Analysis of Algorithms

BLG 335E

Project 3 Report

İbrahim Karateke
150210705
karateke21@itu.edu.tr

Faculty of Computer and Informatics Engineering
Department of Computer Engineering
Date of submission: 30.12.2023

1. Implementation

In this chapter, we are going to introduce the implementation detail of each function which is implemented in the project. In total, there are two main data structures, namely Binary Search Tree (BST) and Red-Black Tree (RBT).

1.1. Binary Search Tree

A binary search tree (BST) is a data structure used in computer science to organize a set of elements. It is composed of nodes where each node contains a value and two pointers to its left and right children. The BST follows a specific ordering principle which is the value in each node must be greater than all values in its left subtree and less than all values in its right subtree. This inherent property enables efficient search, insertion, and deletion operations. As a result of this ordering, searching for a specific value in a BST can be done by traversing the tree from the root downward, either left or right, based on whether the value being searched for is smaller or larger than the current node's value.

```
1 namespace BST {
2   struct Node {
3     int data;
4     std::string name;
5     Node *parent;
6     Node *left;
7     Node *right;
8   };
9 }
```

As can be seen from the above code snippet, in our data structure our nodes have 5 different attributes which are, **data**, **name**, **parent**, **left** and **right**. In this project, our nodes will be ordered according to their data, which is the value of the population in each city. Similarly, the name is the name of the city. Note that in this project we are going to use previously provided data consisting of city names and their populations.

1.1.1. Constructor

This function is used to create a BST object from **BinarySearchTree** class. In the private section, we have only the **root** attribute and all others are the functions that are related to other public functions. In this function, the root node is assigned to NULL. The code snippet of that function can be seen below.

```
class BinarySearchTree {
      private:
        BST::Node *root;
        ... // will be explained
      public:
        BinarySearchTree() {
          this->root=NULL;
        }
        ... // will be explained
        };
```

1.1.2. Destructor

The **destroy** function recursively frees memory allocated to each node in the BST. Starting at a given node (x), it traverses by calling itself for the left and right children of x. Upon reaching leaf nodes, it deletes x, releasing its memory. The ∼**BinarySearchTree()** destructor triggers this process by calling destroy(this->root), starting from the root and systematically releasing memory for all nodes. This ensures a thorough cleanup of the entire BST.

```
class BinarySearchTree {
      private:
        void destroy(BST::Node* x)
        {
            if(x!=NULL){
            destroy(x->left);
            destroy(x->right);
            delete x;
            }
        }
        ... // will be explained
      public:
        ~BinarySearchTree(){
            destroy(this->root);
        }
        ... // will be explained
        };
```

1.1.3. preorder()

In the private section, there exists a method named **preorderTraverse**. This method is responsible for conducting a recursive preorder traversal of the binary search tree. It takes in parameters: a pointer to a node (representing the current node), an array of pairs (each pair containing a string and an integer), and a reference to an index variable. The method checks if the current node exists (i.e. is not NULL). If the node exists, it stores the node's name and data into the array at the given index, increments the index, and recursively calls itself for the left and right child nodes of the current node. In the public section, there is a method named preorder. This method acts as a public interface to initiate the preorder traversal of the binary search tree. It takes parameters: an array of pairs and an index. Inside this method, it initiates the traversal by calling the preorderTraverse method, starting the traversal from the root node of the tree. The code snippet of that function can be seen below.

```
class BinarySearchTree {
      private:
        void preorderTraverse(BST::Node* node,std::pair<std::string, int> array[],
        ↪  int& index)
        {
          if(node!=NULL){
            array[index].first = node->name;
            array[index].second = node->data;
            index++;
            this->preorderTraverse(node->left,array,index);
            this->preorderTraverse(node->right,array,index);
          }
        }
        ... // will be explained
      public:
        void preorder(std::pair<std::string, int> array [], int index)
        {
            preorderTraverse(this->root,array,index);
        }
        ... // will be explained
        };
```

1.1.4. inorder() and postorder()

Those functions perform very similar operations to what **preorder()** does. The only difference is that the order of calling functions recursively and indexing the array which is desired to be filled during traversing. Hence, inorder and postorder functions traverse the BST inorderly and posorderly, respectively. The code snippets of those functions are given on the next page.

```
class BinarySearchTree {
      private:
        void inorderTraverse(BST::Node* node,std::pair<std::string, int> array[],
        ↪  int& index)
        {
          if(node!=NULL){
            this->inorderTraverse(node->left,array,index);
            array[index].first = node->name;
            array[index].second = node->data;
            index++;
            this->inorderTraverse(node->right,array,index);
          }
        }
        ... // will be explained
      public:
        void inorder(std::pair<std::string, int> array [], int index)
        {
            inorderTraverse(this->root,array,index);
        }
        ... // will be explained
        };
```

```
class BinarySearchTree {
      private:
        void postorderTraverse(BST::Node* node,std::pair<std::string, int> array[],
        ↪  int& index)
        {
          if(node!=NULL){
            this->postorderTraverse(node->left,array,index);
            this->postorderTraverse(node->right,array,index);
            array[index].first = node->name;
            array[index].second = node->data;
            index++;
          }
        }
        ... // will be explained
      public:
        void postorder(std::pair<std::string, int> array [], int index)
        {
            postorderTraverse(this->root,array,index);
        }
        ... // will be explained
        };
```

