**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра САПР**

**ОТЧЕТ**

**по лабораторной работе №3**

**по дисциплине «АиСД»**

**Тема: Алгоритмы на графах**

|  |  |  |
| --- | --- | --- |
| Студент гр. 8302 |  | Смирнов Е.Д. |
| Преподаватель |  | Тутуева А.В. |

**Оглавление**

[Постановка задачи 2](#_Toc37951820)

[Описание классов и методов 2](#_Toc37951821)

[Оценка временной сложности 3](#_Toc37951822)

[Описание unit-тестов 3](#_Toc37951823)

[Пример работы 4](#_Toc37951824)

[Код 5](#_Toc37951825)

## Постановка задачи

***Задача:***  Дан список возможных авиарейсов в текстовом файле в формате:

Город отправления 1;Город прибытия 1;цена прямого перелета 1;цена обратного перелета 1

*Город отправления 2;Город прибытия 2;цена перелета 2;цена обратного перелета 1*

*…*

*Город отправления N;Город прибытия N;цена перелета N;цена обратного перелета N*

*В случае, если нет прямого или обратного рейса, его цена будет указана как N/A (not available)*

Требуется найти наиболее дешевый путь из i в j используя алгоритм Флойда и матрицы смежности.

## Описание классов и методов

Реализованы следующие классы:

1. Graph – граф, содержит матрицу смежности, массив вершин для быстрого доступа по индексу и ассоциативный массив вершин для быстрого доступа по названию города. Граф позволяет получать вершины по имени и по индексу, соединять вершины, узнавать вес ребра между двумя вершинами и читать граф из файла.
2. Graph::Vertex – вершина графа, содержит имя и свой индекс в графе.
3. array – динамичный массив, автоматически выделяет больше памяти при ее нехватке, позволяет получать доступ к элементам за константное время. Реализует операции копирования, переноса, методы добавления, вывода и очистки. Используется для хранения матрицы смежности и вершин графа, а так же для построения путей.

Из лабораторной работы №1 были использованы классы rb\_map<K, V> и list<T>.

## Оценка временной сложности

## Доступ к массиву и его длине, добавление в массив – O(1), копирование массива и выделение памяти – O(N)

## Сложность алгоритма Флойда – O(V^2 \* E) – сложность самого алгоритма O(VE), однако при построении путей их копирование происходит за линейное время.

## Описание unit-тестов

Сначала проверяется ассоциативный массив под большой случайно нагрузкой – этот тест взят из лабораторной работы №1. После проверяется динамичный массив добавлением и проверкой большого кол-ва элементов.

Далее производится следующий тест: генерируется полностью связный граф с 250 вершинами и с весами всех ребер = 1, в нем производится поиск кратчайших путей от первой вершины, после чего проверяется, что все они имеют длину 1 и содержат 1 ребро.

## Пример работы

graph:

Adjacent to A: B=10 C=25 D=5 F=100

Adjacent to B: C=20

Adjacent to C: A=10 D=25 E=5

Adjacent to D: A=10 C=15

Adjacent to E: A=5

Adjacent to F: A=100 D=25

type 1 to find shortest path between cities

type 2 to show all shortest paths from city

type 3 to read file

type any other number to exit

command:1

input departure city:B

input arrival city:D

path from B to D (weight=35): B C A D

press enter to continue...

graph:

Adjacent to A: B=10 C=25 D=5 F=100

Adjacent to B: C=20

Adjacent to C: A=10 D=25 E=5

Adjacent to D: A=10 C=15

Adjacent to E: A=5

Adjacent to F: A=100 D=25

type 1 to find shortest path between cities

type 2 to show all shortest paths from city

type 3 to read file

type any other number to exit

command:2

input departure city:D

path from D to A (weight=10): D A

path from D to B (weight=20): D A B

path from D to C (weight=15): D C

path from D to D (weight=0): D

path from D to E (weight=20): D C E

path from D to F (weight=110): D A F

press enter to continue...

