МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

ІНСТИТУТ ПРИКЛАДНОГО СИСТЕМНОГО АНАЛІЗУ

Кафедра системного проектування

**ЗВІТ**про виконання лабораторної роботи №4

«Дослідження структури даних бінарне дерево пошуку«

Виконав:

cтудент І курсу

групи ДА-02

Гринчишин Д.Б.

Перевірили:

..

Київ – 2020

**Мета**. Ознайомитись і дослідити структури даних бінарне дерево пошуку та префіксне дерево, розглянути механізми балансування дерева. Набути навичок реалізації бінарного дерева пошуку мовою програмування С++, порівняти власну реалізацію з готовим бібліотечним рішенням STL.

**Варіант завдань – 2**

**Код:**

**autocompleter.h**

#include <iostream>

#include <ctime>

#include <cmath>

#include <cstdlib>

#include <cstring>

#include <algorithm>

#include <fstream>

#include <vector>

#include <unordered\_map>

using namespace std;

struct Node\_char{

    bool isWord = false;

    bool isLeaf = false;

    unordered\_map<char, Node\_char\*> children;

};

class Trie {

    public :

        ~Trie(){destructor(root);cout<<"autocompleter destructed" << endl;}

        Node\_char\* root = new Node\_char;

        void insert(char\* *word*) {

            Node\_char\* node = root;

            int l = strlen(*word*);

            for (int i = 0; i < l; i++) {

                node->isLeaf = false;

                if (node->children.count(*word*[i]) == 0) {

                    Node\_char\* next = new Node\_char;

                    node->children.insert({*word*[i], next });

                    node = next;

                    if (i == l - 1) node->isLeaf = true;

                }

                else node = node->children.at(*word*[i]);

            }

            node->isWord = true;

        }

        void insert\_words\_from\_file(const char\* *filepath*){

            fstream f;

            f.open(*filepath*, ios::in | ios::out);

            char temp[100];

            while(!f.eof()){

                f >> temp;

                insert(temp);

            }

            f.close();

        }

        vector<string\*>\* indByPrefix(string *word*) {

            cout << "search for words starting with " << '"' << *word* << '"' << endl;

            vector<string\*>\* possible\_words = new vector<string\*>;

            Node\_char\* node = root;

            for (int i = 0; i < *word*.length(); i++) {

                if (node->children.count(*word*[i]) == 0) return possible\_words;

                node = node->children.at(*word*[i]);

            }

            indByPrefix(new string(*word*), node, possible\_words);

            return possible\_words;

        }

    private :

        void indByPrefix(string\* *word*, Node\_char\* *local\_root*, vector<string\*>\* *possible\_words*) {

            if(*local\_root*->isWord) {

*possible\_words*->push\_back(*word*);

*word* = new string(\**word*);

            }

            if(!*local\_root*->isLeaf){

                string parentword(\**word*);

                for(auto it = *local\_root*->children.cbegin(); it != *local\_root*->children.cend(); ++it){

                    string\* new\_word = it == *local\_root*->children.cbegin() ? *word* : new string(parentword);

                    new\_word->push\_back(it->first);

                    indByPrefix(new\_word, it->second, *possible\_words*);

                }

            }

        }

        void destructor(Node\_char\* *local\_root*){

            for(auto it = *local\_root*->children.cbegin(); it != *local\_root*->children.cend(); ++it){

                destructor(it->second);

                delete it->second;

            }

        }

};

**Common.h**

#include <iostream>

#include <ctime>

#include <cmath>

#include <string>

#include <cstdlib>

using namespace std;

long long generateRandLong(int *len* = 12) {

    long long num = (rand() % 9 + 1) \* pow(10, *len* - 1);

    for (long long rank = pow(10, *len* - 2); rank != 0; rank /= 10) num += rand() % 10 \* rank;

    return num;

}

struct Data{

    char \* name;

    int mean\_points;

    int will;

    int cheated;

    Data(){

        int namelen = rand() % 5 + 5;

        name = new char[namelen + 1];

        for (size\_t i = 0; i < namelen; i++) name[i] = rand() % 26 + 'a';

        name[namelen] = '\0';

        mean\_points = rand() % 1000000;//(float)(rand() % 100) / (rand() % 100);

        will = rand() % 10;

        cheated = rand() % 10000;