1.1.5. search()

The **searchTree** function in this Binary Search Tree (BST) class aims to locate a specific value within the tree. Given an integer value pop as input, the function starts its search from the root node of the BST. It iterates through the tree by comparing the provided value pop with the data stored in each node. If the value matches the data in a particular node, the function returns a pointer to that node, indicating a successful find. Throughout the process, the function navigates through the tree by moving left or right based on the comparison of pop with the current node's data. If pop is smaller than the current node's data, the function moves to the left child of the node; otherwise, it moves to the right child. This process continues until it finds a node with the matching value or until it reaches a point where there are no more nodes to explore, indicating that the value pop is not present in the BST. Finally, the function returns a pointer to the node containing the sought-after value if it exists within the BST. If the value is not found in any node along the traversal path, the function returns a nullptr, signifying that the value pop is not present within the tree. Essentially, this function implements a search algorithm within the BST, efficiently navigating through the tree's structure to locate a specific value. The code snippet of this function can be seen below.

```
class BinarySearchTree {
      private:
        ... // will be explained
      public:
        BST::Node *searchTree(int pop) {
            BST::Node *current = root;
            while (current != NULL && current->data != pop) {
                if (pop < current->data) {
                    current = current->left;
                } else {
                    current = current->right;
                }
            }
        return current;
        }
        ... // will be explained
        };
```

1.1.6. successor()

The **successor** function is a part of the Binary Search Tree (BST) class and is used to find the node that comes after a given node, myNode, in the tree based on an inorder traversal sequence. The function uses a conditional approach to handle different scenarios. If myNode has a right subtree, the function moves to the leftmost node within this subtree by continuously traversing left from **myNode->right** until it reaches a node without a left child, which becomes the successor to myNode. If myNode does not have a right subtree, the function traverses upwards through the tree to find the ancestor node where myNode would be located in the left subtree during an inorder traversal. Throughout this process, the function manages two pointers: ancestor for tree traversal and successor to keep track of the potential succeeding node. The function returns the identified successor node, or nullptr if myNode is empty or if there isn't a node succeeding myNode in the tree. Overall, this function effectively determines the node that comes after a given node in an inorder traversal sequence of the BST.

```
class BinarySearchTree {
      private:
        ... // will be explained
      public:
        BST::Node *successor(BST::Node *myNode) {
           if (myNode == nullptr) {
                return nullptr;
            }
            if (myNode->right != nullptr) {
                BST::Node *current = myNode->right;
                while (current->left != nullptr) {
                    current = current->left;
                }
                return current;
            }
            BST::Node *successor = nullptr;
            BST::Node *ancestor = root;
            while (ancestor != nullptr && ancestor != myNode) {
                if (myNode->data < ancestor->data) {
                    successor = ancestor;
                    ancestor = ancestor->left;
                } else {
                    ancestor = ancestor->right;
                }
            }
            return successor;
        }
        return current;
        }
        ... // will be explained  };
```

1.1.7. predecessor()

The **predecessor** function in a Binary Search Tree (BST) class is a method aimed at finding the node that precedes a given node, myNode, in an inorder traversal of the tree. It operates through two distinct scenarios. If myNode possesses a left subtree, the function descends this subtree to locate the rightmost node, representing the largest value smaller than myNode, effectively serving as its predecessor. On the other hand, if myNode lacks a left subtree, the function traverses upwards from the assumed root node, seeking the first ancestor node where myNode would fit into the right subtree during an inorder traversal. Throughout this traversal, it maintains two pointers: "ancestor" for tree traversal and "predecessor" to track the potential preceding node. The function updates the predecessor whenever it encounters an ancestor node containing myNode in its right subtree. Once the traversal concludes, the function returns the identified predecessor node. However, if myNode is nullptr (indicating an empty node) or if myNode is the minimum value node without a predecessor, the function returns nullptr. Finally, this function effectively determines the node preceding a given myNode in an inorder traversal sequence of the BST. The code snippet of this function can be seen below.

```
class BinarySearchTree {
      private:
        ... // will be explained
      public:
        BST::Node *predecessor(BST::Node* myNode) {
              if (myNode == nullptr) {
                  return nullptr;
              }
              if (myNode->left != nullptr) {
                  BST::Node *current = myNode->left;
                  while (current->right != nullptr) {
                      current = current->right;
                  }
                  return current;
              }
              BST::Node *predecessor = nullptr;
              BST::Node *ancestor = root;
              while (ancestor != myNode) {
                  if (myNode->data > ancestor->data) {
                      predecessor = ancestor;
                      ancestor = ancestor->right;
                  } else {
                      ancestor = ancestor->left;
                  }
              }
              return predecessor;
        } ... // will be explained };
```

1.1.8. insert()