## Код

main.cpp

#include "graph.h"  
#include "pathfinding.h"  
  
**int** main() {  
 Graph mGraph;  
 mGraph.read("../../lab3/data.txt");  
  
 **while** (**true**) {  
 std::cout << "\n\ngraph: \n";  
 mGraph.print();  
 std::cout << "\n\ntype 1 to find shortest path between cities\ntype 2 to show all shortest paths from city\ntype 3 to read file\ntype any other number to exit\ncommand: ";  
  
 **int** choice = 1;  
 std::cin >> choice;  
 **switch** (choice) {  
 **case** 1: {  
 std::string name;  
 std::cout << "input departure city: ";  
 std::cin >> name;  
 **auto** vertex1 = mGraph.get\_vertex(name);  
 std::cout << "input arrival city: ";  
 std::cin >> name;  
 **auto** vertex2 = mGraph.get\_vertex(name);  
 **if** (vertex1 != **nullptr** && vertex2 != **nullptr**) {  
 **auto** path = bellman\_floid\_algorithm::get\_shortest\_paths(mGraph, \*vertex1)[vertex2->index];  
 **if** (path.weight != Graph::INF) {  
 std::cout << "path from " << vertex1->name << " to " << vertex2->name << " (weight=" << path.weight << "): ";  
 **for** (**int** j = 0; j < path.path.length(); j++) {  
 std::cout << path.path[j]->name << " ";  
 }  
 std::cout << "\n";  
 } **else** {  
 std::cout << "no path found\n";  
 }  
 } **else** {  
 std::cout << "invalid city name(s)\n";  
 }  
 std::cout << "press enter to continue...";  
 getchar();  
 getchar();  
 **break**;  
 }  
 **case** 2: {  
 std::string name;  
 std::cout << "input departure city: ";  
 std::cin >> name;  
 Graph::Vertex\* start\_vertex = mGraph.get\_vertex(name);  
 **if** (start\_vertex != **nullptr**) {  
 **auto** paths = bellman\_floid\_algorithm::get\_shortest\_paths(mGraph, \*start\_vertex);  
 **for** (**int** i = 0; i < paths.length(); i++) {  
 **auto** path = paths[i];  
 std::cout << "path from " << name << " to " << mGraph.get\_vertex(i)->name << " (weight=" << path.weight << "): ";  
 **for** (**int** j = 0; j < path.path.length(); j++) {  
 std::cout << path.path[j]->name << " ";  
 }  
 std::cout << "\n";  
 }  
 } **else** {  
 std::cout << "invalid city name(s)\n";  
 }  
 std::cout << "press enter to continue...";  
 getchar();  
 getchar();  
 **break**;  
 }  
 **case** 3: {  
 std::string name;  
 std::cout << "input filename: ";  
 std::cin >> name;  
 std::cout << (mGraph.read(name) ? "success" : "fail") << "\n";  
 std::cout << "press enter to continue...";  
 getchar();  
 getchar();  
 **break**;  
 }  
 **default**:  
 **return** 0;  
 }  
 }  
}

tests.cpp

#include "gtest/gtest.h"  
#include "rb\_map.h"  
#include "array.h"  
#include "graph.h"  
#include "pathfinding.h"  
  
// graph tests  
TEST (graph, fully\_connected\_pathfinding) {  
 Graph mGraph;  
 **const int** size = 25 \* 10;  
 **for** (**int** i = 0; i < size; i++) {  
 **char** name[] = {**char**('A' + i % 25), **char**('A' + (i / 25) % 25), '\0'};  
 mGraph.get\_or\_make\_vertex(name);  
 }  
 **for** (**int** i = 0; i < size; i++) {  
 **for** (**int** j = 0; j < size; j++) {  
 mGraph.connect(i, j, 1);  
 }  
 }  
  
 **auto** paths = bellman\_floid\_algorithm::get\_shortest\_paths(mGraph, \*mGraph.get\_vertex(0));  
 **for** (**int** i = 1; i < paths.length(); i++) {  
 ASSERT\_EQ(paths[i].weight, 1);  
 ASSERT\_EQ(paths[i].path.length(), 2);  
 }  
}  
  
// array tests  
TEST (array, massive\_add\_and\_check) {  
 array<std::string> arr;  
 **for** (**int** i = 0; i < 1000000; i++) {  
 arr.add(i % 2 ? "abc" : "edf");  
 }  
 ASSERT\_EQ(arr.length(), 1000000);  
 **for** (**int** i = 0; i < 1000000; i++) {  
 ASSERT\_EQ(arr[i], i % 2 ? "abc" : "edf");  
 }  
 arr.clear();  
 ASSERT\_EQ(arr.length(), 0);  
}  
  
// keep hardest test for rb\_map  
TEST (rb\_map, massive\_random\_load) {  
 rb\_map<**int**, **int**> map;  
 **for** (**int** i = 0; i < 1000000; i++) {  
 map[rand() % 1000000] = rand();  
 }  
 ASSERT\_EQ(map.length(), map.tree\_size());  
  
