**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе №5**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: «**ПОДДЕРЖКА ПРОИЗВОЛЬНОЙ ПОСЛЕДОВАТЕЛЬНОСТИ В СТРУКТУРЕ ДАННЫХ ДЛЯ МНОЖЕСТВ»**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 7307 | Торопов В.А. |  |
| Преподаватель | Колинько П.Г. |  |

Санкт-Петербург

2019

Оглавление

[Цель работы 3](#_Toc10459618)

[Задание 3](#_Toc10459619)

[Выбор структуры данных 3](#_Toc10459620)

[Вывод. 4](#_Toc10459621)

[Результаты работы программы 5](#_Toc10459622)

[Код программы 6](#_Toc10459623)

# Цель работы

Научиться работать со структурами данных для множеств.

# Задание

Дополнить программу, составленную по теме «Хеш-таблицы операциями над последовательностями: MERGE, CONCAT, SUBST.

Формула для вычислений: A \ (B ∩ C) ∪ D ⊕ E

средняя мощность множества: 10.

# Выбор структуры данных

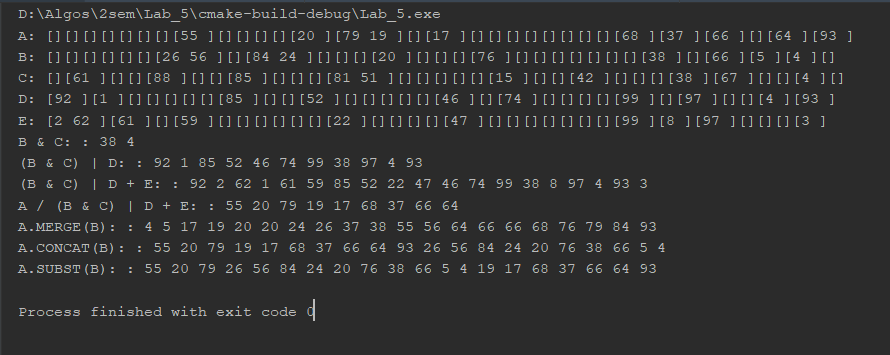
# 

Я выбрал vector указателей на ключи, так как этот способ работы с множеством очень удобен, а недостаток покрывается тем, что мы знаем количество ключей заранее.

# Вывод.

Выбор структуры данных оказался удачным, так как время работы функций с множествами с данной структурой данных составляет О(n). Также с данной структурой данных было удобно работать, что также является плюсом при её выборе.

# Результаты работы программы



# 

# Код программы

Main.cpp

#include <iostream>  
#include "Hash\_Table.h"  
  
**int** main(){  
 srand(time(0));  
 Hash A, B, C, D, E, R, B\_and\_C;  
 A.create('A');  
 B.create('B');  
 C.create('C');  
 D.create('D');  
 E.create('E');  
 R.create('R');  
 A.Out();  
 B.Out();  
 C.Out();  
 D.Out();  
 E.Out();  
 R = B & C;  
 R.Out("B & C: ");  
 R = R | D;  
 R.Out("(B & C) | D: ");  
 R = R + E;  
 R.Out("(B & C) | D + E: ");  
 R = A / R;  
 R.Out("A / (B & C) | D + E: ");  
 R = A.MERGE(B);  
 R.Out\_table\_id("A.MERGE(B): ");  
 R = A.CONCAT(B);  
 R.Out\_table\_id("A.CONCAT(B): ");  
 R = A.SUBST(B, 3);  
 R.Out\_table\_id("A.SUBST(B): ");  
 **return** 0;  
}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Hash\_Table.h

#ifndef LAB\_3\_HASH\_TABLE\_H  
#define LAB\_3\_HASH\_TABLE\_H  
#endif //LAB\_3\_HASH\_TABLE\_H  
#include <set>  
#include <random>  
#include <time.h>  
#include <string.h>  
#include <vector>  
#include <list>  
#include <unordered\_set>  
#include <algorithm>  
  
