Mohammed Mamun
CSCI 377
Project 2: Red-Black Tree and B-Tree

Red-Black Tree is a self-balancing Binary Search Tree (BST), where every node follows these four properties. First, every node has a color, which is either red or black. Second, root and NIL leaves are always black. Third, a red node can not have a red child or parent; it has to be black. Lastly, every simple path from root to null has the same number of black nodes. BST has a guaranteed height of O(logn) for n number of nodes. Red black trees provides a way to keep a tree balanced and the basic idea is that all the nodes are labeled either red or black. When writing the code, the color can be represented as a boolean value that is set to either true or false, true representing red and false representing black, for example. In my case, I'm using an integer for the color that I've defined in an enum. A zero represents red and one represents black. Everytime we add something to the tree, such as another node we have to make sure the new tree doesn't violate the rules listed earlier. Depending on the violation we would have to change a few things, such as flipping colors and rotating the children and parent nodes.

My code contains five major public functions that insert, delete, delete all, search, and print. I created some helper functions that are private to the RBTree class and will assist the major functions into getting the needed result.

Node* RBTree::RBTInsert(int key);
The insert function will insert a new node with the key that was passed in. It will return a pointer to that new node. First, we create a new node instance and assign the key value. The default color is red, but we will change that later, as needed. With this node, we will call a helper function that will insert the node into the correct location of the tree. We recursively iterate from the beginning of the tree starting at root and will go down the tree depending on the new node's key. If the key is greater than the current node we're looking at in the tree, we will go right, if it's less than, we will go left. All the way until we find the correct spot. This function will return the pointer to the new node and the original function will continue to correct the balancing of the tree. Here, we will start at the bottom of the new node and will iterate up to the root. We first retrieve the parent and go through if/else logic. If the parent was a left child and our node's right uncle was red, set the uncle and the parent to black, set the grandparent to red and go with the grandparent on the next iteration. If the parent was a left child and our node's uncle was black, If our node is a right child, call the rotate function to rotate left. Then always set the node to the parent to iterate on the parent next, rotate the grandparent to the right by calling the rotate function, and swap the parent and the grandparent colors. If the parent was a right child and our node's left uncle is red, set the uncle to black, set the parent to black, set the grandparent to red, and make the node the grandparent to iterate through him next. If the parent was a right child and our node's left uncle is black, then rotate the parent to the right. Then always rotate the grandparent to the left by calling the rotate function and swap the parent and the grandparent colors. Set the node to the parent to iterate through the parents next. After all the iterations are complete for any red nodes that we assign, the tree will be balanced. It will rotate and correct the children of the new node, and will balance out the colors so the RBT rules are followed.

Node* RBTree::RBTSearch(int key)

The search function will find and return a node that contains the key that was passed to the function. The function starts at the root of the tree and iterates through the nodes to find the key we're looking for. If the key is greater than the key of the node we're at, we will move right. If it's less than we will move left. If we find the key, we will return it to the calling funciton.

int RBTree::RBTDeleteAll(int key)
The delete all function deletes all instances of nodes with key that is passed in. In this method we keep a count and call the Delete function I will discus next. If a node was deleted, we will increment the counter and call delete again, until no more nodes exist to be deleted. It will return the counter.

Node* RBTree::RBTDelete(int key)
The delete function is the most complex. This function returns a pointer to the node that was deleted. It iterates through the tree to find the node, starting at the root and going down the tree, to the left if the key is less than the node, and right if the key is greater than. If it finds the node, it will fix the tree so it connects to the deleted nodes children correctly. If the deleted node has no children, the "grandchildren" (deleted node's parent's new children) will be null. If the deleted node has children, then we need to choose the successor. When the deleted node has a left child, assign the child node as the left child's most right grandchild, otherwise the right child's most left grandchild. Then go through and assign the parents to make sure the nodes are connected. All this is done in many if/else, so not all cases are hit at the same time. If there's no parents, make the grandchild the new root. Otherwise, assign the children for every flow going up to the root. If the final child is black and has grandchildren, then we must call the helper function to fix the colors after deletion. This function iterates from the bottom of the child and up to the root. If this current node is a left child with black color, and its sibling is red, then change the sibling to black and change the parent to red. Then rotate the parent to the left. If this current node is a left child with black color, and its sibling has two black children, make the sibling red and go up a node with to make a current node the parent. If this current node is a left child with a black color, and its sibling's right child is black, then make the sibling's left child black as well and the sibling itself red. Rotate the sibling to the right. Then assign the sibling color to the current parent's color and make that parent black, the siblings right child black, and rotate the parent to the left. If this node is a right child and the sibling is red, then make the sibling black and their parent red. Rotate the parent to the right If the sibling has two black children, make the sibling color red and change the current node to the parent. If the sibling doesn't have two black children, but the left child is black then make the right child black and the sibling red. Then rotate the sibling to the left make the sibling color match the parent color and set the parent to black. Set the sibling's left to black (if it wasn't above) and rotate the parent to the right.

void RBTree::RBTInOrderPrintTree()
The print function will call a helper function passing in the root of the tree that will call itself recursively until the entire tree is printed. It will start at the left most node and finish at the right-most node to print the nodes in order. It will specify the key and color of each node, will mark the root node with "(Root)" specifier, and will print the parent's and both children's key and color. If there are no children, it will print "NIL." Root doesn't have a parent, so it will print "None."

B-Tree is a self balancing tree data structure. It's a generalization of a binary tree because a node can have more than two children. It requires logarithmic time to search, insert, access, and delete nodes, so it's well suited for storage systems. The nodes can have a different number of child nodes, but the range is specified. The number of child nodes can change, but will be joined or split when the maximum limit hits. B-trees can often waste some space as they're not fully filled, but they do not need re-balancing as often as other self-balancing trees.

void BTree::BTInsert(int key)
The insert function in my code will insert a new node with the given key into the tree. If the tree is empty, then the new node will become the root. If the tree is not empty, then it will find a suitable place for the new node. If the existing root is already full, then we'll make the new node the new root and make the old root its child. We split the old root and move one key to the new root by calling the split helper method. If the new root is less than the key we want to insert, then it will get the new key.

int BTree::BTDeleteAll(int key)
The function to delete all will call the delete function in a loop and return the count of how many nodes were deleted.

Node* BTree::BTDelete(int key)
The delete function will call a helper function to delete a node with the given key from the tree, which will call itself recursively until it find the correct node to delete. This delete function will then correct the root when the number of keys in the root drops to zero. The helper function does the heavy duty. It will first find a subtree by calling itself recursively with the next node. If it finds the node, it will call another helper function, depending on whether the delete is from a leaf, or from a non leaf node. It will return the deleted node. Deleting from leaf is simplest as we only move all the other keys a place backward and decrement the number of keys that node has. A non leaf deletion is a little more complicated as the code has to go through the children to find the predecessors or successors.

void BTree::BTInOrderPrintTree(Node* currentNode)
The print function will print out all of the nodes of the tree in order. It prints each node's key, whether it's a leaf node or not, and the number of keys it has, as well as the keys themselves. The same node will be printed multiple times as we iterate through the keys because the node contains an array of keys.

Node* BTree::BTSearch(Node* searchNode, int key)
The search function finds the given key in the tree. It calls itself recursively after figuring out which node to go to next. If it doesn't find the key in the current node, it will increment go to the next child. If it reaches a leaf, then the tree doesn't contain the key, so it will return null.