

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра САПР

ОТЧЕТ
по лабораторной работе №1
по дисциплине «Алгоритмы и структуры данных»
Тема: «Ассоциативный массив»

Студент гр. 9302

Точилин А.Е.

Преподаватель

Тутуева А.В.

Санкт-Петербург

2021

Цель работы

Реализовать шаблонный ассоциативный массив (map) на основе красно-черного дерева.

Постановка задачи. Описание реализуемого класса и методов

Нужно написать шаблонный класс ассоциативного массива на основе красно-черного дерева поиска и реализовать следующие методы:

insert(ключ, значение) // добавление элемента с ключом и значением

remove(ключ) // удаление элемента дерева по ключу

find(ключ) // поиск элемента по ключу

clear // очищение ассоциативного массива

get_keys // возвращает список ключей

get_values // возвращает список значений

print // вывод в консоль

Каждая функция должна быть протестирована с помощью CppUnitTestFixture.

В результате выполнения данной лабораторной работы был создан список `List<Node<int, int>*>*` В для записи в него списка значений ассоциативного массива.

Оценка временной сложности каждого метода

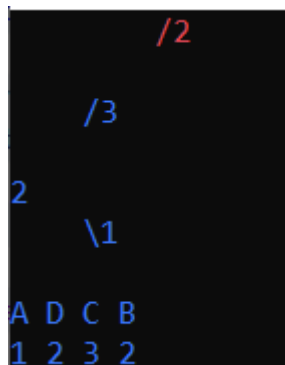
Функция	Сложность
insert(ключ, значение)	$O(\log(n))$
remove(ключ)	$O(\log(n))$
find(ключ)	$O(\log(n))$
clear	$O(n)$
get_keys	$O(n)$
get_values	$O(n)$
print	$O(n)$

Описание реализованных unit-тестов

Перед запуском каждого теста инициализируется ассоциативный массив значений типа `string` и ключей типа `int` `{(30, a); (40, a); (15, daada); (10, daada); (20, daada); (5, daada); (3, daada);}` в методе `TEST_METHOD_INITIALIZE`.

- 1) `test_insert` – Тестирует метод `void insert(T1 key, T2 data)`. В массив добавляются элементы `(2, oleg)` и `(21, ne sdast labu)` после чего с помощью обхода по дереву проверяем правильность расположений элементов в дереве и корректность вставленных значений.
- 2) `test_remove1` – Тестирует метод `void remove(T1 key)`. Удаляем элемент с ключом 15 после чего проверяем правильность дерева с помощью обхода и вывода в консоль.
- 3) `test_remove2` – Тестирует метод `void remove(T1 key)`. Удаляем элемент с ключом 20 после чего проверяем правильность дерева с помощью обхода и вывода в консоль.
- 4) `test_find` – Тестирует метод `void find(T1 key)`. Находим в массиве значение лежащее в элементе под ключом 40 и проверяем с заданным.
- 5) `test_print_to_console` – проверяем метод `void print()` на корректность и наличие ошибок.
- 6) `test_get_keys` – проверка метода `get_keys()` на корректность и наличие ошибок.
- 7) `test_get_values` – проверка метода `get_values()` на корректность и наличие ошибок.

Пример работы



```
      /2
     /  \
    /3   \1
   /  \  \1
  /  \  \1
 A  D  \1
1  2  \2
```