**using** std::sort;  
  
**enum**{*N* = 10, *max\_num* = 100, *n\_table* = 30};  
**using namespace** std;  
  
**class** Hash{  
 **char** Name;  
 //unordered\_set<int, int> table;  
 vector<list<**int**>> table;  
 vector<**int**> keys;  
 vector<list<**int**>::iterator> table\_id;  
**public**:  
 **void** create(**char** Name\_set);  
 **bool** proof(**int** a);  
 **void** Out();  
 **void** Out(string \_name);  
 **void** Out\_table\_id(string \_name);  
 **void** sort\_id();  
 **int** hash\_table(**int** a);  
 Hash MERGE(Hash );  
 Hash CONCAT(Hash );  
 Hash SUBST(Hash , **int** k);  
 Hash **operator**& (Hash );  
 Hash **operator** &= (Hash );  
 Hash **operator** | (Hash );  
 Hash **operator** |= (Hash );  
 Hash **operator** + (Hash );  
 Hash **operator** += (Hash );  
 Hash **operator** / (Hash );  
 Hash **operator** /= (Hash );  
};  
  
**void** Hash::create(**char** Name\_set){  
 (\***this**).Name = Name\_set;  
 (\***this**).table.resize(*n\_table*);  
 **for** (**int** i = 0; i < *N*; i++){  
 **int** a = rand() % *max\_num*;  
 **while**(!proof(a)){  
 a = rand() % *max\_num*;  
 }  
 }  
 sort(keys.begin(), keys.**end**());  
 **int** k = 0;  
 **for**(**auto** i: (\***this**).keys){  
 **for**(**auto** j: (\***this**).table[i]){  
 **for**(**int** j1 = 0; j1 < table\_id.size(); j1++){  
 **if**(j == \*table\_id[j1]){  
 swap((\***this**).table\_id[k],(\***this**).table\_id[j1]);  
 k++;  
 }  
 }  
 }  
 }  
};  
  
**bool** Hash::proof(**int** a){  
 **int** func = hash\_table(a);  
 **if**(table[func].empty()){  
 (\***this**).keys.push\_back(func);  
 table[func].push\_back(a);  
 (\***this**).table\_id.push\_back(--(\***this**).table[func].**end**());  
 **return true**;  
 } **else**{  
 **bool** flag = **false**;  
 **for**(**auto** i: table[func]){  
 **if**(i == a) flag = **true**;  
 }  
 **if** (!flag){  
 table[func].push\_back(a);  
 (\***this**).table\_id.push\_back(--(\***this**).table[func].**end**());  
 **return true**;  
 } **else return false**;  
 }  
}  
  
**int** Hash::hash\_table(**int** a) {  
 **return** (29\*a + 2) % (3\**N*);  
}  
  
**void** Hash::sort\_id() {  
 **for**(**int** i = 0; i < (\***this**).table\_id.size() - 1; i++){  
 **for**(**int** j = i + 1; j < (\***this**).table\_id.size(); j++){  
 **if**(\*(\***this**).table\_id[i] > \*(\***this**).table\_id[j]){  
 swap((\***this**).table\_id[i], (\***this**).table\_id[j]);  
 }  
 }  
 }  
}  
  
**void** Hash::Out(){  
 cout << Name << ": ";  
 **for** (**int** i = 0; i < 3\**N*; i++){  
 cout << "[";  
 **for**(**auto** j:table[i]){  
 cout << j << " ";  
 }  
 cout << "]";  
 }  
 cout << endl;  
}  
  
**void** Hash::Out(string \_name) {  
 cout << \_name << ": ";  
 **for** (**int** i = 0; i < 3\**N*; i++){  
 //cout << "[";  
 **for**(**auto** j:table[i]){  
 cout << j << " ";  
 }  
 //cout << "]";  
 }  
 cout << endl;  
}  
  
**void** Hash::Out\_table\_id(string \_name) {  
 cout << \_name << ": ";  
 **for**(**auto** i: (\***this**).table\_id){  
 cout << \*i << ' ';  
 }  
 cout << endl;  
}  
  
Hash Hash::**operator**&(Hash B) {  
 Hash R(\***this**);  
 **return** R &= B;  
}  
  