    }

};

bool operator<(const Data & *e1*, const Data & *e2*){ return *e1*.mean\_points + *e1*.cheated < *e2*.mean\_points + *e2*.cheated; }

bool operator>(const Data & *e1*, const Data & *e2*){ return *e1*.mean\_points + *e1*.cheated > *e2*.mean\_points + *e2*.cheated; }

bool operator>=(const Data & *e1*, const Data & *e2*){ return *e1*.mean\_points + *e1*.cheated >= *e2*.mean\_points + *e2*.cheated; }

bool operator<=(const Data & *e1*, const Data & *e2*){ return *e1*.mean\_points + *e1*.cheated <= *e2*.mean\_points + *e2*.cheated; }

bool operator==(const Data & *e1*, const Data & *e2*){ return *e1*.mean\_points + *e1*.cheated == *e2*.mean\_points + *e2*.cheated; }

bool operator!=(const Data & *e1*, const Data & *e2*){ return *e1*.mean\_points + *e1*.cheated != *e2*.mean\_points + *e2*.cheated; }

ostream& operator<<(ostream & *os*, Data & *e*){

*os* << *e*.name << ' ' << *e*.mean\_points << ' ' << *e*.will << ' ' << *e*.cheated;

    return *os*;

}

template<typename T> struct BinaryNode{

    T data;

    BinaryNode<T>\* left = NULL;

    BinaryNode<T>\* right = NULL;

    BinaryNode<T>\* parent = NULL;

    BinaryNode(T *val*, BinaryNode<T>\* *ptr*) {

        data = *val*;

        parent = *ptr*;

    }

    BinaryNode(BinaryNode<T>\* *ptr*) {

        data = *ptr*->data;

        left = *ptr*->left;

        right = *ptr*->right;

        parent = *ptr*->parent;

    }

};

template<typename T> struct AVLNode{

    T data;

    int height = 1;

    AVLNode<T>\* left = NULL;

    AVLNode<T>\* right = NULL;

    AVLNode<T>\* parent = NULL;

    AVLNode(T *val*, AVLNode<T>\* *ptr*) {

        data = *val*;

        parent = *ptr*;

    }

    AVLNode(AVLNode<T>\* *ptr*) {

        data = *ptr*->data;

        height = *ptr*->height;

        left = *ptr*->left;

        right = *ptr*->right;

        parent = *ptr*->parent;

    }

};

**avl\_tree.h**

#include <iostream>

#include <ctime>

#include <cmath>

#include <cstdlib>

#include <cstring>

#include "test\_binary\_search\_tree.h"

using namespace std;

template<typename T> class AVLTree{

    public :

        typedef AVLNode<T> Node;

        Node\* root = NULL;

        int Size = 0;

        AVLTree(){

        }

        AVLTree(Node\* *node*){

            root = *node*;

            Size = size(root);

            updateHeightGlobalRecursive(root);

        }

        ~AVLTree(){

            cout << "destructed" << endl;

            auto f = [=](Node\* *node*){ delete *node*; };

            postorderTraversal(root, f);

        }

        void insert(T *val*){

            Size++;

            if(root == NULL) root = new Node{*val*, NULL};

            else insert(root, *val*);

        }

        void erase(T *val*){

            if(root){

                Size--;

                root = erase(root, *val*);

            }

        }

        T eraseMin(Node\* *local\_root*){

            if (*local\_root*->left == NULL){

*local\_root*->parent->left = *local\_root*->right;

                if(*local\_root*->right) *local\_root*->right->parent = *local\_root*->parent;

                T val = *local\_root*->data;

                delete *local\_root*;

                return val;

            }

            T val = eraseMin(*local\_root*->left);

            balance(*local\_root*);

            return val;

        }

        T eraseMax(Node\* *local\_root*){

            if (*local\_root*->right == NULL){

*local\_root*->parent->right = *local\_root*->left;

                if(*local\_root*->left) *local\_root*->left->parent = *local\_root*->parent;

                T val = *local\_root*->data;

                delete *local\_root*;

                return val;

            }

            T val = eraseMax(*local\_root*->right);

            balance(*local\_root*);

            return val;