The **insert** function in this Binary Search Tree (BST) class takes a city name and its population as arguments to add a new node into the BST. It invokes the inserter function, assigning the result to the root of the tree. The inserter function, designed to facilitate recursive insertion, takes in a node, an integer data value, and a string name. It checks if the provided node is NULL. If so, it creates a new node with the given data and name, initializing its left, right, and parent pointers to NULL. If the node exists, it compares the provided data with the current node's data to determine whether to navigate left or right. For data less than the current node's data, it recursively calls inserter for the left subtree, updating the parent pointer accordingly. Conversely, for data greater than or equal to the current node's data, it performs the recursive call for the right subtree, also updating the parent pointer. This recursive process continues until an appropriate position is found for insertion, and the updated or new node is returned to maintain the BST structure. The function effectively ensures that nodes are inserted in accordance with the BST property, where values to the left of a node are smaller and values to the right are larger, maintaining the order of the tree. The code snippet of this function can be seen below.

```
class BinarySearchTree {
      private:
        ... // will be explained
        BST::Node* inserter(BST::Node* node, int data, const std::string& name) {
              if (node == NULL) {
                  node = new BST::Node;
                  node->data = data;
                  node->name = name;
                  node->left = NULL;
                  node->right = NULL;
                  node->parent = NULL;
              } else if (data < node->data) {
                  node->left = inserter(node->left, data, name);
                  node->left->parent = node;
              } else {
                  node->right = inserter(node->right, data, name);
                  node->right->parent = node;
              }
              return node;
          }
      public:
        void insert(std::string city, int population)
        {
          this->root = inserter(this->root, population, city);
        }
        ... // will be explained
        };
```

1.1.9. deleteNode()

The **deleteNode** function within the BinarySearchTree class provides an interface for removing a node based on its population value from the tree. This function is accessible to external users and acts as a gateway to the internal deletion mechanism. When invoked, deleteNode internally calls a private member function named **deleter**, passing in the tree's root and the population value to be deleted. The deleter function is responsible for the actual deletion process and operates recursively. It navigates through the tree by comparing the provided population value with the data stored in each node. This comparison determines whether to traverse left or right in the tree. Upon locating the node to be deleted, deleter handles various scenarios: nodes with no children, nodes with only one child, and nodes with both left and right children. If the node to be deleted has no children or only one child, deleter adjusts pointers and deletes the node accordingly, ensuring the tree structure is maintained. However, when the node has both left and right children, deleter finds the successor node - the smallest node in the right subtree of the node to be deleted. It copies the successor's data to the current node, effectively replacing the current node with the successor node. Finally, it deletes the successor node while preserving the BST properties. Finally, the deleteNode function encapsulates this intricate deletion process, enabling users to simply specify the population value they want to remove from the tree. By calling the deleter function internally, it orchestrates the deletion of nodes while maintaining the integrity of the Binary Search Tree structure. The code snippet of this function can be seen below.

```
class BinarySearchTree {
      private:
        ... // will be explained
        BST::Node* deleter (BST::Node* root, int pop){
        if (root == NULL)
            return root;
        if (root->data > pop) {
            root->left = deleter(root->left, pop);
            return root;
        }
        else if (root->data < pop) {
            root->right = deleter(root->right, pop);
            return root;
        }
        if (root->left == NULL) {
            BST::Node* temp = root->right;
            delete root;
            return temp;
        }
        else if (root->right == NULL) {
            BST::Node* temp = root->left;
            delete root;
```

```
23              return temp;
24          }
25          else {
26              BST::Node* succParent = root;
27              BST::Node* succ = root->right;
28              while (succ->left != NULL) {
29                  succParent = succ;
30                  succ = succ->left;
31              }
32              if (succParent != root)
33                  succParent->left = succ->right;
34              else
35                  succParent->right = succ->right;
36              root->data = succ->data;
37              delete succ;
38              return root;
39          }
40          }
41        public:
42          void deleteNode(int pop)
43          {
44              deleter(this->root,pop);
45          }
46          ... // will be explained
47          };
```

1.1.10. getHeight()

The **getHeight** function in the BinarySearchTree class serves as an interface to retrieve the height of the tree. This method calculates the height of the tree by calling a private member function named **heightCalculator**. Internally, heightCalculator is a recursive function that takes in a node pointer as its argument. It recursively computes the height of the tree by traversing through each level of the tree, starting from the root. If the provided node is NULL, indicating an empty subtree, the function returns 0. When processing a non-empty node, heightCalculator recursively calls itself for the left and right subtrees. It retrieves the heights of the left and right subtrees and determines the maximum between these heights using the std::max function. This step identifies the longer subtree, representing the maximum height from either the left or right side. The height of the current node is then calculated as 1 plus the maximum height between its left and right subtrees. This computation accounts for the height of the current node in the total height calculation of the tree. In the getHeight function, the result from heightCalculator is returned after subtracting 1. This adjustment is made to account for the height calculation starting from 1 at the root node, aligning with the convention where an empty tree has a height of -1. Therefore, subtracting 1 from the result ensures that an empty tree returns a height of -1, while a non-empty tree returns its actual height as

calculated by heightCalculator. Finally, the getHeight function provides an easy-to-use method for users to obtain the height of the Binary Search Tree while internally employing the heightCalculator to perform the recursive height computation based on the tree's structure. The code snippet of this function can be seen below

```
class BinarySearchTree {
      private:
        ... // will be explained
        int heightCalculator(BST::Node* root)
        {
            if(root==NULL){
                return 0;
            }
                int Left=heightCalculator(root->left);
                int Right=heightCalculator(root->right);
                return 1+std::max(Left,Right);
        }
      public:
        int getHeight()
        {
            return heightCalculator(this->root)-1;
        }
        ... // will be explained
        };
```