 **int** remaining\_length = map.length();  
 **for** (**int** i = 0; i < 1000000; i++) {  
 **if** (map.remove(rand() % 1000000)) {  
 remaining\_length--;  
 }  
 }  
  
 ASSERT\_EQ(map.length(), map.tree\_size());  
 ASSERT\_EQ(map.length(), remaining\_length);  
}

array.h

#include <iostream>  
  
  
#ifndef M\_ARRAY\_H  
#define M\_ARRAY\_H  
  
**template** <**typename** T>  
**class** array {  
 T\* allocated\_memory = **nullptr**;  
 **int** allocated\_memory\_size = 0;  
  
 **int** size = 0;  
  
 **void** ensure\_size(**int** size) {  
 **if** (allocated\_memory\_size < size) {  
 **int** old\_size = allocated\_memory\_size;  
 **while** (allocated\_memory\_size < size) {  
 **if** (allocated\_memory\_size < 32) {  
 allocated\_memory\_size += 16;  
 } **else** {  
 allocated\_memory\_size \*= 2;  
 }  
 }  
  
 **if** (allocated\_memory != **nullptr**) {  
 T\* new\_memory = **new** T[allocated\_memory\_size];  
 **for** (**int** i = 0; i < old\_size; i++) {  
 new\_memory[i] = allocated\_memory[i];  
 }  
 **delete**[] (allocated\_memory);  
 allocated\_memory = new\_memory;  
 } **else** {  
 allocated\_memory = **new** T[allocated\_memory\_size];  
 }  
 }  
 }  
  
**public**:  
 array() = **default**;  
  
 array(**int** size) {  
 resize(size);  
 }  
  
 array(array<T> **const**& other) {  
 ensure\_size(other.size);  
 size = other.size;  
 **for** (**int** i = 0; i < size; i++) {  
 allocated\_memory[i] = other.allocated\_memory[i];  
 }  
 }  
  
 array(array<T>&& other) **noexcept** {  
 allocated\_memory = other.allocated\_memory;  
 size = other.size;  
 allocated\_memory\_size = other.allocated\_memory\_size;  
 other.allocated\_memory = **nullptr**;  
 other.size = 0;  
 other.allocated\_memory\_size = 0;  
 }  
  
 array<T>& **operator**= (array<T> **const**& other) {  
 ensure\_size(other.size);  
 size = other.size;  
 **for** (**int** i = 0; i < size; i++) {  
 allocated\_memory[i] = other.allocated\_memory[i];  
 }  
 **return** \***this**;  
 }  
  
 array<T>& **operator**= (array<T>&& other) **noexcept** {  
 allocated\_memory = other.allocated\_memory;  
 size = other.size;  
 allocated\_memory\_size = other.allocated\_memory\_size;  
 other.allocated\_memory = **nullptr**;  
 other.size = 0;  
 other.allocated\_memory\_size = 0;  
 **return** \***this**;  
 }  
  
 **int** length() {  
 **return** size;  
 }  
  
 T& **operator**[] (**int** index) {  
 **return** allocated\_memory[index];  
 }  
  
 **void** resize(**int** new\_size) {  
 **if** (new\_size == 0) {  
 clear();  
 } **else** {  
 ensure\_size(new\_size);  
 size = new\_size;  
 }  
 }  
  
 **void** add(T **const**& elem) {  
 ensure\_size(size + 1);  
 allocated\_memory[size++] = elem;  
 }  
  
 **void** clear() {  
 **if** (allocated\_memory != **nullptr**) {  
 **delete**[] (allocated\_memory);  
 allocated\_memory = **nullptr**;  
 }  
 size = allocated\_memory\_size = 0;  
 }  
  
 ~array() {  
 clear();  
 }  
};  
  
#endif

graph.h

#include <fstream>  
#include <string.h>  
  
#include "array.h"  
#include "rb\_map.h"  
  
  
#ifndef M\_GRAPH\_H  
#define M\_GRAPH\_H  
  
**class** Graph {  
**public**:  
 **static const double constexpr** INF = -1;  
  
 **class** Vertex {  
 **public**:  
 **int** index;  
 std::string name;  
  
 Vertex();  
 Vertex(std::string **const**& name, **int** index);  
 };  
  
