```cpp
/*************************************************************/
/* Author:          Philipp Riedel                           */
/* Due Date:        April 30, 2023                           */
/* Course:          CSC237                                   */
/* Professor Name:  Dr. Spiegel                              */
/* Project:         #4                                       */
/* Filename:        TreeTest.cpp                             */
/* Purpose:         This program will store given numbers    */
/*                  in a binarySearchTree and the user can   */
/*                  remove nodes and print the tree.          */
/*************************************************************/

/**
 * @mainpage Documentation of Project 4 (Binary Search Tree)
 * @author Philipp Riedel
 * @brief This programm will store given numbers in a Binary Search Tree and the user can remove
nodes and print the tree.
 */

/**
 * @file TreeTest.cpp
 * @brief Driver for Binary Tree ADT
 */

#include <iostream>
#include <string>
#include "BinarySearchTree.h"

using namespace std;

typedef BinaryTree<int> IntTree;

/*!
 * \fn getChoice
 * @brief gets what the user inputs
 */
char getChoice(string ok);

/*!
 * \fn addToTree
 * @brief Insert Value to Search Tree
 */
void addToTree(IntTree &TheTree);

/*!
 * \fn removeFromTree
 * @brief Remove Value from Search Tree
 */
void removeFromTree(IntTree &TheTree);

/*!
 * \fn change
 * @brief change value from Search Tree
 */
void change(IntTree &TheTree);

/*!
```

```
 * \fn main
 * @brief main program
 */
int main()
{
  IntTree Tree;
  int entry, *result;
  char Choice;
  do
  {
    cout << "Select: A)dd    R)emove    C)hange     P)rint     T)ree Print     Q)uit\n";
    Choice = getChoice("ARCPTQ");
    switch (Choice)
    {
    case 'A':
      addToTree(Tree);
      break;
    case 'C':
      change(Tree);
      break;
    case 'P':
      cout << "The Tree:" << endl;
      Tree.inorder();
      break;
    case 'R':
      removeFromTree(Tree);
      break;
    case 'T':
      cout << "The tree, as it appears (sort of)..\n";
      Tree.treePrint();
    }
  } while (Choice != 'Q');
}

// get the choice of the user
char getChoice(string ok)
{
  char ch = ' ';
  do
    ch = toupper(cin.get());
  while (ok.find(ch) == string::npos);
  cin.get(); // eat CR
  return (toupper(ch));
}

// Insert Value to Search Tree
void addToTree(IntTree &TheTree)
{
  int entry;
  cout << " Enter an Integer >";
  cin >> entry;
  TheTree.insertToTree(entry);
}

// Remove Value from Search Tree
void removeFromTree(IntTree &TheTree)
{
```

```
  int entry, *result;
  cout << "Value to Delete? >";
  cin >> entry;
  result = TheTree.treeSearch(entry);
  if (!result)
    cout << entry << " Not Found\n";
  else
    TheTree.deleteFromTree(entry);
}

// Change value
void change(IntTree &TheTree)
{
  int entry, *result;
  cout << "Enter the number you wish to replace: ";
  cin >> entry;
  result = TheTree.treeSearch(entry);
  if (!result)
  {
    cout << entry << " Not Found\n";
    return;
  }
  else
    TheTree.deleteFromTreeChange(entry);

  int entry2;
  cout << "What number would you like to put in place of " << entry << ": ";
  cin >> entry2;
  TheTree.insertToTree(entry2);
}
```

```
// File: BinarySearchTree.h
// Binary Tree ADT defined using Linked Structures

/**
 * @file BinarySearchTree.h
 * @brief Binary Tree ADT defined using Linked Structures (header file)
 */
#ifndef TREE_H
#define TREE_H

template <typename treeEltType>
class BinaryTree;

/*!
 * \class TreeNode
 * @brief TreeNode class
 */
template <typename eltType>
class TreeNode
{
private:
    eltType info;
    int count;
    TreeNode<eltType> *left, *right;
```

```
    TreeNode(const eltType &data, const int &count2 = 0, TreeNode<eltType> *lChild = NULL,
TreeNode *rChild = NULL)
    {
        count = count2;
        info = data;
        left = lChild;
        right = rChild;
    }

    friend class BinaryTree<eltType>;
};

/*!