1.1.11. getMinimum()

The **getMinimum** function in the Binary Search Tree class aims to find and return the node containing the minimum value in the entire tree. It starts by checking if the tree is empty by examining whether the root node is NULL. If the tree is empty, indicating no nodes exist, the function returns NULL as there is no minimum value to retrieve. When the tree isn't empty, the function initializes a pointer named current and sets it to point at the root node. It then enters a loop that iterates as long as the left child of the current node isn't NULL. This loop effectively navigates down the left side of the tree, moving from node to node until it reaches the leftmost node, which is the node containing the minimum value in a Binary Search Tree. By consistently moving to the left child of each node, the function progresses down the tree until it arrives at the node with no left child. At this point, it signifies that the current node holds the minimum value in the tree, as there can't be a smaller value within a BST due to its structure. Finally, the function returns this node containing the minimum value. Overall, the getMinimum function provides a straightforward and efficient approach to finding the node with the smallest value within a Binary Search Tree by iteratively navigating down the left side of the tree until it reaches the node with the minimum value. The code snippet of this function can be seen on the next page.

```
class BinarySearchTree {
      private:
        ... // will be explained
      public:
        BST::Node* getMinimum()
        {
            if(root==NULL)
                return NULL;
            BST::Node* current = root;
            while(current->left!=NULL)
                current=current->left;
            return current;
        }
        ... // will be explained
        };
```

1.1.12. getMaximum()

The **getMaximum** function within the Binary Search Tree class aims to locate and return the node that contains the maximum value in the entire tree. It starts by checking if the tree is empty by verifying if the root node is NULL. If the tree is empty (i.e., no nodes exist), the function returns NULL because there is no maximum value to retrieve. When the tree isn't empty, the function initializes a pointer called current and sets it to point at the root node. It then enters a loop that continues as long as the right child of the current node isn't NULL. This loop effectively traverses down the right side of the tree, moving from node to node until it reaches the rightmost node, which contains the maximum value in a Binary Search Tree. By consistently moving to the right child of each node, the function progresses down the tree until it reaches the node without a right child. This indicates that the current node holds the maximum value in the tree, as no larger value can exist within a BST due to its structure. Ultimately, the function returns this node, which contains the maximum value. In summary, the getMaximum function provides a straightforward and efficient approach to finding the node with the largest value within a Binary Search Tree by iteratively navigating down the right side of the tree until it reaches the node with the maximum value. The code snippet of this function can be seen on the next page.

```
class BinarySearchTree {
      private:
        ... // will be explained
      public:
        BST::Node* getMaximum()
        {
            if(root==NULL)
                return NULL;
            BST::Node* current = root;
            while(current->right!=NULL)
                current=current->right;
            return current;
        }
        ... // will be explained
        };
```

1.1.13. getTotalNodes()

The **getTotalNodes** function within the BinarySearchTree class provides a means to retrieve the total number of nodes present in the entire tree. It utilizes a private member function called **Counter** to calculate this count recursively. Internally, the Counter function is recursive and takes a node pointer, root, as its argument. It starts by checking if the provided node is NULL, indicating an empty subtree. If so, it returns 0, denoting that there are no nodes in this part of the tree. When encountering a non-empty node, the Counter function counts the current node (incrementing by 1) and recursively calls itself for the left and right subtrees of the current node. This recursive process traverses through the entire tree, summing up the counts of nodes in each subtree and accumulating them to compute the total number of nodes in the entire Binary Search Tree. The getTotalNodes function serves as an interface for users to obtain the total node count of the tree. It calls the Counter function, passing the root of the tree as an argument, and retrieves the overall count returned by Counter, providing a straightforward method for external access to the total node count functionality. In summary, getTotalNodes employs the Counter recursive function to traverse the entire tree, counting nodes at each step and summing up these counts to determine and return the total number of nodes present in the Binary Search Tree. The code snippet of this function can be seen on the next page.

```
class BinarySearchTree {
      private:
        ... // will be explained
        int Counter(BST::Node* root)
        {
            if (root==NULL)
                return 0;

            return 1+ Counter(root->left)+ Counter(root->right);
        }
      public:
        int getTotalNodes()
        {
            return Counter(this->root);
        }
        ... // will be explained
        };
```

1.2. Red-Black Tree

Red-Black Tree is a specialized type of binary search tree designed to maintain balance during operations like insertion, deletion, and searching. It's structured to prevent the tree from becoming heavily skewed, which could otherwise slow down these operations. Each node in a Red-Black Tree is augmented with an additional attribute denoting its color—either red or black. Four fundamental rules define the properties of these trees: the root node is always black, red nodes cannot have red children, every path from a node to its descendant leaves contains the same number of black nodes, and these properties ensure a balanced structure. When nodes are added or removed, the tree may temporarily violate these rules. To restore balance, the tree undergoes various balancing operations, such as rotations and color changes. This self-balancing feature guarantees efficient lookup, insertion, and deletion operations with a maximum time complexity of O(log n).

```
namespace BST {
  struct Node {
    int data;
    std::string name;
    Node *parent;
    Node *left;
    Node *right;
    int color;
  };
}
```

As can be seen from the code snippet which is on the previous page, the node structure is the same as with BST's node structure. The only addition is made by adding color attribute. Note that while **red nodes** are denoted by **1**, **black nodes** are denoted by **0** in this data structure.