**private**:  
 array<Vertex> vertices;  
 rb\_map<std::string, Vertex\*> vertex\_map; // map for faster access  
 array<array<**double**>> weights;  
  
**public**:  
 **void** connect(**int** index1, **int** index2, **double** weight);  
 **void** connect(Vertex& v1, Vertex& v2, **double** weight);  
 **double** get\_weight(**int** index1, **int** index2);  
 **double** get\_weight(Vertex& v1, Vertex& v2);  
 Vertex\* get\_vertex(**int** index);  
 Vertex\* get\_vertex(std::string **const**& name);  
 Vertex& get\_or\_make\_vertex(std::string **const**& name);  
 **int** size();  
 **void** clear();  
 **bool** read(std::string filename);  
 **void** print();  
};  
  
#endif

graph.cpp

#include "graph.h"  
#include "pathfinding.h"  
  
**int** main() {  
 Graph mGraph;  
 mGraph.read("../../lab3/data.txt");  
  
 **while** (**true**) {  
 std::cout << "\n\ngraph: \n";  
 mGraph.print();  
 std::cout << "\n\ntype 1 to find shortest path between cities\ntype 2 to show all shortest paths from city\ntype 3 to read file\ntype any other number to exit\ncommand: ";  
  
 **int** choice = 1;  
 std::cin >> choice;  
 **switch** (choice) {  
 **case** 1: {  
 std::string name;  
 std::cout << "input departure city: ";  
 std::cin >> name;  
 **auto** vertex1 = mGraph.get\_vertex(name);  
 std::cout << "input arrival city: ";  
 std::cin >> name;  
 **auto** vertex2 = mGraph.get\_vertex(name);  
 **if** (vertex1 != **nullptr** && vertex2 != **nullptr**) {  
 **auto** path = bellman\_floid\_algorithm::get\_shortest\_paths(mGraph, \*vertex1)[vertex2->index];  
 **if** (path.weight != Graph::INF) {  
 std::cout << "path from " << vertex1->name << " to " << vertex2->name << " (weight=" << path.weight << "): ";  
 **for** (**int** j = 0; j < path.path.length(); j++) {  
 std::cout << path.path[j]->name << " ";  
 }  
 std::cout << "\n";  
 } **else** {  
 std::cout << "no path found\n";  
 }  
 } **else** {  
 std::cout << "invalid city name(s)\n";  
 }  
 std::cout << "press enter to continue...";  
 getchar();  
 getchar();  
 **break**;  
 }  
 **case** 2: {  
 std::string name;  
 std::cout << "input departure city: ";  
 std::cin >> name;  
 Graph::Vertex\* start\_vertex = mGraph.get\_vertex(name);  
 **if** (start\_vertex != **nullptr**) {  
 **auto** paths = bellman\_floid\_algorithm::get\_shortest\_paths(mGraph, \*start\_vertex);  
 **for** (**int** i = 0; i < paths.length(); i++) {  
 **auto** path = paths[i];  
 std::cout << "path from " << name << " to " << mGraph.get\_vertex(i)->name << " (weight=" << path.weight << "): ";  
 **for** (**int** j = 0; j < path.path.length(); j++) {  
 std::cout << path.path[j]->name << " ";  
 }  
 std::cout << "\n";  
 }  
 } **else** {  
 std::cout << "invalid city name(s)\n";  
 }  
 std::cout << "press enter to continue...";  
 getchar();  
 getchar();  
 **break**;  
 }  
 **case** 3: {  
 std::string name;  
 std::cout << "input filename: ";  
 std::cin >> name;  
 std::cout << (mGraph.read(name) ? "success" : "fail") << "\n";  
 std::cout << "press enter to continue...";  
 getchar();  
 getchar();  
 **break**;  
 }  
 **default**:  
 **return** 0;  
 }  
 }  
}

pathfinding.h

#include "array.h"  
#include "graph.h"  
  
  
#ifndef M\_PATHFINDING\_H  
#define M\_PATHFINDING\_H  
  
**namespace** bellman\_floid\_algorithm {  
 **struct** graph\_path {  
 array<Graph::Vertex\*> path;  
 **double** weight = Graph::INF;  
 };  
  
 array<graph\_path> get\_shortest\_paths(Graph& graph, Graph::Vertex& offset);  
};  
  