Рисунок 1 - Пример работы программы

```
#include <iostream>
#include "Node.h"

#include <windows.h>

#include "NormalList.h"
void SetColor(int text, int background)
{
    HANDLE hConsoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(hConsoleHandle, (WORD)((background << 4) | text));
}

using namespace std;

template<typename T1, typename T2>
class Map {

    Node<T1, T2>* root;

    void leftRotate(Node<T1, T2>* x) {
        Node<T1, T2>* nParent = x->right;
        if (x == root)
            root = nParent;
        x->moveDown(nParent);
        x->right = nParent->left;
        if (nParent->left != NULL)
            nParent->left->parent = x;
        nParent->left = x;
    }

    void rightRotate(Node<T1, T2>* x) {

        Node<T1, T2>* nParent = x->left;
        if (x == root)
            root = nParent;

        x->moveDown(nParent);
        x->left = nParent->right;
        if (nParent->right != NULL)
            nParent->right->parent = x;
        nParent->right = x;
    }

    void swapColors(Node<T1, T2>* x1, Node<T1, T2>* x2) {

        COLOR temp;

        temp = x1->color;

        x1->color = x2->color;

        x2->color = temp;
    }
}
```

```

void swapkeyues(Node<T1, T2>* u, Node<T1, T2>* v) {
    int temp;
    temp = u->key;
    u->key = v->key;
    v->key = temp;
}

void fixRedRed(Node<T1, T2>* newElement) {
    if (newElement == root) {
        newElement->color = BLACK;
        return;
    }
    Node<T1, T2>* parent = newElement->parent, * grandparent = parent->parent,
        * uncle = newElement->uncle();

    if (parent->color != 1) {
        if (uncle != NULL && uncle->color == 0) {

            parent->color = BLACK;
            uncle->color = BLACK;
            grandparent->color = RED;
            fixRedRed(grandparent);
        }
        else {
            // Else perform LR, LL, RL, RR
            if (parent->isOnLeft()) {
                if (newElement->isOnLeft()) {
                    // for left right
                    swapColors(parent, grandparent);
                }
                else {
                    leftRotate(parent);
                }
            }
            else {
                if (newElement->isOnRight()) {
                    // for right left
                    swapColors(parent, grandparent);
                }
                else {
                    rightRotate(parent);
                }
            }
        }
    }
}

```

```

        swapColors(newElement, grandparent);

    }

    // for left left and left right
    rightRotate(grandparent);

}
else {
    if (newElement->isOnLeft()) {

        // for right left
        rightRotate(parent);

        swapColors(newElement, grandparent);

    }
    else {

        swapColors(parent, grandparent);

    }

    // for right right and right left
    leftRotate(grandparent);

}

}

}

}

Node<T1, T2>* successor(Node<T1, T2>* x) {
    Node<T1, T2>* temp = x;

    while (temp->left != NULL)
        temp = temp->left;

    return temp;
}

Node<T1, T2>* BSTreplace(Node<T1, T2>* x) {
    if (x->left != NULL && x->right != NULL)
        return successor(x->right);
    if (x->left == NULL && x->right == NULL)

```

```

        return NULL;

    if (x->left != NULL)
        return x->left;
    else
        return x->right;
}

void deleteNode(Node<T1, T2>* v) {
    Node<T1, T2>* u = BSTreplace(v);

    // True when u and v are both 1
    bool uv1 = ((u == NULL || u->color == 1) && (v->color == 1));
    Node<T1, T2>* parent = v->parent;

    if (u == NULL) {
        // u is NULL therefore v is leaf
        if (v == root) {
            // v is root, making root null
            root = NULL;
        }
        else {
            if (uv1) {
                // u and v both black
                // v is leaf, fix double black at v
                fixDoubleBlack(v);
            }
            else {
                // u or v is 0
                if (v->sibling() != NULL)
                    // sibling is not null, make it 0"
                    v->sibling()->color = RED;
            }
        }

        // delete v from the tree
    }
}

```

```

        if (v->isOnLeft()) {
            parent->left = NULL;
        }
        else {
            parent->right = NULL;
        }
    }
    delete v;
    return;
}

if (v->left == NULL || v->right == NULL) {
    // v has 1 child
    if (v == root) {
        // v is root, assign the keyue of u to v, and delete u
        v->key = u->key;
        v->left = v->right = NULL;
        delete u;
    }
    else {
        // Detach v from tree and move u up
        if (v->isOnLeft()) {
            parent->left = u;
        }
        else {
            parent->right = u;
        }
        delete v;
        u->parent = parent;
        if (uv1) {
            // u and v both 1, fix double 1 at u
            fixDoubleBlack(u);
        }
        else {

```



```

        // u or v 0, color u 1
        u->color = BLACK;

    }

}

return;
}

// v has 2 children, swap keyues with successor and recurse
swapkeyues(u, v);
deleteNode(u);
}

void fixDoubleBlack(Node<T1, T2>* x) {
    if (x == root)
        // Reached root
        return;

    Node<T1, T2>* sibling = x->sibling(), * parent = x->parent;
    if (sibling == NULL) {
        // No sibling, double black pushed up
        fixDoubleBlack(parent);
    }
    else {
        if (sibling->color == 0) {
            // Sibling red
            parent->color = RED;
            sibling->color = BLACK;
            if (sibling->isOnLeft()) {
                // left case
                rightRotate(parent);
            }
            else {