 * \class BinaryTree
 * @brief BinaryTree class
 */
template <typename treeEltType>
class BinaryTree
{

public:
    // Constructor
    BinaryTree();

    // Place Element into Tree
    // Returns 1 if inserted, 0 if Data already in tree
    int insertToTree(const treeEltType &data);

    // Search for Element in Tree
    // Assumes == is defined for treeEltType
    // Returns pointer to data, or NULL, according to success
    treeEltType *treeSearch(const treeEltType &data);

    // Retrieve Element from Tree (leaving Tree Intact)
    // Precondition: Item is in Tree
    treeEltType &retrieveFromTree(const treeEltType &data);

    // Remove an Element from the tree
    // Pre: Element is in the Tree
    void deleteFromTree(const treeEltType &data);

    // Display Tree using InOrder Traversal
    void inorder() const;

    // Display Tree using PreOrder Traversal
    void preorder() const;

    // Display Tree using PostOrder Traversal
    void postorder() const;

    // Breadth first print
    void treePrint() const;

    // Remove an Element from the tree
    // Pre: Element is in the Tree
    void deleteFromTreeChange(const treeEltType &data);
```

```
private:
    TreeNode<treeEltType> *root;

    // Display Tree using InOrder Traversal
    void printInorder(TreeNode<treeEltType> *) const;

    // Display Tree using PreOrder Traversal
    void printPreorder(TreeNode<treeEltType> *) const;

    // Display Tree using PostOrder Traversal
    void printPostorder(TreeNode<treeEltType> *) const;

    void treePrintHelper(TreeNode<treeEltType> *) const;
};

#endif
```

```
/**
 * @file BinarySearchTree.cpp
 * @brief Binary Tree ADT implemented with TreeNode linked structures
 */
#include <iostream>
#include <string>
#include <queue>
#include "BinarySearchTree.h"

// ToDo:
// - Remove ding nochmal überprüfen

using namespace std;

/*!
 * \fn BinaryTree
 * @brief Constructor
 */
template <typename treeEltType>
BinaryTree<treeEltType>::BinaryTree()
{
    root = NULL;
}

/*!
 * \fn insertToTree
 * @brief Place Element into Tree (Returns 1 if inserted, 0 if data already in tree)
 */
template <typename treeEltType>
int BinaryTree<treeEltType>::insertToTree(const treeEltType &data)
{
    if (root == NULL)
    { // Empty Tree
        root = new TreeNode<treeEltType>(data);
        root->count = 1; // update: create root with count of 1
        return (1);
    }
    // Search for spot to put data; We only stop when we get to the bottom (NULL)
    TreeNode<treeEltType> *t = root, *parent;
```

```
    while (t != NULL)
    {
        if (t->info == data) // data already in Tree
        {
            t->count = t->count + 1; // update: increment
            return (1);
        }
        parent = t; // Set the trail pointer to the ancestor of where we're going
        if (data < t->info)
            t = t->left;
        else
            t = t->right;
    }
    // Found the spot; insert node.
    if (data < parent->info)
    {
        parent->left = new TreeNode<treeEltType>(data);
        t = parent->left;
        t->count = 1; // update: set count to 1
    }
    else
    {
        parent->right = new TreeNode<treeEltType>(data);
        t = parent->right;
        t->count = 1; // update: set count to 1
    }
    return (1);
}

/*!
 * \fn treeSearch
 * @brief Search for Element in Tree, Assumes == is defined for treeEltType, Returns Ptr to Elt
if Found, NULL otherwise
 */
template <typename treeEltType>
treeEltType *BinaryTree<treeEltType>::treeSearch(const treeEltType &key)
{
    TreeNode<treeEltType> *t = root;
    while (t && t->info != key)
        if (key < t->info)
            t = t->left;
        else
            t = t->right;
    if (t)
        return (&t->info);
    return (NULL);
}

/*!
 * \fn retrieveFromTree
 * @brief Retrieve Element from Tree (leaving Tree Intact), Precondition: Item is in Tree
 */
template <typename treeEltType>
treeEltType &BinaryTree<treeEltType>::retrieveFromTree(const treeEltType &key)
{
    TreeNode<treeEltType> *t;
    for (t = root; t->info != key;)
```

```
        if (key < t->info)
            t = t->left;
        else
            t = t->right;
    return (t->info);
}

/*!
