Pet Name** (instead of by pet age like in lab3):

A. **Insert** a node to ShelterBST (returns a TreeNode*)
   This time, you will need to ensure there are no duplicates in your insert function (i.e., the tree should refuse to insert a Pet with a duplicate name. You can print out an error message to screen.).

B. **Search** (return matching TreeNode* or nullptr if not found). Searches a pet by NAME. If a pet is found with the given name, the TreeNode* to that node is returned, nullptr otherwise.

C. **InOrder**, **PreOrder**, **PostOrder** (void, display only) traversals as defined in Lab3. You can reuse this code, but modify it to use Pet **name** for all comparisons (instead of age).

D. Find the **parent** for a given pet name (returns a TreeNode*)
Your private method should take as input an existing Pet name stored in your tree, and return the TreeNode* that is its parent node. Your public method should display the Pet name and age of the parent node. You do not need to consider the situation when the input value is not stored in your BST. Only input the values already stored in your BST.

E. Find the **predecessor** for a given node with 2 children (returns a TreeNode*). You do not need to consider the situation when the input node has one or no children.
Your private method should take as input an existing Pet name stored in your tree, and return the TreeNode* that is its inorder predecessor node. Your public method should display the Pet name and age of the inorder predecessor node.

F. **Number** of nodes in your tree (returns an int)
Return the number of total nodes in your BST. The return value should be an integer.

G. Find the **number of internal nodes** in your tree (returns an int)
Return the number of internal nodes in BST. The return value should be an integer.

H. Find the **number of nodes** at a given level (returns an int)
Return the number of nodes at a given in your BST. For example, if your BST is not empty, at level zero there is one node (the root).

I. Find the **height** of the tree (returns an int)
Return the height of your BST. The return value should be an integer.

J. **IsBalanced** check (returns a boolean)
Input the root node of this tree. If this BST is balanced, return **true**; otherwise, return **false**.

K. **Delete** a node based on a Pet name (returns a TreeNode*)
Delete a node based on the pet name given by the user. After deleting the node, your BST should be reconstructed. If the node to be deleted has two children, reconstruct the tree using the **inorder predecessor (you should use the method you will write for D).** You do not need to consider the situation when the input value is not stored in your BST. Only input the values already stored in your BST.

L. **Destroy** the tree (returns TreeNode*)
Return *nullptr* after the tree is destroyed successfully. Destroying the tree should be done carefully so as not to leak memory.

M. **Test** function (void)

Just like in Lab3, write a test function that inserts at least 10 pets in your ShelterBST (one of them should attempt to insert a duplicate name). Test the rest of the functions in your tree at least once. Clearly label each test for display.

You should **provide users with an interface** to interact with your BST implementation. Create a cpp named assignment3.cpp where your main function is located. *The interface should also be provided within your assignment3.cpp*.

Assignment 3 Submission

You need to submit the following files:

- ShelterBST.cpp
- ShelterBST.h
- assignment3.cpp
- Makefile

Before submission, you should ensure your program has been compiled and tested (extensively). Your assignment receives zero if your code cannot be compiled and executed.

You can submit your program multiple times before the deadline. The last submission will be used for grading.

To submit your assignment, you should execute the following script (assuming your files are on cs1.seattleu.edu):

```
/home/fac/mjilani/submit/24wq2430/assignment3_submit
```

Assignment 3 Grading Breakdown

Breakdown	Points	Note
makefile	5	Your makefile correctly compiles each code file with all the required flags, links and generates the executable program.
parent	5	Return the ancestor/parent node when given an existing Pet name stored in ShelterBST
predecessor	5	Return the predecessor node when given an existing Pet name (with two children) stored in ShelterBST

tree nodes	5	Return the total number of nodes in ShelterBST
internal nodes	5	Return the total number of internal nodes in ShelterBST
nodes at level	10	Return the number of nodes at a given level in ShelterBST
tree height	5	Return the height of your ShelterBST
balance	10	Check if your ShelterBST is balanced or not. If it is balanced, return true; otherwise, return false.
delete	20	Delete an existing node from your ShelterBST and reconstruct it using its inorder predecessor
destroy	10	Destroy your BST. Return nullptr when destroyed successfully
test	10	Test all your functions.
user interface	10	Provide a user interface to interact with your ShelterBST
deductions	0	Points will be deducted if you don't use pet pointers, don't sort or search your BST by pet NAME, allow duplicates, use incorrect return types, or are missing any of the required tests. Points will also be deducted for lack of functional decomposition, use of global variables (not constants), inconsistent indentation, lack of comments, messy output, etc.