Hash Hash::**operator**&=(Hash B){  
 **for**(**auto** i: keys){  
 **for**(**auto** j: (\***this**).table[i]){  
 **bool** flag = **false**;  
 **if**(B.table[i].empty()){  
 (\***this**).table[i].erase(table[i].begin(), table[i].**end**());  
 flag = **true**;  
 } **else**{  
 **for**(**auto** j1: B.table[i]){  
 **if**(j == j1) flag = **true**;  
 }  
 }  
 **if**(!flag){  
 (\***this**).table[i].remove(j);  
 }  
 }  
 }  
 **return** (\***this**);  
}  
  
Hash Hash::**operator**|(Hash B) {  
 Hash R(\***this**);  
 **return** R |= B;  
}  
  
Hash Hash::**operator**|=(Hash B) {  
 **for**(**auto** i: B.keys){  
 **for**(**auto** j: B.table[i]){  
 **bool** flag = **false**;  
 **if**((\***this**).table[i].empty()){  
 (\***this**).table[i].assign(B.table[i].begin(),B.table[i].**end**());  
 flag = **true**;  
 } **else**{  
 **for**(**auto** j1: (\***this**).table[i]){  
 **if**(j == j1) flag = **true**;  
 }  
 }  
 **if**(!flag){  
 (\***this**).table[i].push\_back(j);  
 }  
 }  
 }  
 **return** (\***this**);  
}  
  
Hash Hash::**operator**/(Hash B) {  
 Hash R(\***this**);  
 **return** R /= B;  
}  
  
  
Hash Hash::**operator**/=(Hash B) {  
 **for**(**auto** i: keys){  
 **for**(**auto** j: (\***this**).table[i]){  
 **bool** flag = **false**;  
 **for**(**auto** j1: B.table[i]){  
 **if**(j == j1) flag = **true**;  
 }  
 **if**(flag){  
 (\***this**).table[i].remove(j);  
 }  
 }  
 }  
 **return** (\***this**);  
}  
  
Hash Hash::**operator**+(Hash B) {  
 Hash R(\***this**);  
 **return** R += B;  
}  
  
Hash Hash::**operator**+=(Hash B) {  
 Hash A = (\***this**) | B;  
 Hash C = (\***this**) & B;  
 **return** A / C;  
}  
  
Hash Hash::MERGE(Hash B) {  
 Hash R(\***this**);  
 R.table.resize(2\*R.table.size());  
 **for**(**int** i = *n\_table*; i < 2 \* *n\_table*; i++){  
 R.table[i].assign(B.table[i - *n\_table*].begin(),B.table[i - *n\_table*].**end**());  
 }  
 R.table\_id.resize(2\*R.table\_id.size());  
 **for**(**int** i = *N*; i < 2\**N*; i++){  
 R.table\_id[i] = B.table\_id[i - *N*];  
 }  
 R.sort\_id();  
 **return** R;  
}  
  
Hash Hash::CONCAT(Hash B) {  
 Hash R(\***this**);  
 R.table.resize(2\*R.table.size());  
 **for**(**int** i = *n\_table*; i < 2 \* *n\_table*; i++){  
 R.table[i].assign(B.table[i - *n\_table*].begin(),B.table[i - *n\_table*].**end**());  
 }  
 R.table\_id.resize(2\*R.table\_id.size());  
 **for**(**int** i = *N*; i < 2\**N*; i++){  
 R.table\_id[i] = B.table\_id[i - *N*];  
 }  
 **return** R;  
}  
  
Hash Hash::SUBST(Hash B, **int** k) {  
 Hash R(\***this**);  
 R.table.resize(2\*R.table.size());  
 **for**(**int** i = *n\_table*; i < 2 \* *n\_table*; i++){  
 R.table[i].assign(B.table[i - *n\_table*].begin(),B.table[i - *n\_table*].**end**());  
 }  
 R.table\_id.resize(2\*R.table\_id.size());  
 /\*for(int i = N; i < 2\*N; i++){  
 R.table\_id[i] = B.table\_id[i - N];  
 }\*/  
 **for**(**int** i = k; i < *N*; i++){  
 R.table\_id[i + *N*] = R.table\_id[i];  
 }  
 **for**(**int** i = 0; i < *N*; i++){  
 R.table\_id[i + k] = B.table\_id[i];  
 }  
 **return** R;  
}