Note that in this data structure some functions which does not cause any kind of a change in the tree (i.e. deletion, insertion...) is not changed. For example, getMinimum function returns the minimum node in both trees. However, it returns a BST node in Binary Search Tree, whereas it returns a RBT node in Red-Black Tree. Except for some minor adjustments like changing BSTs to RBT in both inside the function and in inputs of the function, no change is made. The logic and the algorithm that lies on this logic are the same for each function and its helper function which exists in both classes. Those functions can be listed as:

- Constructor
- Destructor
- preorder
  - preorderTraverse
- inorder
  - inorderTraverse
- postorder
  - postorderTraverse
- searchTree
- successor
- predecessor
- getHeight
  - heightCalculator
- getMinimum
- getMaximum
- getTotalNodes
  - Counter

The code snippets of those functions which are listed above can be seen in the BST chapter.

1.2.1. insert()

The **insert** function initiates the addition of a new node into a Red-Black Tree by creating a new node z with specified data and name attributes. It sets the color of this new node to red, indicating the node's initial state. The function then begins traversing the existing tree structure to find the appropriate location for the insertion. It moves through the tree nodes, comparing data values to determine the placement of the new node z. During this traversal, it keeps track of the current node x and its parent y. Once the suitable position for insertion is identified, the new node z is linked to its parent y either as a left or right child based on its data value. In the case where the tree was initially empty, z becomes the root node. After establishing the new node's placement, the function calls **inserterRBT** to handle any potential violations of Red-Black Tree properties that might have occurred due to this insertion. The inserterRBT function assesses the tree structure around the newly inserted node z and makes necessary adjustments to ensure the tree adheres to Red-Black Tree rules, specifically focusing on maintaining the constraints related to node colors and structural integrity. The iterative process in inserterRBT continuously checks and rectifies violations by examining the colors and positions of nodes relative to their parents and grandparents. It performs rotations and color changes on nodes to restore balance and uphold the Red-Black Tree properties, preventing any violations that might disrupt the tree's structure.

The **leftRotate** function facilitates a leftward reconfiguration around a designated node x. It involves reassigning pointers to properly rotate the tree: it sets y as the right child of x and updates the right child of x to be the former left child of y. By ensuring that parent pointers are appropriately adjusted and handling the case where x might have been the root, the function concludes by positioning y as the new root of the subtree, with x as its left child.

On the other hand, the **rightRotate** function performs a rightward transformation around a specified node y. Similar to the left rotation, it rearranges pointers to effectuate the rotation: x becomes the left child of y, and the former right child of x becomes the left child of y. It then manages parent pointers, potentially updating the root if y was the original root, and finalizes the rotation by setting x as the new root of the subtree with y as its right child.

Finally, the insert function ensures that the newly added node is correctly positioned within the tree and that the Red-Black Tree maintains its balanced structure, optimizing it for efficient search, insertion, and deletion operations while following the specific rules governing its organization and color configurations. The code snippets of those functions on the next pages.

```
void insert(const std::string& name,int data) {
        RBT::Node *z = new RBT::Node;
        z->data = data;
        z->name = name;
        z->left = NULL;
        z->right = NULL;
        z->color = 1;
        RBT::Node *y = NULL;
        RBT::Node *x = root;
        while (x != NULL) {
            y = x;
            if (z->data < x->data) {
                x = x->left;
            } else {
                x = x->right;
            }
        }
        z->parent = y;
        if (y == NULL) {
            root = z;
        } else if (z->data < y->data) {
            y->left = z;
        } else {
            y->right = z;
        }
        inserterRBT(z);
    }
```

```
    void inserterRBT(RBT::Node *z) {
        while (z != root && z->parent->color == 1) {
            if (z->parent == z->parent->parent->left) {
                RBT::Node *y = z->parent->parent->right;
                if (y != NULL && y->color == 1) {
                    z->parent->color = 0;
                    y->color = 0;
                    z->parent->parent->color = 1;
                    z = z->parent->parent;
                } else {
                    if (z == z->parent->right) {
                        z = z->parent;
                        leftRotate(z);
                    }
                    z->parent->color = 0;
                    z->parent->parent->color = 1;
                    rightRotate(z->parent->parent);
                }
            } else {
                RBT::Node *y = z->parent->parent->left;
                if (y != NULL && y->color == 1) {
                    z->parent->color = 0;
                    y->color = 0;
                    z->parent->parent->color = 1;
                    z = z->parent->parent;
                } else {
                    if (z == z->parent->left) {
                        z = z->parent;
                        rightRotate(z);
                    }
                    z->parent->color = 0;
                    z->parent->parent->color = 1;
                    leftRotate(z->parent->parent);
                }
            }
        }
        root->color = 0;
    }
```

```
void leftRotate(RBT::Node *x)
{
    RBT::Node *y = x->right;
    x->right = y->left;
    if (y->left != NULL) {
        y->left->parent = x;
    }
    y->parent = x->parent;
    if (x->parent == NULL) {
        root = y;
    } else if (x == x->parent->left) {
        x->parent->left = y;
    } else {
        x->parent->right = y;
    }
    y->left = x;
    x->parent = y;
}
```

```
void rightRotate(RBT::Node *y)
{
    RBT::Node *x = y->left;
    y->left = x->right;
    if (x->right != NULL) {
        x->right->parent = y;
    }
    x->parent = y->parent;
    if (y->parent == NULL) {
        root = x;
    } else if (y == y->parent->left) {
        y->parent->left = x;
    } else {
        y->parent->right = x;
    }
    x->right = y;
    y->parent = x;
}
```

