Claire Dildy Avery Nelson

010870295 010920903

crdildy@uark.edu amn018@uark.edu

Algorithms Assignment 3 Report

Description of implementation

The goal of this assignment was to implement a Binary Search Tree (BST). The BST stores data about the path taken in order so that the user can locate the minimum value quickly. The map is a graph constructed by a series of nodes, each meant to represent a building. Within the graph, an undirected edge between two nodes represents the path between the two nodes.

The BST nodes have five attributes: key of the node, height of the node, metadata of the node, left pointers, and right pointers to children. There are three methods—first, a default constructor for if no attributes are loaded. There should also be a copy constructor as well, and a deconstructor to delete the object and free up memory. The BST includes a root, which is a private attribute. The tree implements seven methods, including the constructor, a deconstructor, a function to find a given key, a function to remove a given key, a function to pop the maximum node, one to pop the minimum node, and one to insert a new node with a given key. The BST implementation for this assignment does allow for the insertion of duplicated values.

Deconstructor

The deconstructor of the BST deallocates the memory of all nodes in the BST. In this program, a helper function is utilized. Per the hints given in the lecture slides, this program successfully implements a deconstructor that clears all nodes in a BST.

```
31  */
32  BST::~BST() {
33      releaseHelper(root);
34  }
35
36  void BST::releaseHelper(BSTNode *node){
37      //null pointer
38      if (node == nullptr)
39      {
40          return;
41      }
42      //clear left
43      releaseHelper(node->left);
44      //clear right
45      releaseHelper(node->right);
46      //release memory of node
47      delete node;
48  }
49
```

Figure 1: Deconstructor

Find method

The Find method of the BST will find the node containing the given value for a key. This implementation utilized a helper method that recursively calls itself to return the desired node. The find helper method has parameters of a given key and node. The find helper method begins with checking the node's contents, and if the node is a null pointer, the method will return a null pointer. Next, the method checks if the node key matches the search key. If so, the method returns that node. If the search key is less than the current node key, then the method recursively calls itself using the left child of the node to search. If the search key is greater than the current node key, the method recursively calls itself using the right child of the node to search. The find method just simply calls on the helper function to provide the desired node containing the given key.

```
*/
BSTNode* BST::find(int key) {
    return findHelper(key, this->root);
}

BSTNode* BST::findHelper(int key, BSTNode *node){
    //if null pointer
    if(node == nullptr){
        return nullptr;
    }
    //if node contains key
    else if(node->key == key){
        return node;
    }
    //if key is less than
    else if(key < node->key){
        return findHelper(key, node->left);
    }
    //if key if greater than
    else{
        return findHelper(key, node->right);
    }
}
```

Figure 2: Find Method

Insert method

The Insert method of the BST will insert a node with a given key and metadata into the BST. In this implementation, a helper method recursively calls itself to insert the node into the appropriate spot in the BST. The insert helper function parameters are a given key, metadata, and node. The helper method begins by checking if the node is null. If it is, then a new node is allocated with the given parameters. When inserting a node, the BST must follow the rules of node <= key < node. If the given key is less than or equal to the current node key, the pointer moves to the left child of the current node and recursively calls the helper method. If the key is greater than the current node key, the pointer moves to the right child of the node and recursively calls the helper method.

```
*/
BSTNode* BST::insert(int key, int meta) {
    return this->root = insertHelper(key, meta, this->root);
}

BSTNode* BST::insertHelper(int key, int meta, BSTNode *node){
    //if node is null
    if(node == NULL){
        //Allocate a new node with data
        node = new BSTNode();
        node->key = key;
        node->meta = meta;
        return node;
    }
    //if key is LESS or EQUAL than node key
    if(key <= node->key){
        //Go to left
        node->left = insertHelper(key, meta, node->left);
    }

    //if key is GREATER than node key
    else{
        //Go to right
        node->right = insertHelper(key, meta, node->right);
    }
    //return the node
    return node;
}
```