 * \fn deleteFromTree
 * @brief Remove an Element from the tree
 * (if there is more than one Element, ask if only one or all elements should be removed)
 */
template <typename treeEltType>
void BinaryTree<treeEltType>::deleteFromTree(const treeEltType &data)
{
    TreeNode<treeEltType> *nodeWithData, *nodeToDelete, *t = root, *trailT = NULL;
    // Find spot
    while (t->info != data)
    { // safe because of precondition
        trailT = t;
        if (data < t->info)
            t = t->left;
        else
            t = t->right;
    }
    nodeWithData = t;             // Hold onto the data Node for later deletion
    if (nodeWithData->count == 1) // update: check if count is one
    {
        // Case 1: Leaf?
        if (!(nodeWithData->left) && !(nodeWithData->right))
        {
            // No Subtrees, node is leaf...Wipe it
            // Is it the root?
            if (nodeWithData == root)
                root = NULL;
            else if (trailT->right == nodeWithData) // Parent's right child
                trailT->right = NULL;
            else
                trailT->left = NULL;
            nodeToDelete = nodeWithData; // free this at the end
        }
        else if (!(nodeWithData->left))
        {
            // If 1st condition false and this one's true, there's a right subtree
            if (!trailT)
            { // Node to delete is root and there is no left subtree
                nodeToDelete = root;
                root = root->right;
            }
            else
            { // Point parent's pointer to this node to this node's right child
                if (trailT->right == nodeWithData)
                    trailT->right = nodeWithData->right;
                else
                    trailT->left = nodeWithData->right;
                nodeToDelete = nodeWithData;
            }
```

```
        }
        else if (!(nodeWithData->right))
        {
            // If 1st 2 conditions false and this one's true, there's a left subtree
            if (!trailT)
            { // Node to delete is root and there is no left subtree
                nodeToDelete = root;
                root = root->left;
            }
            else
            { // Otherwise, move up the right subtree
                if (trailT->right == nodeWithData)
                    trailT->right = nodeWithData->left;
                else
                    trailT->left = nodeWithData->left;
                nodeToDelete = nodeWithData;
            }
        }
        else
        { // If you make it here, node has two children
            // Go to rightmost node in left subtree; we know there's a right child...
            for (trailT = nodeWithData, t = nodeWithData->left;
                 t->right != NULL; trailT = t, t = t->right)
                ;
            // Want to copy data from node with 0 or 1 child to node with data to delete
            // Place node data in NodeWithData
            nodeWithData->info = t->info;
            // Set the parent of source node to point at source node's left child
            //   (We know it hasn't a right child. Also ok if no left child.)
            if (trailT == nodeWithData)
                // Need to point parent correctly.
                //   See if after the we went left there was no right child
                //   If there was no right child, this is rightmost node in left subtree
                trailT->left = t->left;
            else // we did go right; after going left, there was a right child
                // rightmost node has no r. child, so point its parent at its l. child
                trailT->right = t->left;
            nodeToDelete = t;
        }
        delete nodeToDelete;
    }
    else
    {
        char selection;
        cout << "Do you want to remove all " << nodeWithData->count << " copies or only One? (A
or O)";
        cin >> selection;
        switch (selection)
        {
        case 'A':
        case 'a':
            // Case 1: Leaf?
            if (!(nodeWithData->left) && !(nodeWithData->right))
            {
                // No Subtrees, node is leaf...Wipe it
                // Is it the root?
                if (nodeWithData == root)
```

```
                root = NULL;
            else if (trailT->right == nodeWithData) // Parent's right child
                trailT->right = NULL;
            else
                trailT->left = NULL;
            nodeToDelete = nodeWithData; // free this at the end
        }
        else if (!(nodeWithData->left))
        {
            // If 1st condition false and this one's true, there's a right subtree
            if (!trailT)
            { // Node to delete is root and there is no left subtree
                nodeToDelete = root;
                root = root->right;
            }
            else
            { // Point parent's pointer to this node to this node's right child
                if (trailT->right == nodeWithData)
                    trailT->right = nodeWithData->right;
                else
                    trailT->left = nodeWithData->right;
                nodeToDelete = nodeWithData;
            }
        }
        else if (!(nodeWithData->right))
        {
            // If 1st 2 conditions false and this one's true, there's a left subtree
            if (!trailT)
            { // Node to delete is root and there is no left subtree
                nodeToDelete = root;
                root = root->left;
            }
            else
            { // Otherwise, move up the right subtree
                if (trailT->right == nodeWithData)
                    trailT->right = nodeWithData->left;
                else
                    trailT->left = nodeWithData->left;
                nodeToDelete = nodeWithData;
            }
        }
        else
        { // If you make it here, node has two children
            // Go to rightmost node in left subtree; we know there's a right child...
            for (trailT = nodeWithData, t = nodeWithData->left;
                 t->right != NULL; trailT = t, t = t->right)
                ;
            // Want to copy data from node with 0 or 1 child to node with data to delete
            // Place node data in NodeWithData
            nodeWithData->info = t->info;
            // Set the parent of source node to point at source node's left child
            //   (We know it hasn't a right child. Also ok if no left child.)
            if (trailT == nodeWithData)
                // Need to point parent correctly.