        }

        bool find(T *val*){ return root == NULL ? false : find\_ptr(*val*)->data == *val*; }

        vector<T> findInRange\_vector(T *minObject*, T *maxObject*){

            vector<T> elements;

            if(*minObject* < *maxObject*)findInRange\_vector(*minObject*, *maxObject*, &elements, root);

            return elements;

        }

        int findInRange(T *minObject*, T *maxObject*){

            int num = 0;

            if(*minObject* < *maxObject*) num = findInRange(*minObject*, *maxObject*, root);

            return num;

        }

        void eraseRange(T *minObject*, T *maxObject*){

            vector<T> elems\_to\_erase;

            findInRange\_vector(*minObject*, *maxObject*, &elems\_to\_erase, root);

            for(auto it = elems\_to\_erase.begin(); it != elems\_to\_erase.end(); ++it) erase(\*it);

        }

        int height(){ return root->height; }

        bool empty(){ return root == NULL; }

        int size(){ return this->Size; }

        AVLTree<T>\* split(T *key*){

            pair<Node\*, Node\*> p = split(root, *key*);

            Node\* new\_root = p.second;

            root = p.first;

            AVLTree<T>\* new\_tree = new AVLTree<T>(new\_root);

            updateHeightGlobalRecursive(root);

            return new\_tree;

        }

        void print(){

            auto f = [this](Node\* *node*){ cout << *node*->data << " | "; };

            cout << "inorder traversal :" << endl;

            if(root == NULL) cout << "No elements" << endl;

            else inorderTraversal(root, f);

            cout << endl;

        }

        void preorderTraversal(){

            cout << "preorder traversal :" << endl;

            //auto f = [this](Node\* node){ cout << node->data << ' ' << bfactor(node) << endl <<  " | "; };

            auto f = [this](Node\* *node*){ cout << *node*->data <<  " | "; };

            if(root != NULL) preorderTraversal(root, f);

            else cout << "No elements to traverse" << endl;

            cout << endl;

        }

        void merge(AVLTree<T> &*tree*){

            Node\* new\_root = *tree*.root;

            root = merge(root, new\_root);

*tree*.root = NULL;

            updateHeightGlobalRecursive(root);

        }

        int bfactor(Node\* *local\_root*){

            return *local\_root* == NULL ? 0 : (*local\_root*->left != NULL ? *local\_root*->left->height : 0) - (*local\_root*->right != NULL ? *local\_root*->right->height : 0);

        }

        void balance(Node\* *local\_root*){

            updateHeightLocal(*local\_root*);

            int factor = bfactor(*local\_root*);

            if (factor == 2) {

                if (bfactor(*local\_root*->left) < 0){

                    rotateLeft(*local\_root*->left);

                }

                rotateRight(*local\_root*);

            }

            else if(factor == -2){

                if (bfactor(*local\_root*->right) > 0){

                    rotateRight(*local\_root*->right);

                }

                rotateLeft(*local\_root*);

            }

        }

        void updateHeightLocal(Node\* *r*){

            if(*r* != NULL)

*r*->height = 1 + max(*r*->right ? *r*->right->height : 0, *r*->left ? *r*->left->height : 0);

        }

        void updateHeightGlobalRecursive(Node\* *node*){

            if(*node* != NULL){

                updateHeightGlobalRecursive(*node*->left);

                updateHeightGlobalRecursive(*node*->right);

*node*->height = 1 + max(*node*->left ? *node*->left->height : 0, *node*->right ? *node*->right->height : 0);

            }

        }

        void rotateRight(Node\* *r*){

            Node\* temp\_node = *r*->right;

            T temp\_val = *r*->data;

*r*->right = *r*->left;

*r*->left = *r*->right->left;

            if(*r*->left) *r*->left->parent = *r*;

*r*->right->left = *r*->right->right;

*r*->right->right = temp\_node;

            if (temp\_node) temp\_node->parent = *r*->right;

*r*->data = *r*->right->data;

*r*->right->data = temp\_val;

            updateHeightLocal(*r*->right);

            updateHeightLocal(*r*);

        }

        void rotateLeft(Node\* *r*){

            Node\* temp\_node = *r*->left;

            T temp\_val = *r*->data;

*r*->left = *r*->right;

*r*->right = *r*->left->right;

            if(*r*->right) *r*->right->parent = *r*;