1.2.2. deleteNode()

Note that this function takes an integer as an input. Hence, if there exists a node with that value this node will be deleted. In order to find this node, **searchTree** function is called. Afterwards, the returned value of this function is sent to the **deleter** function. The function begins by checking if the given node (z) exists. If it's a null pointer (nullptr), indicating the absence of a node, the function exits immediately. y is initialized as z, representing the node to be deleted. x is set to nullptr, acting as a temporary placeholder for a node. y_original_color stores the color of node y before any changes are made.

The function considers three scenarios based on the children of node z:

- If z lacks a left child, it designates the right child (z->right) as the replacement (x). Subsequently, it performs a **transplant** operation, replacing z with its right child.
- If z has no right child, it designates the left child (z->left) as x and replaces z with its left child through a **transplant** operation.
- When z has both left and right children, it finds the successor node (y) by searching the minimum node in the right subtree of z using the **minimumNode** function. y becomes the replacement for z.

It updates pointers to replace z with y and reconfigures the parent-child relationships to reflect the changes in the tree structure. and it adjusts y's pointers to link to the correct subtrees and sets y's color to match z's color to maintain color properties. If the original color of the deleted node (y) was black y_original_color == 0, it may violate the black height property. **fixDelete** function is called to correct any potential violations by rebalancing the tree.

The **transplant** function is responsible for replacing one node with another while preserving the tree's structure and relationships. This function works by adjusting pointers to ensure the proper replacement occurs.

When transplant is invoked, it takes two parameters: u, the node to be replaced, and v, the node that will take u's position. Initially, it identifies whether the node to be replaced, u, is the root of the tree. If u is indeed the root (confirmed by checking if u has no parent), v is directly set as the new root of the entire tree. Otherwise, if u has a parent, the function checks if u is a left child or a right child of its parent. If u is a left child, v takes its place as the left child of u's parent. Conversely, if u is a right child, v becomes the new right child of u's parent. Additionally, regardless of whether v is nullptr or a valid node, v's parent pointer is updated to point to u's original parent. This step ensures that after the replacement, the node v correctly acknowledges its parent within the tree structure.

The **fixDelete** function is responsible for the maintenance of a Red-Black Tree's integrity, ensuring that its fundamental properties remain intact. It operates within a while loop, systematically addressing potential violations that might arise due to the