                //   See if after the we went left there was no right child
                //   If there was no right child, this is rightmost node in left subtree
                trailT->left = t->left;
```

```
                else // we did go right; after going left, there was a right child
                    // rightmost node has no r. child, so point its parent at its l. child
                    trailT->right = t->left;
                nodeToDelete = t;
            }
            delete nodeToDelete;
            break;
        case 'O':
        case 'o':
            nodeWithData->count = nodeWithData->count - 1;
            break;
        default:
            break;
        }
    }
}

/*!
 * \fn printInorder
 * @brief prints the Tree in order (is the helper function)
 */
template <typename treeEltType>
void BinaryTree<treeEltType>::printInorder(TreeNode<treeEltType> *t) const
// void printTheTree(TreeNode *T)
{
    if (t)
    {
        printInorder(t->left);
        cout << t->info;
        if (t->count > 1)
        {
            cout << "(" << t->count << ")" << endl;
        }
        else
        {
            cout << endl;
        }
        printInorder(t->right);
    }
}

/*!
 * \fn inorder
 * @brief Display Tree using InOrder Traversal
 */
template <typename treeEltType>
void BinaryTree<treeEltType>::inorder() const
{
    printInorder(root);
}

/*!
 * \fn printPreorder
 * @brief Need Helper to Recursively Print the Tree
 */
template <typename treeEltType>
void BinaryTree<treeEltType>::printPreorder(TreeNode<treeEltType> *t) const
```

```
// void printTheTree(TreeNode *t)
{
    if (t)
    {
        cout << t->info << endl;
        printPreorder(t->left);
        printPreorder(t->right);
    }
}

/*!
 * \fn preorder
 * @brief Display Tree using preorder Traversal
 */
template <typename treeEltType>
void BinaryTree<treeEltType>::preorder() const
{
    printInorder(root);
}

/*!
 * \fn printPostorder
 * @brief Need Helper to Recursively Print the Tree
 */
template <typename treeEltType>
void BinaryTree<treeEltType>::printPostorder(TreeNode<treeEltType> *t) const
// void printTheTree(TreeNode *t)
{
    if (t)
    {
        printPostorder(t->left);
        printPostorder(t->right);
        cout << t->info << endl;
    }
}

/*!
 * \fn postorder
 * @brief Display Tree using InOrder Traversal (calls helper function)
 */
template <typename treeEltType>
void BinaryTree<treeEltType>::postorder() const
{
    printInorder(root);
}

/*!
 * \fn treePrint
 * @brief calls the helper function to print the tree
 */
template <typename treeEltType>
void BinaryTree<treeEltType>::treePrint() const
{
    treePrintHelper(root);
}

/*!
```

```
 * \fn treePrintHelpers
 * @brief prints the tree
 */
template <typename treeEltType>
void BinaryTree<treeEltType>::
    treePrintHelper(TreeNode<treeEltType> *root) const
{
    queue<TreeNode<treeEltType> *> Q;
    // A dummy node to mark end of level
    TreeNode<treeEltType> *dummy = new TreeNode<treeEltType>(-1);
    if (root)
    {
        cout << root->info;
        if (root->count > 1)
        {
            cout << "(" << root->count << ")" << endl;
        }
        else
        {
            cout << endl;
        }
        Q.push(root->left);
        Q.push(root->right);
        Q.push(dummy);
    }
    TreeNode<treeEltType> *t = root;
    while (!Q.empty())
    {
        t = Q.front();
        Q.pop();
        if (t == dummy)
        {
            if (!Q.empty())
                Q.push(dummy);
            cout << endl;
        }
        else if (t)
        {
            cout << t->info;
            if (t->count > 1)
            {
                cout << "(" << t->count << ") ";
            }
            else
            {
                cout << " ";
            }
            Q.push(t->left);
            Q.push(t->right);
        }
    }
}

/*!
 * \fn deleteFromTreeChange
 * @brief Remove an Element from the tree (is the version for the Change option)
 */
```

```
template <typename treeEltType>
void BinaryTree<treeEltType>::deleteFromTreeChange(const treeEltType &data)
{
    TreeNode<treeEltType> *nodeWithData, *nodeToDelete, *t = root, *trailT = NULL;
    // Find spot
    while (t->info != data)
    { // safe because of precondition
        trailT = t;
        if (data < t->info)
            t = t->left;
        else
            t = t->right;
    }
    nodeWithData = t;             // Hold onto the data Node for later deletion
    if (nodeWithData->count == 1) // update: check if count is one
    {
        // Case 1: Leaf?