*r*->left->right = *r*->left->left;

*r*->left->left = temp\_node;

            if (temp\_node) temp\_node->parent = *r*->left;

*r*->data = *r*->left->data;

*r*->left->data = temp\_val;

            updateHeightLocal(*r*->left);

            updateHeightLocal(*r*);

        }

    private :

        void insert(Node\* *r*, T *val*){

            if (*r*->data != *val*){

                if(*val* > *r*->data){

                    if(*r*->right == NULL) *r*->right = new Node{*val*, *r*};

                    else insert(*r*->right, *val*);

                }

                else if(*val* < *r*->data){

                    if(*r*->left == NULL) *r*->left = new Node{*val*, *r*};

                    else insert(*r*->left, *val*);

                }

                balance(*r*);

            }

            else Size--;

        }

        void preorderTraversal(Node\* *node*, auto& *action*){

            if(*node* != NULL){

*action*(*node*);

                preorderTraversal(*node*->left, *action*);

                preorderTraversal(*node*->right, *action*);

            }

        }

        void inorderTraversal(Node\* *node*, auto& *action*){

            if(*node* != NULL){

                inorderTraversal(*node*->left, *action*);

*action*(*node*);

                inorderTraversal(*node*->right, *action*);

            }

        }

        void postorderTraversal(Node\* *node*, auto& *action*){

            if(*node* != NULL){

                postorderTraversal(*node*->left, *action*);

                postorderTraversal(*node*->right, *action*);

*action*(*node*);

            }

        }

        Node\* merge(Node\* *t1*, Node\* *t2*){

            if (*t2* == NULL) return *t1*;

            if (*t1* == NULL) return *t2*;

            else if (*t1*->data > *t2*->data){

*t1*->right = merge(*t1*->right, *t2*);

                return *t1*;

            }

            else{

*t2*->left = merge(*t1*, *t2*->left);

                return *t2*;

            }

        }

        pair<Node\*, Node\*> split(Node\* *local\_root*, T *key*){

            if(*local\_root* == NULL) return pair<Node\*, Node\*>{NULL, NULL};

            else if(*key* > *local\_root*->data){

                pair<Node\*, Node\*> p = split(*local\_root*->right, *key*);

*local\_root*->right = p.first;

                if(p.first) p.first->parent = *local\_root*;

                return pair<Node\*, Node\*>{*local\_root*, p.second};

            }

            else{

                pair<Node\*, Node\*> p = split(*local\_root*->left, *key*);

*local\_root*->left = p.second;

                if(p.second) p.second->parent = *local\_root*;

                return pair<Node\*, Node\*>{p.first, *local\_root*};

            }

        }

        int size(Node\* *node*){ return *node* == NULL ? 0 : 1 + size(*node*->left) + size(*node*->right); }

        void findInRange\_vector(T *minObject*, T *maxObject*, vector<T>\* *elements*, Node\* *local\_root*){

            if(*local\_root* != NULL){

                T data = *local\_root*->data;

                if(data <= *maxObject* && data >= *minObject*) *elements*->push\_back(*local\_root*->data);

                if(data >= *minObject*) findInRange\_vector(*minObject*, *maxObject*, *elements*, *local\_root*->left);

                if(data <= *maxObject*) findInRange\_vector(*minObject*, *maxObject*, *elements*, *local\_root*->right);

            }

        }

        int findInRange(T *minObject*, T *maxObject*, Node\* *local\_root*){

            int num = 0;

            if(*local\_root* != NULL){

                T data = *local\_root*->data;

                if(data <= *maxObject* && data >= *minObject*) num++;

                if(data > *minObject*) num += findInRange(*minObject*, *maxObject*, *local\_root*->left);

                if(data < *maxObject*) num += findInRange(*minObject*, *maxObject*, *local\_root*->right);

            }

            return num;

        }

        Node\* erase(Node\* *local\_root*, T *val*){

            if (*local\_root* == NULL){

                Size++;

                return NULL;

            }

            Node\* reserve = *local\_root*;

            if (*val* < *local\_root*->data){

*local\_root*->left = erase(*local\_root*->left, *val*);

                if(*local\_root*->left) *local\_root*->left->parent = *local\_root*;