```

```

        // right case
        leftRotate(parent);
    }
    fixDoubleBlack(x);
}
else {
    // Sibling 1
    if (sibling->has0Child()) {
        // at least 1 0 children
        if (sibling->left != NULL && sibling->left->color == 0) {
            if (sibling->isOnLeft()) {
                // left left
                sibling->left->color = sibling->color;
                sibling->color = parent->color;
                rightRotate(parent);
            }
            else {
                // right left
                sibling->left->color = parent->color;
                rightRotate(sibling);
                leftRotate(parent);
            }
        }
        else {
            if (sibling->isOnLeft()) {
                // left right
                sibling->right->color = parent->color;
                leftRotate(sibling);
                rightRotate(parent);
            }
            else {
                // right right
                sibling->right->color = sibling->color;
                sibling->color = parent->color;
            }
        }
    }
}

```

```

        leftRotate(parent);
    }
}
parent->color = BLACK;
}
else {
    // 2 1 children
    sibling->color = RED;
    if (parent->color == 1)
        fixDoubleBlack(parent);
    else
        parent->color = BLACK;
}
}
}
}

public:

Map() { root = NULL; }

~Map() { clear(root);
root = NULL;
};

Node<T1, T2>* getRoot() { return root; }

Node<T1, T2>* find(T1 n) {
    Node<T1, T2>* temp = root;
    while (temp != NULL) {
        if (n < temp->key) {
            if (temp->left == NULL)
                break;
            else
                temp = temp->left;
        }
        else if (n == temp->key) {

```

```

        break;
    }
    else {
        if (temp->right == NULL)
            break;
        else
            temp = temp->right;
    }
}

return temp;
}

void insert(T1 key, T2 data) {
    Node<T1, T2>* newNode = new Node<T1, T2>(key, data);
    if (root == NULL) {
        newNode->color = BLACK;
        root = newNode;
    }
    else {
        Node<T1, T2>* temp = find(key);

        newNode->parent = temp;

        if (key < temp->key)
            temp->left = newNode;
        else
            temp->right = newNode;

        fixRedRed(newNode);
    }
}
}

```

```

void remove(T1 n) {
    if (root == NULL)
        // Tree is empty
        return;

    Node<T1, T2>* v = find(n), * u;

    if (v->key != n) {
        cout << "No Node<T1, T2> found to delete with keyue:" << n << endl;
        return;
    }

    deleteNode(v);
}

void print(Node<T1, T2>* root, int lvl)
{
    if (root != NULL)
    {
        print(root->right, lvl + 2);

        for (int i = 0; i < lvl; i++)
        {
            cout << "  ";
        }
        if (root->parent != NULL && root->parent->key >= root->key)
        {
            if (root->color == 0)
                SetColor(12, 0);
            else
                SetColor(9, 0);
            cout << "\\\" << root->key;

            cout << endl;
        }
        else if (root->parent != NULL && root->parent->key < root->key)
        {
            if (root->color == 0)
                SetColor(12, 0);
            else
                SetColor(9, 0);
            cout << "/" << root->key;

            cout << endl;
        }
        else
    }
}

```

```

        {
            SetColor(9, 0);
            cout << root->key;
        }

        cout << endl;

        print(root->left, lvl + 2);
    }
}

List<Node<T1, T2>*>* getList(Node<T1, T2>* root, List<Node<int, int>*>* A)
{
    if (root)
    {
        getList(root->left, A);
        getList(root->right, A);
        A->push_back(root);
    }

    return A;
}

void printList(List<Node<T1, T2>*>* list)
{
    while (!list->isEmpty())
    {
        cout << list->at(0)->data;
        cout << endl;
        list->pop_front();
    }
}

void clear(Node<T1, T2>* root)
{
    if (root)
    {
        clear(root->left);
        clear(root->right);
        delete root;
    }
}

};

int main() {
    Map<char, int> map;

    ifstream input;
    input.open("input.txt");

    char ch;

    while (!input.eof())