removal of a node and takes careful steps to restore the tree to a valid Red-Black state. The function's first phase involves a series of checks to determine whether the node x is a valid candidate for corrective actions. It evaluates whether x is not the root node, isn't a null pointer, and that its color is black—highlighting the critical conditions required to proceed with the fix. Subsequently, fixDelete delves into distinguishing between scenarios where x is a left child or a right child of its parent node. For the case when x is identified as a left child, the function explores the state of its sibling (w). If w is red, indicating a potential violation, the function initiates color adjustments and rotations to rectify the imbalance. In cases where w is black or exhibits specific color configurations among its children, the function meticulously orchestrates rotations and color modifications, aligning with the Red-Black Tree's rules. These adjustments aim to ensure that the tree's properties regarding node colors and structure are meticulously maintained. When x represents a right child, a mirrored approach is adopted, scrutinizing the sibling (w) in the opposite direction. The function then undergoes a similar sequence of evaluations and corrective measures, recalibrating the tree to ensure its adherence to Red-Black Tree principles. Throughout its execution, fixDelete strategically manages the tree's structure, handling rotations and color alterations to reestablish the Red-Black Tree's properties post-deletion. Finally, it concludes by affirming that the color of the current node (x) is set to black. All code snippets of the function which are mentioned for this operation can be seen below.

```
void deleteNode(int data)
{
    RBT::Node* k= searchTree(data);
    deleter(k);
}
```

```
void deleter(RBT::Node* z) {
        if (z == nullptr)
            return;

        RBT::Node* y = z;
        RBT::Node* x = nullptr;
        int y_original_color = y->color;

        if (z->left == nullptr) {
            x = z->right;
            transplant(z, z->right);
        } else if (z->right == nullptr) {
            x = z->left;
            transplant(z, z->left);
        } else {
            y = minimumNode(z->right);
            y_original_color = y->color;
            x = y->right;
```

```
19
20              if (y->parent == z) {
21                  if (x != nullptr)
22                      x->parent = y;
23              } else {
24                  transplant(y, y->right);
25                  y->right = z->right;
26                  if (y->right != nullptr)
27                      y->right->parent = y;
28              }
29              transplant(z, y);
30              y->left = z->left;
31              if (y->left != nullptr)
32                  y->left->parent = y;
33              y->color = z->color;
34          }
35
36          if (y_original_color==0){
37              fixDelete(x);
38          }
39
40          delete z;
41      }
```

```
1   RBT::Node* minimumNode(RBT::Node* x)
2        {
3           while (x->left != nullptr) {
4               x = x->left;
5           }
6           return x;
7       }
```

```
1   void transplant(RBT::Node* u, RBT::Node* v)
2       {
3           if (u->parent == nullptr) {
4               root = v;
5           } else if (u == u->parent->left) {
6               u->parent->left = v;
7           } else {
8               u->parent->right = v;
9           }
10          if (v != nullptr) {
11              v->parent = u->parent;
12          }
13      }
```

```
void fixDelete(RBT::Node* x) {
        while (x != root && x != nullptr && x->color == 0) {
            if (x == x->parent->left) {
                RBT::Node* w = x->parent->right;
                if (w->color == 1) {
                    w->color = 0;
                    x->parent->color = 1;
                    leftRotate(x->parent);
                    w = x->parent->right;
                }
                if ((w->left == nullptr || w->left->color == 0) &&
                    (w->right == nullptr || w->right->color == 0)) {
                    w->color = 1;
                    x = x->parent;
                } else {
                    if (w->right == nullptr || w->right->color == 0) {
                        if (w->left != nullptr)
                            w->left->color = 0;
                        w->color = 1;
                        rightRotate(w);
                        w = x->parent->right;
                    }
                    w->color = x->parent->color;
                    x->parent->color = 0;
                    if (w->right != nullptr)
                        w->right->color = 0;
                    leftRotate(x->parent);
                    x = root;
                }
            } else {
                RBT::Node* w = x->parent->left;
                if (w->color == 1) {
                    w->color = 0;
                    x->parent->color = 1;
                    rightRotate(x->parent);
                    w = x->parent->left;
                }
                if ((w->right == nullptr || w->right->color == 0) &&
                    (w->left == nullptr || w->left->color == 0)) {
                    w->color = 1;
                    x = x->parent;
                } else {
                    if (w->left == nullptr || w->left->color == 0) {
                        if (w->right != nullptr)
                            w->right->color = 0;
                        w->color = 1;
                        leftRotate(w);
                        w = x->parent->left;
```

```
                    }
                    w->color = x->parent->color;
                    x->parent->color = 0;
                    if (w->left != nullptr)
                        w->left->color = 0;
                    rightRotate(x->parent);
                    x = root;
                }
            }
        }
        if (x != nullptr)
            x->color = 0;
    }
```

2. Differences between BST and RBT

Both data structures lie on the same principle which is the fact that a node's left child must be smaller than the node, and the right child of this node must be greater than or equal to the node. Rather than this Red-Black Tree has some rules in order to preserve its balance. Those rules can be listed as:

- Every node has a color either red or black.
- The root of the tree is always black.
- There are no two adjacent red nodes (A red node cannot have a red parent or red child).
- Every path from a node (including root) to any of its descendants NULL nodes has the same number of black nodes.
- Every leaf (e.i. NULL node) must be colored BLACK.

Thus, those rules help RBT to have a more balanced structure compared to BST. For example, suppose we are creating both trees by inserting n numbers one by one. If those input numbers are in ascending order, in BST we will have a very similar structure to the linked list, whereas in RBT we will have a balanced tree. After this operation, if we want to perform a searching operation for the greatest number, in BST it must be traversed the whole structure until we reach the last element (i.e leaf) which is the greatest number. However, in RBT, since we have a balanced structure it will take less time to find the desired node. In other words, the order of the data may cause the BST to have time complexity in average $O(\log n)$, but it can degrade to $O(n)$ in the worst case for skewed trees. However, due to their enforced balance, RBT has an average time complexity of $O(\log n)$, providing more predictable performance compared to standard BSTs.

	Population1	Population2	Population3	Population4
RBT	21	24	24	16
BST	835	13806	12204	65

Table 2.1: Tree Height Comparison of RBT and BST on input data.

As can be seen in the above table, the height of trees is given in both data structures in different types of input after the insertion of whole data and one deleting operation. The data is inserted one by one into the trees. In particular, Population 2 is sorted in ascending order with respect to the population of the cities. Population 3 is substantially similar to Population 2. Furthermore, Population 1 is sorted in descending order and Population 4 is in random order. Although the order of the input is different in every dataset, the height of RBT is bounded by 27 in each scenario. However, BST has major differences in those. As we previously explained, linking the data which is given in ascending order causes BST to be formed skewed.