            }

            else if (*val* > *local\_root*->data) {

*local\_root*->right = erase(*local\_root*->right, *val*);

                if(*local\_root*->right) *local\_root*->right->parent = *local\_root*;

            }

            else if (*local\_root*->left != NULL and local\_root->right != NULL) {

                if(*local\_root*->right->left == NULL){

*local\_root*->data = *local\_root*->right->data;

                    reserve = *local\_root*->right;

*local\_root*->right = *local\_root*->right->right;

                    delete reserve;

                    if(*local\_root*->right) *local\_root*->right->parent = *local\_root*;

                }

                else

*local\_root*->data = eraseMin(*local\_root*->right);

            }

            else {

                reserve = *local\_root*;

                if (*local\_root*->left != NULL)

*local\_root* = *local\_root*->left;

                else if (*local\_root*->right != NULL)

*local\_root* = *local\_root*->right;

                else

*local\_root* = NULL;

                delete reserve;

            }

            balance(*local\_root*);

            return *local\_root*;

        }

        Node\* find\_min(Node\* *local\_root*){

            Node\* node = *local\_root*;

            for(; node->left != NULL; node = node->left){}

            return node;

        }

        Node\* find\_max(Node\* *local\_root*){

            Node\* node = *local\_root*;

            for(; node->right != NULL; node = node->right){}

            return node;

        }

        Node\* find\_ptr(T *val*){

            if (root != NULL){

                Node\* node = root;

                while(*val* > node->data ? node->right != NULL : node->left != NULL){

                    if(node->data == *val*) return node;

                    else node = *val* > node->data ? node->right : node->left;

                }

                return node;

            }

            return NULL;

        }

};

**binary\_search\_tree.h**

#include <iostream>

#include <ctime>

#include <cmath>

#include <cstdlib>

#include <cstring>

#include <set>

#include <algorithm>

#include "common.h"

using namespace std;

template<typename T> class BinarySearchTree{

    public :

        typedef BinaryNode<T> Node;

        Node\* root = NULL;

        int Size = 0;

        BinarySearchTree(){}

        BinarySearchTree(Node\* *node*){

            root = *node*;

            Size = size(root);

        }

        ~BinarySearchTree(){

            cout << "destructed" << endl;

            auto f = [=](Node\* *node*){ delete *node*; };

            postorderTraversal(root, f);

        }

        void insert(T *new\_val*){

            Size ++;

            if(root == NULL) root = new Node{*new\_val*, NULL};

            else{

                Node\* node = find\_ptr(*new\_val*);

                if (*new\_val* > node->data) node->right = new Node{*new\_val*, node};

                else if(*new\_val* < node->data) node->left = new Node{*new\_val*, node};

                else Size--;

            }

        }

        void erase(T *val*){

            if(root != NULL){

                Node\* elem\_to\_delete = find\_ptr(*val*);

                if(elem\_to\_delete->data == *val*){

                    if(elem\_to\_delete->right != NULL){

                        Node\* node = find\_min(elem\_to\_delete->right);

                        elem\_to\_delete->data = node->data;

                        if(node->parent != elem\_to\_delete) node->parent->left = node->right;

                        else node->parent->right = node->right;

                        if(node->right != NULL) node->right->parent = node->parent;

                        delete node;

                    }

                    else if(elem\_to\_delete->left != NULL){

                        Node\* node = find\_max(elem\_to\_delete->left);

                        elem\_to\_delete->data = node->data;

                        if(node->parent != elem\_to\_delete) node->parent->right = node->left;

                        else node->parent->left = node->left;

                        if(node->left != NULL) node->left->parent = node->parent;

                        delete node;

                    }

                    else{

                        if(elem\_to\_delete->parent != NULL){

                            if(elem\_to\_delete->parent->left == elem\_to\_delete) elem\_to\_delete->parent->left = NULL;

                            else elem\_to\_delete->parent->right = NULL;

                        }

                        else root = NULL;

                        delete elem\_to\_delete;

                    }

                    Size--;

                }

            }

        }

        bool find(T *val*){ return root == NULL ? false : find\_ptr(*val*)->data == *val*; }

        vector<T> findInRange\_vector(T *minObject*, T *maxObject*){

            vector<T> elements;

            if(*minObject* < *maxObject*)findInRange\_vector(*minObject*, *maxObject*, &elements, root);

            return elements;

