Name: Erfan Rasti

Student Number: 9823034

Report of 3rd Homework of Advanced Programming C++

GitHub Repository HTTPS:

ErfanRasti/AdvancedProgramming-HW3 (github.com)

In this code, we are going to implement a binary search tree and its related algorithms to manage nodes.

This code consists of two nested classes, the outer class is "BST" and the inner one is "Node". We define the "Node" class as a public member of the "BST" class to make a nested class.

Let's explain the member functions of class "Node":

- **Default Constructor of Class "Node"**
  This constructor initializes the class member variables when there are no initial values. The default value for member value "value" is zero and two children nodes' pointers are *"nullptr".*
  It should be mentioned that when we want to access the inner class, we should use the "OUTTER_CLASS::INNER_CLASS::MEMBER_FUNCTION" structure as a header.

- **Parameterized Constructor of Class "Node"**
  This constructor takes the desired values for initializing the class as the input and passes these values to the member variables. We should receive the input children nodes as pointer variables.

- **Copy Constructor of Class "Node"**
  This constructor takes another node as input and copies the member variables of that node to the new node.

- **Overloaded Operator << for Class "Node"**
  << operator by itself is not associated with class "Node", therefore if we want to define it as a member function of class "Node" we should define it as a *"friend"* member function. It takes the *output stream typedef* variable (left operand) and referenced variable "node". It returns the "ostrem" typedef variable as the output(right operand). This operator prints the address of the "node", the value of the node, and the address of the left and right children nodes.
  This class doesn't have access to the "BST" class, so we should determine where the "Node" datatype is using "BST::Node".

- **Overloaded Operator <=> with Node for Class "Node"**
  <=> operator is used from <compare> library. This operator is used for comparing an object of the class with another object of the class. It is both the right and left operand, making the comparison operator very easy. This class returns "partial_ordering" class type from <compare> library which admits all six relational operators (==, !=, <, <=, >, >=), but we should define == operator to complete our comparison operators.

- **Overloaded Operator == with Node for Class "Node"**
  == operator checks the equality of two objects of class "Node". It returns a Boolean variable that determines equality.

- **Overloaded Operator <=> with "int" for Class "Node"**
  As I mentioned before, this operator is from <compare> library. It compares the object of class "Node" with an "int" variable and returns, the "partial_ordering" class type.

- **Overloaded Operator == with "int" for Class "Node"**
  == operator checks the equality of an object of the class with an "int" variable. It returns a Boolean variable.

The member functions of the "Node" class have ended. The explanations related to class "BST" will be started next.

- **Member Function “get_root” of Class “BST”**
  This member function gives us referenced access to the private member value “root”. It is a getter function for “root” and it shouldn’t change the “root”. So we cast the variable “root” to a const datatype and define this function as a const member function.

- **Breadth-First Search Algorithm of Class “BST”**
  This is a popular algorithm to apply an action on the entire BST. It is not a very optimized algorithm when we deal with part of the tree. We use it when we want to handle the whole tree and every node of it.
  We use a data type called “queue” from the <queue> library to implement this algorithm. First of all, we check if the “root” member variable has been initialized. Then we push the “root” variable as the first node to the queue. Now we start with a while loop till the queue is empty. We store the first element of the queue in the variable “node” then we delete that element from the queue and take the functionality on the variable “node”. Now we check if the “node” has left or right children to add these nodes to the end of the queue. This cycle will continue until all of the nodes take the functionality.
  The input of this member function is itself a primary class template and acts like a lambda expression. To implement this expression the “std::function” from <functional> library is used. This lambda expression returns nothing and takes the argument type referenced “Node” pointer.

- **Member Function “length” of Class “BST”**
  We implement this function using the “bfs” member function. It takes referenced length as the capture variable, takes the input variable node, and applies the increment functionality on the length variable. “bfs” function will add one in each cycle, and finally the “length” variable will be equal to the number of the nodes.

- **Member Function of “add_node” Class “BST”**
  This function adds a determined value node to the tree. At first, it takes reference of the “root” member variable as the pointer of the “Node”’s pointer datatype. For better understanding, we can assume the “Node*” datatype as a new hypothetical like “QDataType”. At the definition of the “root” member variable, we can substitute our “QDataType”. Then we have
  “$\text{Node} * \text{”} \equiv \text{“QDataType”}$
  - QDataType root;

  Now at the “add_node” member function, we have:

  - QDataType* node { &root };

  This is the basic definition of a pointer variable!

  We continue the explanation. First of all, we check if the root is *nullptr* or not(root by itself is a pointer but we take its pointer as a new data type so we should use “*node” to determine that we want to access “QDataType” which is “Node*”). If it is, we assign a new “Node” block with the determined value to that empty block of memory.

  According to the main property of BST, the right child of a node is greater than its value and the left child of a node is smaller than it. Then we check if this left or right child is empty. If it is empty we find the correct place and we assign a new “Node” pointer to that block of memory, if it is not, we change the “node” variable with this left or right node and repeat this process till we find an empty block of memory. If the found node is equal to the desired node, we don’t add anything and return false, because we cannot add duplicate values.

  We use “*node” in “(*node) -> right” to access the pointer value of that node and accessing to its member variable (left or right).

  At “node = &((*node)->left);” we can substitute our “QDataType” and write “&QDataType”. So we are assigning a referenced “QDataType” to our value “node”.

- **Member Function “find_node” of Class “BST”**
  This member function works like the “add_node” member function. The only difference is that when it reaches a *“nullptr”* node, it figures out that there is no node with that specific value. Otherwise, it checks the left node, if it is smaller than that node or it checks the right node if it is greater than that. If the value of the node is equal to the specific value, we find our desired node and return it.