3. Maximum height of the RBTree

A Red-Black Tree n internal nodes can have a maximum height of $2\log(n)$.

The proof of this statement can be done as follows.

Let $h(x)$ be the height of any node x and, $bh(x)$ is the black height of x. Note that black height is the number of black nodes on any simple path from a node x (not including it) to a leaf.

Firstly, we need to prove the following statements first:

1. A subtree rooted at any node x has at least $2^{bh(x)} - 1$
2. Any node x with height $h(x)$ has $bh(x) \geq \frac{h(x)}{2}$

We are going to prove the first statement by the method of induction. The base case will be when x is 0 i.e., x is a leaf. According to the statement, the number of internal nodes is $2^0 - 1 = 0$. Since x is a leaf, this statement is true in the base case.

Now, consider a node x with two children l and r.

Let $bh(x) = b$ Now if the color of the child is red, then its black height will also be b. However, if the color of the child is black, then its black height will be $b - 1$.

According to the inductive hypothesis, a child must have at least $2^{b-1} - 1 = 2^{bh(x)} - 1$ internal nodes. We have assumed that the inductive hypothesis is true for the child, now we need to show that it is also true for the parent i.e., node x and hence completing the proof.
The node x must have at least 1 + the least number of internal nodes that can be present on the right child + the least number of internal nodes that can be present on the left child.

Hence, The node x must have at least $2^{bh(l)-1} + 2^{bh(r)-1} + 1$ internal nodes.

Internal nodes, $x \geq 2^{bh(x)-1} + 2^{bh(x)-1} + 1$

$x \geq 2 \times (2^{bh(x)-1}) + 1$

$x \geq 2 \times (2^{bh(x)}) + 1$

We assumed it to be true for the child and it is also true for the node x, so the statement is proved.

Let's prove the second statement.

Since the leaves are black and there can't be two consecutive red nodes, so the number of red nodes can't exceed the number of black nodes on any simple path from a node to a leaf. Therefore, we can also say that at least half of the nodes on any simple path from a root to a leaf, not including the node, must be black.

This means $bh(x) \geq \frac{h(x)}{2}$

Therefore, the second statement is also true.

According to the second statement, $bh(root) \geq \frac{h(x)}{2}$, where h is the height of the tree.

Using statement 1

$n \geq 2^{bh(root)} - 1$

$n \geq 2^{\frac{h}{2}} - 1$, since $bh(root) \geq \frac{h(x)}{2}$

$n + 1 \geq 2^{\frac{h}{2}}$

$\log(n + 1) \geq \frac{h}{2}$

$h \leq 2 \times \log(n + 1)$

Previously we mentioned that the height of RBT is bounded by 27. In given datasets, we have 13807 populations. If we apply our proof on this:

$n = 13807$, and $2 \times \log_2(13807 + 1) = 27.506$

$h \leq 27.506$

4. Big-o complexity for operations

4.1. Binary Search Tree (BST)

4.1.1. Searching

Average Case: $O(\log n)$
Worst Case: $O(n)$
In an unbalanced BST, where nodes form a linear chain, searching can degrade to $O(n)$ complexity as each node has only one child, making it effectively a linked list. The worst-case scenario occurs when the tree is skewed, and the search operation needs to traverse through all nodes along the unbalanced path.

4.1.2. Insertion

Average Case: $O(\log n)$
Worst Case: $O(n)$
Similar to searching, if the BST becomes unbalanced due to skewed insertion patterns, the tree can degenerate into a linked list. This scenario happens when nodes are inserted in sorted or reverse-sorted order, leading to linear chains and a worst-case complexity of $O(n)$.

4.1.3. Deletion

Average Case: $O(\log n)$
Worst Case: $O(n)$
Deletion's worst-case scenario aligns with the unbalanced tree scenario, which can occur when deleting nodes that would cause the tree to become unbalanced. This unbalancing can lead to the tree's degeneration into a linked list, requiring traversal through all nodes for deletion, resulting in $O(n)$ complexity.

4.1.4. Walking

Inorder, Preorder, Postorder: $O(n)$
Traversal involves visiting each node exactly once. In all three cases, as every node needs to be visited, the time complexity is$O(n)$.

4.2. Red-Black Tree (RBT)

4.2.1. Searching

Average Case: $O(\log n)$
Worst Case: $O(\log n)$
Red-Black Trees maintain balanced properties, ensuring the maximum height remains logarithmic. Consequently, the worst-case complexity for searching in RBTree remains O(log n) due to the balanced nature of the tree.

4.2.2. Insertion

Average Case: $O(\log n)$
Worst Case: $O(\log n)$
RBTree insertion involves balancing operations, but the worst-case complexity remains logarithmic due to the tree's ability to rebalance efficiently after insertions.

4.2.3. Deletion

Average Case: $O(\log n)$
Worst Case: $O(\log n)$
Similar to insertion, RBTree deletion involves rebalancing, maintaining the tree's balanced structure. Therefore, the worst-case complexity for deletion remains logarithmic.

4.2.4. Walking

Inorder, Preorder, Postorder: $O(n)$
Traversal in RBTree still involves visiting every node once, resulting in a time complexity of $O(n)$ for any traversal type.