        if (!(nodeWithData->left) && !(nodeWithData->right))
        {
            // No Subtrees, node is leaf...Wipe it
            // Is it the root?
            if (nodeWithData == root)
                root = NULL;
            else if (trailT->right == nodeWithData) // Parent's right child
                trailT->right = NULL;
            else
                trailT->left = NULL;
            nodeToDelete = nodeWithData; // free this at the end
        }
        else if (!(nodeWithData->left))
        {
            // If 1st condition false and this one's true, there's a right subtree
            if (!trailT)
            { // Node to delete is root and there is no left subtree
                nodeToDelete = root;
                root = root->right;
            }
            else
            { // Point parent's pointer to this node to this node's right child
                if (trailT->right == nodeWithData)
                    trailT->right = nodeWithData->right;
                else
                    trailT->left = nodeWithData->right;
                nodeToDelete = nodeWithData;
            }
        }
        else if (!(nodeWithData->right))
        {
            // If 1st 2 conditions false and this one's true, there's a left subtree
            if (!trailT)
            { // Node to delete is root and there is no left subtree
                nodeToDelete = root;
                root = root->left;
            }
            else
            { // Otherwise, move up the right subtree
                if (trailT->right == nodeWithData)
```

```
                trailT->right = nodeWithData->left;
            else
                trailT->left = nodeWithData->left;
            nodeToDelete = nodeWithData;
        }
    }
    else
    { // If you make it here, node has two children
        // Go to rightmost node in left subtree; we know there's a right child...
        for (trailT = nodeWithData, t = nodeWithData->left;
             t->right != NULL; trailT = t, t = t->right)
            ;
        // Want to copy data from node with 0 or 1 child to node with data to delete
        // Place node data in NodeWithData
        nodeWithData->info = t->info;
        // Set the parent of source node to point at source node's left child
        //   (We know it hasn't a right child. Also ok if no left child.)
        if (trailT == nodeWithData)
            // Need to point parent correctly.
            //   See if after the we went left there was no right child
            //   If there was no right child, this is rightmost node in left subtree
            trailT->left = t->left;
        else // we did go right; after going left, there was a right child
            // rightmost node has no r. child, so point its parent at its l. child
            trailT->right = t->left;
        nodeToDelete = t;
    }
    delete nodeToDelete;
}
else
{

    // Case 1: Leaf?
    if (!(nodeWithData->left) && !(nodeWithData->right))
    {
        // No Subtrees, node is leaf...Wipe it
        // Is it the root?
        if (nodeWithData == root)
            root = NULL;
        else if (trailT->right == nodeWithData) // Parent's right child
            trailT->right = NULL;
        else
            trailT->left = NULL;
        nodeToDelete = nodeWithData; // free this at the end
    }
    else if (!(nodeWithData->left))
    {
        // If 1st condition false and this one's true, there's a right subtree
        if (!trailT)
        { // Node to delete is root and there is no left subtree
            nodeToDelete = root;
            root = root->right;
        }
        else
        { // Point parent's pointer to this node to this node's right child
            if (trailT->right == nodeWithData)
                trailT->right = nodeWithData->right;
```

```
                else
                    trailT->left = nodeWithData->right;
                nodeToDelete = nodeWithData;
            }
        }
        else if (!(nodeWithData->right))
        {
            // If 1st 2 conditions false and this one's true, there's a left subtree
            if (!trailT)
            { // Node to delete is root and there is no left subtree
                nodeToDelete = root;
                root = root->left;
            }
            else
            { // Otherwise, move up the right subtree
                if (trailT->right == nodeWithData)
                    trailT->right = nodeWithData->left;
                else
                    trailT->left = nodeWithData->left;
                nodeToDelete = nodeWithData;
            }
        }
        else
        { // If you make it here, node has two children
            // Go to rightmost node in left subtree; we know there's a right child...
            for (trailT = nodeWithData, t = nodeWithData->left;
                 t->right != NULL; trailT = t, t = t->right)
                ;
            // Want to copy data from node with 0 or 1 child to node with data to delete
            // Place node data in NodeWithData
            nodeWithData->info = t->info;
            // Set the parent of source node to point at source node's left child
            //   (We know it hasn't a right child. Also ok if no left child.)
            if (trailT == nodeWithData)
                // Need to point parent correctly.
                //   See if after the we went left there was no right child
                //   If there was no right child, this is rightmost node in left subtree
                trailT->left = t->left;
            else // we did go right; after going left, there was a right child
                // rightmost node has no r. child, so point its parent at its l. child
                trailT->right = t->left;
            nodeToDelete = t;
        }
        delete nodeToDelete;
    }
}

template class BinaryTree<int>;
```