  
#endif

pathfinding.cpp

#include "pathfinding.h"  
  
array<bellman\_floid\_algorithm::graph\_path> bellman\_floid\_algorithm::get\_shortest\_paths(Graph &graph, Graph::Vertex &offset) {  
 array<graph\_path> paths(graph.size());  
 **for** (**int** i = 0; i < paths.length(); i++) {  
 paths[i].path.add(&offset);  
 }  
 paths[offset.index].weight = 0;  
  
 **for** (**int** i = 1; i < graph.size() - 1; i++) {  
 **for** (**int** edge\_start = 0; edge\_start < graph.size(); edge\_start++) {  
 **for** (**int** edge\_end = 0; edge\_end < graph.size(); edge\_end++) {  
 **double** weight = graph.get\_weight(edge\_start, edge\_end);  
 **if** (weight != Graph::INF && paths[edge\_start].weight != Graph::INF &&  
 (paths[edge\_end].weight == Graph::INF ||  
 paths[edge\_end].weight > paths[edge\_start].weight + weight)) {  
 paths[edge\_end].path = paths[edge\_start].path;  
 paths[edge\_end].path.add(graph.get\_vertex(edge\_end));  
 paths[edge\_end].weight = paths[edge\_start].weight + weight;  
 }  
 }  
 }  
 }  
 **return** paths;  
}

list.h

#include <string>  
  
  
#ifndef M\_LIST\_H  
#define M\_LIST\_H  
  
**template** <**typename** T>  
**class** list {  
**public**:  
 **class** list\_node {  
 **public**:  
 T value;  
 list\_node\* next = **nullptr**;  
 list\_node\* prev = **nullptr**;  
  
 list\_node() {};  
 list\_node(T **const**& v) : value(v) {}  
 };  
  
 **class** iterator {  
 **public**:  
 list\_node\* node = **nullptr**;  
  
 iterator() {}  
 iterator(list\_node\* n) : node(n) {}  
  
 iterator **operator**++(**int**) {  
 list\_node\* last = node;  
 node = node->next;  
 **return** iterator(last);  
 }  
  
 iterator **operator**--(**int**) {  
 list\_node\* last = node;  
 node = node->prev;  
 **return** iterator(last);  
 }  
  
 T& **operator**\*() {  
 **return** node->value;  
 }  
  
 **bool operator**==(iterator **const**& it) {  
 **return** it.node == node;  
 }  
  
 **bool operator**!=(iterator **const**& it) {  
 **return** it.node != node;  
 }  
 };  
  
**private**:  
 list\_node\* first = **nullptr**;  
 list\_node\* last = **nullptr**;  
 list\_node list\_end;  
 **int** length = 0;  
  
**public**:  
 list() = **default**;  
  
 list(list **const**& other) {  
 **for** (**auto** it = other.begin(); it != other.**end**(); it++) {  
 add(\*it);  
 }  
 }  
  
 list(list&& other) {  
 first = other.first;  
 last = other.last;  
 length = other.length;  
 other.first = other.last = **nullptr**;  
 other.length = 0;  
 }  
  
 list& **operator**= (list **const**& other) {  
 **for** (**auto** it = other.begin(); it != other.**end**(); it++) {  
 add(\*it);  
 }  
 **return** \***this**;  
 }  
  
 list& **operator**= (list&& other) {  
 first = other.first;  
 last = other.last;  
 length = other.length;  
 other.first = other.last = **nullptr**;  
 other.length = 0;  
 **return** \***this**;  
 }  
  
 iterator begin() {  
 **return** first != **nullptr** ? iterator(first) : **end**();  
 }  
  
 iterator **end**() {  
 **return** iterator(&list\_end);  
 }  
  
 iterator add(T **const**& value) {  
 list\_node\* node = **new** list\_node(value);  
 **if** (last != **nullptr**) {  
 last->next = node;  
 node->prev = last;  
 } **else** {  
 first = last = node;  
 }  
 list\_end.prev = node;  
 node->next = &list\_end;  
 last = node;  
 length++;  
 **return** iterator(node);  
 }  
  
 **void** erase(iterator iterator) {  
 list\_node\* node = iterator.node;  
 **if** (node->next == &list\_end) {  
 last = list\_end.prev = node->prev;  
 }  
 **if** (node->prev == **nullptr**) {  
 first = node->next;  
 } **else** {  
 node->prev->next = node->next;  
 }  
 node->next->prev = node->prev;  
 length--;  
 **delete** (node);  
 }  
  
 **void** print() {  
 std::cout << "[";  
 **for** (**auto** it = begin(); it != **end**(); it++) {  
 std::cout << \*it << ", ";  
 }  
 std::cout << "]";  
 }  
  