```

```

    {
        input.get(ch);
        map.insert(ch, 1);
    }

    map.print(map.getRoot(), 0);

    map.printListKey(map.getListKey(map.getRoot(), new List<char>));
    cout << endl;
    map.printListData(map.getListData(map.getRoot(), new List<int>));

    return 0;
}

```

Node.h

```

#include<iostream>

#pragma once

#ifndef Node_h
#define Node_h

using namespace std;

enum COLOR { RED, BLACK };

template<typename T1, typename T2>
class Node {
public:
    T1 key;
    T2 data;
    COLOR color;
    Node<T1, T2>* left, * right, * parent;

    Node(T1 key, T2 data) : key(key), data(data) {
        parent = left = right = NULL;

        // Node<T1, T2> is created during insertion
        // Node<T1, T2> is 0 at insertion
        color = RED;
    }

    ~Node() = default;

    // returns pointer to uncle

```

```

Node<T1, T2>* uncle() {
    // If no parent or grandparent, then no uncle
    if (parent == NULL || parent->parent == NULL)
        return NULL;

    if (parent->isOnLeft())
        // uncle on right
        return parent->parent->right;
    else
        // uncle on left
        return parent->parent->left;
}

// check if node is left child of parent
bool isOnLeft() { return this == parent->left; }

// returns pointer to sibling
Node<T1, T2>* sibling() {
    // sibling null if no parent
    if (parent == NULL)
        return NULL;

    if (isOnLeft())
        return parent->right;

    return parent->left;
}

// moves node down and moves given node in its place
void moveDown(Node<T1, T2>* nParent) {
    if (parent != NULL) {
        if (isOnLeft()) {

```



```

        parent->left = nParent;
    }
    else {
        parent->right = nParent;
    }
}

nParent->parent = parent;
parent = nParent;
}

bool has0Child() {
    return (left != NULL && left->color == 0) ||
           (right != NULL && right->color == 0);
}

};

#define Node_h
#endif //Node_h

```

Тесты

```

#include "pch.h"
#include "CppUnitTest.h"
#include <string>
#include "../RB/RB.cpp"

using namespace Microsoft::VisualStudio::CppUnitTestFramework;

namespace RBTest
{
    TEST_CLASS(RBTest)
    {
    private:
        Map<int, string> tree;
    public:

        TEST_METHOD_INITIALIZE(init)
        {
            tree.insert(30, "a");
            tree.insert(40, "a");
            tree.insert(15, "daada");
            tree.insert(10, "daada");
            tree.insert(20, "daada");
            tree.insert(5, "daada");
            tree.insert(3, "daada");
        }
    }
}

```

```

    }
    TEST_METHOD(test_insert)
    {
        tree.insert(2, "oleg");
        tree.insert(21, "ne sdast labu");
        Assert::AreEqual(tree.getRoot()->left->left->left->key, 2);
        Assert::AreEqual(tree.getRoot()->right->left->right->key, 21);
    }
    TEST_METHOD(test_remove1)
    {
        tree.remove(15);
        Assert::AreEqual(tree.getRoot()->left->key, 5);
    }
    TEST_METHOD(test_remove2)
    {
        tree.remove(20);
        Assert::AreEqual(tree.getRoot()->left->key, 5);
    }
    TEST_METHOD(find)
    {
        Assert::AreEqual(tree.find(40)->data, (string)"a");
    }
    TEST_METHOD(test_print_to_console)
    {
        tree.print(tree.getRoot(), 0);
    }
};

```

NormalList.h

```

template<typename T>
class List
{
private:
    class Node
    {
    private:
        T data;
        Node* next, * prev;
    public:
        Node() : next(NULL), prev(NULL) {};
        Node(T data) {
            this->data = data;
            next = NULL;
            prev = NULL;
        }
        ~Node() {
            next = NULL;
            prev = NULL;
        }
        void set_data(T data) {
            this->data = data;
        }
        T get_data() {
            return data;
        }
        Node* get_next() {
            return next;
        }
        Node* get_prev() {
            return prev;
        }
    }
};