        }

        int findInRange(T *minObject*, T *maxObject*){

            int num = 0;

            if(*minObject* < *maxObject*) num = findInRange(*minObject*, *maxObject*, root);

            return num;

        }

        void eraseRange(T *minObject*, T *maxObject*){

            vector<T> elems\_to\_erase;

            if(*minObject* < *maxObject*){

                findInRange\_vector(*minObject*, *maxObject*, &elems\_to\_erase, root);

                for(auto it = elems\_to\_erase.begin(); it != elems\_to\_erase.end(); ++it) erase(\*it);

            }

        }

        int height(){ return height(root); }

        bool empty(){ return Size <= 0; }

        int size(){ return this->Size; }

        void print(){

            auto f = [this](Node\* *node*){ cout << *node*->data << " | "; };

            cout << "inorder traversal :" << endl;

            if(root == NULL) cout << "No elements" << endl;

            else inorderTraversal(root, f);

            cout << endl;

        }

        void preorderTraversal(){

            cout << "preorder traversal :" << endl;

            auto f = [this](Node\* *node*){ cout << *node*->data << " | "; };

            if(root != NULL) preorderTraversal(root, f);

            else cout << "No elements to traverse" << endl;

            cout << endl;

        }

        void merge(BinarySearchTree<T>& *tree*){

            Node\* new\_root = *tree*.root;

            root = merge(root, new\_root);

*tree*.root = NULL;

        }

        BinarySearchTree<T>\* split(T *key*){

            pair<Node\*, Node\*> p = split(root, *key*);

            Node\* new\_root = p.second;

            root = p.first;

            BinarySearchTree<T>\* new\_tree = new BinarySearchTree<T>(new\_root);

            return new\_tree;

        }

    private :

        Node\* merge(Node\* *t1*, Node\* *t2*){

            if (*t2* == NULL) return *t1*;

            if (*t1* == NULL) return *t2*;

            else if (*t1*->data > *t2*->data){

*t1*->right = merge(*t1*->right, *t2*);

                if(*t1*->right) *t1*->right->parent = *t1*;

                return *t1*;

            }

            else{

*t2*->left = merge(*t1*, *t2*->left);

                if(*t2*->left) *t2*->left->parent = *t2*;

                return *t2*;

            }

        }

        void preorderTraversal(Node\* *node*, auto& *action*){

            if(*node* != NULL){

*action*(*node*);

                preorderTraversal(*node*->left, *action*);

                preorderTraversal(*node*->right, *action*);

            }

        }

        void inorderTraversal(Node\* *node*, auto& *action*){

            if(*node* != NULL){

                inorderTraversal(*node*->left, *action*);

*action*(*node*);

                inorderTraversal(*node*->right, *action*);

            }

        }

        void postorderTraversal(Node\* *node*, auto& *action*){

            if(*node* != NULL){

                postorderTraversal(*node*->left, *action*);

                postorderTraversal(*node*->right, *action*);

*action*(*node*);

            }

        }

        pair<Node\*, Node\*> split(Node\* *local\_root*, T *key*){

            if(*local\_root* == NULL) return pair<Node\*, Node\*>{NULL, NULL};

            else if(*key* > *local\_root*->data){

                pair<Node\*, Node\*> p = split(*local\_root*->right, *key*);

*local\_root*->right = p.first;

                if(p.first) p.first->parent = *local\_root*;

                return pair<Node\*, Node\*>{*local\_root*, p.second};

            }

            else{

                pair<Node\*, Node\*> p = split(*local\_root*->left, *key*);

*local\_root*->left = p.second;

                if(p.second) p.second->parent = *local\_root*;

                return pair<Node\*, Node\*>{p.first, *local\_root*};

            }

        }

        int size(Node\* *node*){ return *node* == NULL ? 0 : 1 + size(*node*->left) + size(*node*->right); }

        Node\* find\_ptr(T *val*){

            if (root != NULL){

                Node\* node = root;

                while(*val* > node->data ? node->right != NULL : node->left != NULL){

                    if(node->data == *val*) return node;

                    else node = *val* > node->data ? node->right : node->left;

                }

                return node;

            }

            return NULL;