 **void** clear() {  
 list\_node\* node = first;  
 **while** (node != **nullptr** && node != &list\_end) {  
 list\_node\* next = node->next;  
 **delete**(node);  
 node = next;  
 }  
 list\_end.next = list\_end.prev = **nullptr**;  
 first = last = **nullptr**;  
 length = 0;  
 }  
  
 **int** get\_length() {  
 **return** length;  
 }  
  
 ~list() {  
 clear();  
 }  
};  
  
#endif

rb\_map.h

#include <iostream>  
#include "list.h"  
  
  
#ifndef M\_MAP\_H  
#define M\_MAP\_H  
  
**template** <**typename** K, **typename** V>  
**class** rb\_map {  
**public**:  
 **class** rb\_tree {  
 **public**:  
 **enum** node\_color : **int** {  
 *BLACK* = 0,  
 *RED* = 1  
 };  
  
 // red-black tree node  
 **class** rb\_node {  
 **public**:  
 **typedef typename** list<K>::iterator key\_iter;  
 **typedef typename** list<V>::iterator value\_iter;  
  
 K key;  
 key\_iter key\_iterator;  
 value\_iter value\_iterator;  
 node\_color color = *BLACK*;  
  
 rb\_node\* left = **nullptr**;  
 rb\_node\* right = **nullptr**;  
 rb\_node\* parent = **nullptr**;  
  
 rb\_node(key\_iter key\_iter, value\_iter value\_iter) {  
 **this**->key = \*key\_iter;  
 key\_iterator = key\_iter;  
 value\_iterator = value\_iter;  
 }  
  
 V& **operator**\*() {  
 **return** \*value\_iterator;  
 }  
  
 ~rb\_node() {  
 **delete**(right);  
 **delete**(left);  
 }  
  
 **int** get\_size() {  
 **return** 1 + (left != **nullptr** ? left->get\_size() : 0) + (right != **nullptr** ? right->get\_size() : 0);  
 }  
  
 **void** show\_tree(**int** depth = 0) {  
 **if** (left != **nullptr**) {  
 left->show\_tree(depth + 1);  
 }  
 **for** (**int** i = 0; i < depth; i++) {  
 std::cout << " ";  
 }  
 std::cout << key << ":" << \*value\_iterator << (color == *RED* ? "[R]" : "[B]") << "\n";  
 **if** (right != **nullptr**) {  
 right->show\_tree(depth + 1);  
 }  
 }  
  
 **void** print() {  
 **if** (left != **nullptr**) {  
 left->print();  
 }  
 std::cout << key << ": " << \*value\_iterator << ", ";  
 **if** (right) {  
 right->print();  
 }  
 }  
 };  
  
 rb\_node\* root = **nullptr**;  
  
 **int** get\_size() {  
 **return** root != **nullptr** ? root->get\_size() : 0;  
 }  
  
 ~rb\_tree() {  
 **delete**(root);  
 }  
  
 **void** clear() {  
 **delete**(root);  
 root = **nullptr**;  
 }  
  
 **void** show\_tree() {  
 **if** (root != **nullptr**) {  
 root->show\_tree();  
 } **else** {  
 std::cout << "empty tree\n";  
 }  
 }  
  
 rb\_node\* get\_node(K key) {  
 rb\_node\* node = root;  
 **while** (node != **nullptr**) {  
 **if** (node->key == key) {  
 **return** node;  
 }  
 **if** (node->key < key) {  
 node = node->right;  
 } **else** {  
 node = node->left;  
 }  
 }  
 **return nullptr**;  
 }  
  
 **void** left\_rotate(rb\_node\* node) {  
 rb\_node\* tmp = node->right;  
 node->right = tmp->left;  
 **if** (tmp->left != **nullptr**) {  
 tmp->left->parent = node;  
 }  
 tmp->parent = node->parent;  
  
 **if** (node->parent == **nullptr**) {  
 root = tmp;  
 } **else** {  
 **if** (node == node->parent->left) {  
 node->parent->left = tmp;  
 } **else** {  
 node->parent->right = tmp;  
 }  
 }  
 tmp->left = node;  
 node->parent = tmp;  
 }  
  
 **void** right\_rotate(rb\_node\* node) {  
 rb\_node\* tmp = node->left;  
 node->left = tmp->right;  
 **if** (tmp->right != **nullptr**) {  
 tmp->right->parent = node;  
 }  
 tmp->parent = node->parent;  
  