```

```

        void set_next(Node* pointer) {
            next = pointer;
        }
        void set_prev(Node* pointer) {
            prev = pointer;
        }
    };
    Node* head, * tail;
    Node* get_pointer(size_t index)
    {
        if (isEmpty() || (index > get_size() - 1))
        {
            throw out_of_range("Invalid argument");
        }
        else if (index == get_size() - 1)
            return tail;
        else if (index == 0)
            return head;
        else
        {
            Node* temp = head;
            while ((temp) && (index--))
            {
                temp = temp->get_next();
            }
            return temp;
        }
    }
public:
    List() : head(NULL), tail(NULL) {}
    List(const List<T>& list) {
        clear();
        Node* temp = list.head;
        while (temp) {
            push_back(temp->get_data());
            temp = temp->get_next();
        }
    }
    ~List()
    {
        while (head)
        {
            tail = head->get_next();
            delete head;
            head = tail;
        }
        head = NULL;
    }
    void push_back(T data)
    {
        Node* temp = new Node;
        temp->set_data(data);
        if (head)
        {
            temp->set_prev(tail);
            tail->set_next(temp);
            tail = temp;
        }
        else
        {
            head = temp;
            tail = head;
        }
    }

```

```

}
void push_front(T data)
{
    Node* temp = new Node;
    temp->set_data(data);
    if (head)
    {
        temp->set_next(head);
        head->set_prev(temp);
        head = temp;
    }
    else
    {
        head = temp;
        tail = head;
    }
}
void push_front(List& ls2)
{
    Node* temp = ls2.tail;
    size_t n = ls2.get_size();
    while ((temp) && (n--))
    {
        push_front(temp->get_data());
        temp = temp->get_prev();
    }
}
void pop_back()
{
    if (head != tail)
    {
        Node* temp = tail;
        tail = tail->get_prev();
        tail->set_next(NULL);
        delete temp;
    }
    else if (!isEmpty())
    {
        Node* temp = tail;
        tail = head = NULL;
        delete temp;
    }
    else
        throw out_of_range("The list is empty");
}
void pop_front()
{
    if (head != tail)
    {
        Node* temp = head;
        head = head->get_next();
        head->set_prev(NULL);
        delete temp;
    }
    else if (!isEmpty())
    {
        Node* temp = head;
        head = tail = NULL;
        delete temp;
    }
    else
        throw out_of_range("The list is empty");
}

```

```

void insert(size_t index, T data)
{
    Node* temp;
    temp = get_pointer(index);
    if (temp == head)
        push_front(data);
    else
    {
        Node* newElem = new Node;
        newElem->set_data(data);
        temp->get_prev()->set_next(newElem);
        newElem->set_prev(temp->get_prev());
        newElem->set_next(temp);
        temp->set_prev(newElem);
    }
}

T at(size_t index)
{
    Node* temp;
    temp = get_pointer(index);
    return temp->get_data();
}

void remove(size_t index)
{
    Node* temp;
    temp = get_pointer(index);
    if (temp == head)
        pop_front();
    else if (temp == tail)
        pop_back();
    else
    {
        temp->get_prev()->set_next(temp->get_next());
        temp->get_next()->set_prev(temp->get_prev());
        delete temp;
    }
}

void remove(T data) {
    Node* temp = head;
    while (temp && temp->get_data() != data)
        temp = temp->get_next();
    if (!temp)
        throw out_of_range("Invalid argument");
    if (temp == head)
        pop_front();
    else if (temp == tail)
        pop_back();
    else
    {
        temp->get_prev()->set_next(temp->get_next());
        temp->get_next()->set_prev(temp->get_prev());
        delete temp;
    }
}

size_t get_size()
{
    Node* temp = head;
    size_t length = 0;
    while (temp)
    {
        length++;
        temp = temp->get_next();
    }
}

```

```

        return length;
    }
    void print_to_console()
    {
        Node* temp = head;
        while (temp)
        {
            cout << temp->get_data() << ' ';
            temp = temp->get_next();
        }
        cout << endl;
    }
    void clear()
    {
        while (head)
        {
            tail = head->get_next();
            delete head;
            head = tail;
        }
    }
    void set(size_t index, T data)
    {
        Node* temp;
        temp = get_pointer(index);
        temp->set_data(data);
    }
    bool isEmpty()
    {
        if (!head)
            return true;
        else
            return false;
    }
};

```

Вывод

В ходе выполнения работы я научился реализовывать класс ассоциативного массива на основе красно-черного двоичного дерева поиска.