        }

        void findInRange\_vector(T *minObject*, T *maxObject*, vector<T>\* *elements*, Node\* *local\_root*){

            if(*local\_root* != NULL){

                T data = *local\_root*->data;

                if(data <= *maxObject* && data >= *minObject*) *elements*->push\_back(*local\_root*->data);

                if(data >= *minObject*) findInRange\_vector(*minObject*, *maxObject*, *elements*, *local\_root*->left);

                if(data <= *maxObject*) findInRange\_vector(*minObject*, *maxObject*, *elements*, *local\_root*->right);

            }

        }

        int findInRange(T *minObject*, T *maxObject*, Node\* *local\_root*){

            int num = 0;

            if(*local\_root* != NULL){

                T data = *local\_root*->data;

                if(data <= *maxObject* && data >= *minObject*) num++;

                if(data >= *minObject*) num += findInRange(*minObject*, *maxObject*, *local\_root*->left);

                if(data <= *maxObject*) num += findInRange(*minObject*, *maxObject*, *local\_root*->right);

            }

            return num;

        }

        int height(Node\* *local\_root*) { return *local\_root* == NULL ? 0 : 1 + max( height( *local\_root*->left ), height( *local\_root*->right ) ); }

        Node\* find\_min(Node\* *local\_root*){

            Node\* node = *local\_root*;

            for(; node->left != NULL; node = node->left){}

            return node;

        }

        Node\* find\_max(Node\* *local\_root*){

            Node\* node = *local\_root*;

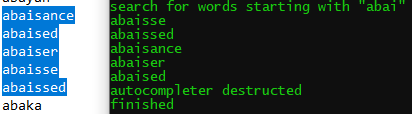
            for(; node->right != NULL; node = node->right){}

            return node;

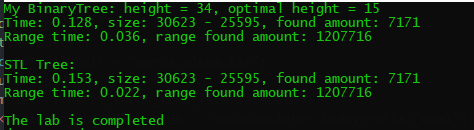
        }

};

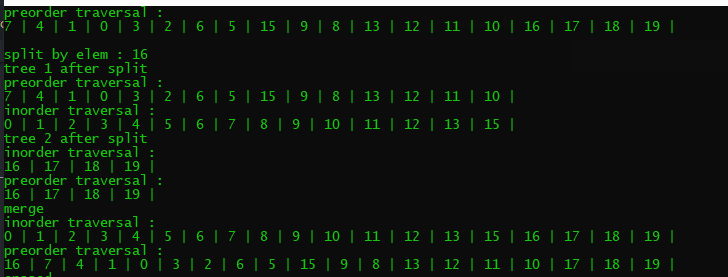
**Результат виконання програми:**

**Trie  
**

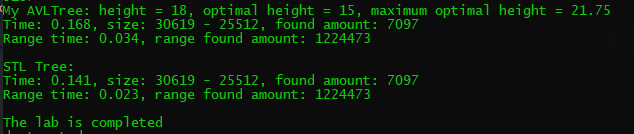
**BST default test**

****

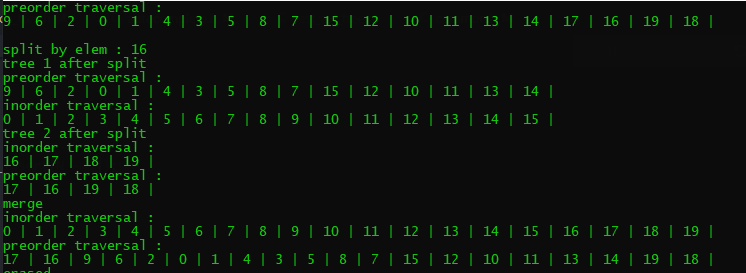
**BST merge/split test**

****

**AVL defaut test**

****

**AVL merge/split test**

****

**Висновок:**

В ході виконання лабораторної роботи було реалізовано такі структури даних як : бінарне дерево пошуку, збалансоване бінарне дерево пошуку АВЛ та всі відповідні їхні методи. Було реалізовано префіксне дерево та методи для авто доповнення для слів, що є в базі даних.

Звичайне бінарне дерево пошуку в середньому працює швидше, ніж set при будь-якій кількості ітерацій та елементів. Виключенням є пошук по діапазону значень.