 **if** (node->parent == **nullptr**) {  
 root = tmp;  
 } **else** {  
 **if** (node == node->parent->left) {  
 node->parent->left = tmp;  
 } **else** {  
 node->parent->right = tmp;  
 }  
 }  
 tmp->right = node;  
 node->parent = tmp;  
 }  
  
 **void** insert\_fixup(rb\_node\* x) {  
 **while** (x->parent != **nullptr** && x->parent->color == *RED*) {  
 **if** (x->parent == x->parent->parent->left) {  
 rb\_node\* y = x->parent->parent->right;  
 **if** (y != **nullptr** && y->color == *RED*) {  
 x->parent->color = *BLACK*;  
 y->color = *BLACK*;  
 x->parent->parent->color = *RED*;  
 x = x->parent->parent;  
 } **else** {  
 **if** (x == x->parent->right) {  
 x = x->parent;  
 left\_rotate(x);  
 }  
 x->parent->color = *BLACK*;  
 x->parent->parent->color = *RED*;  
 right\_rotate(x->parent->parent);  
 }  
 } **else** {  
 rb\_node\* y = x->parent->parent->left;  
 **if** (y != **nullptr** && y->color == *RED*) {  
 x->parent->color = *BLACK*;  
 y->color = *BLACK*;  
 x->parent->parent->color = *RED*;  
 x = x->parent->parent;  
 } **else** {  
 **if** (x == x->parent->left) {  
 x = x->parent;  
 right\_rotate(x);  
 }  
 x->parent->color = *BLACK*;  
 x->parent->parent->color = *RED*;  
 left\_rotate(x->parent->parent);  
 }  
 }  
 }  
 root->color = *BLACK*;  
 }  
  
 **bool** insert(rb\_node\* node) {  
 rb\_node\* last\_node = **nullptr**;  
 rb\_node\* current\_node = root;  
 **while** (current\_node != **nullptr**) {  
 last\_node = current\_node;  
 **if** (node->key == current\_node->key) {  
 \*current\_node->value\_iterator = \*node->value\_iterator;  
 **return false**;  
 }  
 **if** (node->key < current\_node->key) {  
 current\_node = current\_node->left;  
 } **else** {  
 current\_node = current\_node->right;  
 }  
 }  
 node->parent = last\_node;  
 **if** (last\_node == **nullptr**) {  
 root = node;  
 } **else if** (node->key < last\_node->key) {  
 last\_node->left = node;  
 } **else** {  
 last\_node->right = node;  
 }  
 node->left = node->right = **nullptr**;  
 node->color = *RED*;  
 insert\_fixup(node);  
 **return true**;  
 }  
  
 **void** remove\_fixup(rb\_node\* x) {  
 **while** (x != root && (x == **nullptr** || x->color == *BLACK*)) {  
 **if** (x == x->parent->left) {  
 rb\_node\* y = x->parent->right;  
 **if** (y != **nullptr** && y->color == *RED*) {  
 y->color = *BLACK*;  
 x->parent->color = *RED*;  
 left\_rotate(x->parent);  
 y = x->parent->right;  
 }  
 **if** (y == **nullptr**) {  
 **break**;  
 }  
 **if** ((y->left == **nullptr** || y->left->color == *BLACK*) &&  
 (y->right == **nullptr** || y->right->color == *BLACK*)) {  
 y->color = *RED*;  
 x = x->parent;  
 } **else** {  
 **if** (y->right == **nullptr** || y->right->color == *BLACK*) {  
 y->left->color = *BLACK*;  
 y->color = *RED*;  
 right\_rotate(y);  
 y = x->parent->right;  
 }  
 y->color = x->parent->color;  
 x->parent->color = *BLACK*;  
 y->right->color = *BLACK*;  
 left\_rotate(x->parent);  
 x = root;  
 }  
 } **else** {  
 rb\_node\* y = x->parent->left;  
 **if** (y != **nullptr** && y->color == *RED*) {  
 y->color = *BLACK*;  
 x->parent->color = *RED*;  
 right\_rotate(x->parent);  
 y = x->parent->left;  
 }  
 **if** (y == **nullptr**) {  
 **break**;  
 }  
 **if** ((y->left == **nullptr** || y->left->color == *BLACK*) &&  
 (y->right == **nullptr** || y->right->color == *BLACK*)) {  
 y->color = *RED*;  
 x = x->parent;  
 } **else** {  
 **if** (y->left == **nullptr** || y->left->color == *BLACK*) {  
 y->right->color = *BLACK*;  
 y->color = *RED*;  
 left\_rotate(y);  
 y = x->parent->left;  
 }  
 y->color = x->parent->color;  
 x->parent->color = *BLACK*;  
 y->left->color = *BLACK*;  
 right\_rotate(x->parent);  
 x = root;  
 }  
  