Figure 3: Insert Method

Pop Maximum & Pop Minimum methods

The Pop Maximum method of the BST will return the node that contains the maximum value and remove that node from the BST, but it will not release the memory of the node. This code implemented a helper method with two parameters: the current node and the max node. If the current node is null, the method will return a null pointer. If the current node has a right child, it will continue searching for the max value in the next right child. If the current node does not have a right child, the current is the maximum.

```
BSTNode* BST::popMaximumHelper(BSTNode* currNode, BSTNode*& maxNode){
    if(currNode == nullptr){
        return nullptr;
    }

    if(currNode->right){
        currNode->right = popMaximumHelper(currNode->right, maxNode);
    }

    else{
        maxNode = currNode;
        currNode = currNode->left;
    }
    return currNode;
}

BSTNode* BST::popMaximum() {
    //std::cout<<"call to pop max";
    BSTNode* max = nullptr;
    root = popMaximumHelper(root, max);
    return max;
}
```

Figure 4: Pop Maximum Method

The Pop Minimum method of the BST will return the node that contains the minimum value and remove that node from the BST, but it will not release the memory of the node. A helper method with two parameters --- the current node and the minimum node --- is used. The helper method checks if the current node is null, if it has a left child, or if it does not have a left child. The node furthest left will be the minimum node in a BST.

```
*/
BSTNode* BST::popMinimumHelper(BSTNode* currNode, BSTNode*& minNode){
    if(currNode == nullptr){
        return nullptr;
    }

    if(currNode->left){
        currNode->left = popMinimumHelper(currNode->left, minNode);
    }

    else{
        minNode = currNode;
        currNode = currNode->right;
    }
    return currNode;
}

BSTNode* BST::popMinimum() {
    BSTNode* min = nullptr;
    root = popMinimumHelper(root, min);
    return min;
}
```

Figure 5: Pop Minimum Method

Remove Method

The remove method of the BST will remove a node containing the desired key from the tree. This method was the most challenging to implement and was completed by incorporating an additional helper method to remove a node. When removing a node from the BST, several cases must be considered. There is a case when a node has no children, and in that case, the leaf node can be deleted. There is the case of only a left or only a right child. In that case you must replace the deleted node location with the child. Finally, there is the case where a node has two children, and you must decide which child will replace the deleted node.

The removeHelper method has two parameters: the node key and a current node object. This function checks which situation the removal process is in based on whether it has no children, one child, or both children. When it finds a leaf node, it simply deletes that node. When it finds a left child only node, it moves the child up into the current node's position. It does the same operation for a right child only node. If a node has two children, more work is done to ensure the BST remains balanced. An additional helper is used in the last situation. The node with the least value will be moved up to the current node's position. The minChild method returns the leftmost node and sets it equal to the current node's key value. Next, the node position for the vacated node is deleted from the BST.

```
*/
BSTNode* BST::remove(int key) {
    return this->root = removeHelper(key, this->root);
}
```

```
BSTNode* BST::removeHelper(int key, BSTNode* node){
    if(node == nullptr){
        return node;
    }
    //If node contains the given key, then perform the deletion
    if(node->key == key){
       if(node->left == nullptr && node->right ==nullptr){
        delete node;
        return nullptr;
       }
       //if the node has only left child
        else if(node->right == nullptr){
        BSTNode* temp = node->left;
        delete node;
        return temp;
       }
       //if the node has only right child
       else if(node->left == nullptr){
        BSTNode* temp = node->right;
        delete node;
        return temp;
       }
       //Has both children, must find the max between children nodes to r
       else{
        BSTNode* temp = minChild(node->right);
        node->key = temp->key;
        node->right = removeHelper(temp->key, node->right);
       }
    }
    else if(key < node->key){
        node->left = removeHelper(key, node->left);
    }
    else{
        node->right = removeHelper(key, node->right);
    }
    return node;
}
```

```
240  BSTNode* BST::minChild(BSTNode* node){
241      while(node->left){
242          node = node->left;
243      }
244      return node;
245  }
```

Figure 6: Remove Method

Results

This program compiles and implements all the required methods. The program also runs and passes the testing provided by the source code. The homework three review slides and the pseudocode on them offered guidance on this project. Ultimately, it completes the goal of implementing a binary search tree as instructed.