- **Member Function “find_parrent” of Class “BST”**

  The steps of implementing this member function are very similar to the member function “find_node”. The only difference is that we save the pointer of “QDataType” of the parent of the desired node in each step. When we find the node, we return its parent instead.

- **Member Function “find_successor” of Class “BST”**
  At first, we find the specific node we want to find its successor. If the specific node gets found in the tree, we check the left side node of “node”. If it is empty, the successor of that node is equal to itself. Otherwise, we take the rightmost node of the left node. If the left node hasn’t a right node the successor of the “node” is the left node.

- **Member Function “delete_node” of Class “BST”**
  After we find that specific node, we check the left and right children of that node. If it hasn’t children we delete that node using its pointer value (“*node” is equivalent to “QDataType” which is a pointer “Node*”). There is no other value to assign to that node, so we assign *“nullptr”* to it.
  If “node” has just a child, we take the pointer of that child, delete the pointer of the node, and then assign the pointer of the child to it.
  If the “node” has two children, we should move the value of the successor of the “node” to its value. We shouldn’t change the pointer of this node, because we lost the children of it. So first we find the successor of the node and save the double-pointer of it (“QDataType*”) to the variable “successor”.

If the “successor” has no left child, we can delete its pointer value “QDataType” easily and assign *“nullptr”* to it. If the “successor” has a left child we save the pointer of the left child of the “successor” to the variable “leftChildOfSuccessor” to prevent data lost. Then after we move the “successor” value to the “node” value we can delete the pointer of the successor node and replace the pointer of its left child with it.

- **Overloaded Operator << for Class “BST”**
  I explained the behavior of operator “<<” and its input and output variables on the 2nd page. We print out the “BST” class and its nodes in multiple lines. For better visualization, I use “setw” from <iomanip> library. The general format of the output is set as the same as our sample output. This operator uses the “bfs” member function and iterates over the nodes and takes “&stream” as the capture variable to apply the functionality. As I mentioned before, this operator should be defined as a friend member function.

- **Default Constructor of Class “BST”**
  This member function initializes the member variable “root” with *“nullptr”* when the user of the class doesn’t initialize the object.

- **Copy Constructor of Class “BST”**
  This member function takes another object of class “BST” as input(“bst”) and initializes our object with nodes’ values of the “bst” object. I used the “bfs” member function to iterate on each node of object “bst” and take the “this” object as a capture variable to add each node of the "bst" class from the top of the tree to the “this” object.

- **Move Constructor of Class “BST”**
  This member function takes the referenced “bst” object as the input and initializes our object. After initializing the member variable “root” pointer, we assign *“nullptr”* to the “bst.root” variable to complete the move operation.

- **Initializer List Constructor of Class “BST”**
  This member function takes a list of integer numbers and adds them to the object one by one. The input of this constructor is “std:initializer_list” from the <initializer_list> library, but it is included in this version of the <iostream> library I am working with. To make sure that this library is included, I add the <initializer_list> header to the “bst.h” file.

- **Destructor of Class “BST”**
  This member function destruct and deletes objects of class “BST” at the end of the code. This function iterates through nodes of the object and adds the referenced form of these nodes to the vector “nodes”. Then it iterates through the pointer values of the “nodes” vector and deletes them one by one. At last, it assigns *nullptr* to the only member variable of the class (“root”).

- **Overloaded Pre-Increment Operator of Class “BST”**
  This operator adds one to all the values of nodes and returns the value of the referenced form of the “BST” object. To implement this operator we can use the “bfs” member function and return “*this” as the

- **Overloaded Post-Increment Operator of Class “BST”**
  This operator adds one to all the values of nodes but returns the previous version of the “BST” object. To implement this function we create a copy version of the object before modifying it. Then we call the pre-increment operator to add one to each node. Finally, we return the previous copy version of the “BST” object.

- **Overloaded Assignment Operator of Class “BST”**
  This member function assigns a new tree to the “BST” object. To implement this function, first, we delete the previous nodes of the “BST” object using the destructor of the object. Then we use the “bfs” member function, pass the “BST” object as the capture value and iterate through each node to add values of the “bst” object to the current object. We return the value of the referenced form of the current object. It is used in something like this:

"A = B = C"

- **Overloaded Move Assignment Operator of Class "BST"**
  First, we delete the nodes of the current "BST" object using a destructor. Then we assign the pointer to the "root" value of the "bst" object to the current object. As move assignment should be implemented we reassign "bst.root" with *"nullptr".*

Note1: The comparison operators of class "Node" don't change the member values of this class. So they should be defined as const member functions.

Note2: The "BST" class has only one member value "root" pointer. All the member functions of class "BST" are const member functions because they don't change the member value "root" pointer, except constructors, destructors, "add_node", assignment, and move assignment member functions. "add_node" can add the first node of the tree which is the "root". So it can change the member value "root" pointer. Assignment operators, change the entire tree with its nodes including the "root" pointer. "delete_node" member function can delete the value of the "root" but it doesn't change the "root" pointer.

Note3: To debug the code and improve the performance of the code I used some "cout"s on the entire code. At the end of this homework, I didn't want to remove all these "cout"s one by one. so I defined a debug section at the first of the "bst.cpp" file using macros. I commented (#define DEBUG_MSG) at the end to prevent showing the comments. The "do while" structure of the debug section is for making sure that a semicolon is used at the end of *DEBUG("string");* line.