 }  
 }  
 }  
  
 rb\_node\* tree\_successor(rb\_node\* node) {  
 **if** (node->right != **nullptr**) {  
 **while** (node->left != **nullptr**) {  
 node = node->left;  
 }  
 **return** node;  
 }  
 rb\_node\* tmp = node->parent;  
 **while** (tmp != **nullptr** && node == tmp->right) {  
 node = tmp;  
 tmp = tmp->parent;  
 }  
 **return** tmp;  
 }  
  
 rb\_node\* remove(rb\_node\* node) {  
 rb\_node\* y;  
 **if** (node->left == **nullptr** || node->right == **nullptr**) {  
 y = node;  
 } **else** {  
 y = tree\_successor(node);  
 }  
 **if** (y == **nullptr**) {  
 show\_tree();  
 std::cout << " " << node->key << " ";  
 }  
 rb\_node\* x;  
 **if** (y->left != **nullptr**) {  
 x = y->left;  
 } **else** {  
 x = y->right;  
 }  
  
 **if** (x != **nullptr**) {  
 x->parent = y->parent;  
 }  
 **if** (y->parent == **nullptr**) {  
 root = x;  
 } **else** {  
 **if** (y == y->parent->left) {  
 y->parent->left = x;  
 } **else** {  
 y->parent->right = x;  
 }  
 }  
 **if** (y != node) {  
 node->key = y->key;  
 \*node->value\_iterator = \*y->value\_iterator;  
 }  
 **if** (y->color == *BLACK* && x != **nullptr**) {  
 remove\_fixup(x);  
 }  
 **return** y;  
 }  
 };  
  
**public**:  
 **class** invalid\_key\_exception : **public** std::exception {  
  
 };  
  
**private**:  
 rb\_tree tree;  
 list<K> key\_list;  
 list<V> value\_list;  
  
**public**:  
 **typedef typename** rb\_map<K, V>::rb\_tree::rb\_node node\_t;  
  
 V& **operator**[] (K **const**& key) { // insert  
 node\_t\* found = tree.get\_node(key);  
 **if** (found != **nullptr**) {  
 **return** \*(found->value\_iterator);  
 } **else** {  
 node\_t\* node = **new** node\_t(key\_list.add(key), value\_list.add(V()));  
 tree.insert(node);  
 **return** \*(node->value\_iterator);  
 }  
 }  
  
 V **const**& **operator**[] (K **const**& key) **const** { // access  
 node\_t\* node = tree.get\_node(key);  
 **if** (node != **nullptr**) {  
 **return** \*(node->value\_iterator);  
 }  
 **throw** invalid\_key\_exception();  
 }  
  
 **bool** remove(K key) {  
 node\_t\* node = tree.get\_node(key);  
 **if** (node != **nullptr**) {  
 node = tree.remove(node);  
 key\_list.erase(node->key\_iterator);  
 value\_list.erase(node->value\_iterator);  
 node->right = node->left = node->parent = **nullptr**;  
 **delete**(node);  
 **return true**;  
 }  
 **return false**;  
 }  
  
 node\_t\* find(K key) {  
 **return** tree.get\_node(key);  
 }  
  
 **bool** has(K key) {  
 **return** find(key) != **nullptr**;  
 }  
  
 **void** print() {  
 std::cout << "{";  
 **if** (tree.root != **nullptr**) {  
 tree.root->print();  
 }  
 std::cout << "}\n";  
 }  
  
 **void** show\_tree() {  
 std::cout << "rb\_map tree:\n";  
 tree.show\_tree();  
 std::cout << "\n";  
 }  
  
 list<K>& keys() {  
 **return** key\_list;  
 }  
  
 list<V>& values() {  
 **return** value\_list;  
 }  
  
 **int** length() {  
 **return** value\_list.get\_length();  
 }  
  
 **int** tree\_size() {  
 **return** tree.get\_size();  
 }  
  
 **void** clear() {  
 tree.clear();  
 key\_list.clear();  
 value\_list.clear();  
 }  
};  
  
#endif