**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра Вычислительной техники**

отчет

**по лабораторной работе №6**

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

Тема: «**ИСПОЛЬЗОВАНИЕ СТАНДАРТНОЙ БИБЛИОТЕКИ ШАБЛОНОВ»**

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

Санкт-Петербург

2019

Оглавление

[Цель работы 3](#_Toc10460472)

[Задание 3](#_Toc10460473)

[Выбор структуры данных и функций STL. 3](#_Toc10460474)

[Вывод. 4](#_Toc10460475)

[Результаты работы программы 5](#_Toc10460476)

[Код программы 6](#_Toc10460477)

# Цель работы

Научиться работать со стандартной библиотекой шаблонов.

# Задание

Переделать программу, составленную при выполнении темы 5 «Последовательности», под использование контейнеров из стандартной библиотеки шаблонов. Для хранения множеств выбрать контейнер подходящего типа (*set* или *unordered\_set* и т. п.) и доработать его для поддержки операций с последовательностями. Для реализации операций с контейнерами использовать возможности библиотеки алгоритмов.

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

средняя мощность множества: 10.

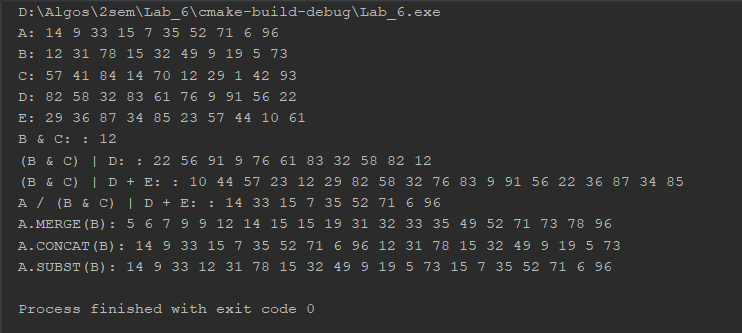
# Выбор структуры данных и функций STL.

Я выбрал контейнер unordered\_set, так как мне необходим контейнер хеш-таблицы, а также в моей хеш-таблице значения совпадают с ключами. Также в программе использовался vector. В работе использовались функции insert(), erase(), push\_back(), find(), а также sort().

# Вывод.

Выбор структуры данных оказался удачным, так как время работы с данными структурами 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\_id("A.MERGE(B): ");  
 R = A.CONCAT(B);  
 R.Out\_id("A.CONCAT(B): ");  
 R = A.SUBST(B, 3);  
 R.Out\_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**> table;  
 vector<**int**> keys;  
**public**:  
 **void** create(**char** Name\_set);  
 **void** Out();  
 **void** Out(string \_name);  
 **void** Out\_id(string \_name);  
 Hash MERGE(Hash );  
 Hash CONCAT(Hash );  
 Hash SUBST(Hash , **int** k);  
 Hash **operator**& (Hash );  
 Hash **operator** | (Hash );  
 Hash **operator** + (Hash );  
 Hash **operator** / (Hash );  
};  
  
**void** Hash::create(**char** Name\_set){  
 (\***this**).Name = Name\_set;  
 **for** (**int** i = 0; i < *N*; i++){  
 **int** a = rand() % *max\_num*;  
 **while**(table.find(a) != table.**end**()){  
 a = rand() % *max\_num*;  
 }  
 table.insert(a);  
 }  
 **for**(**auto** i: table){  
 keys.push\_back(i);  
 }  
};  
  
**void** Hash::Out(){  
 cout << Name << ": ";  
 **for**(**auto** i: table){  
 cout << i << ' ';  
 }  
 cout << endl;  
}  
  
**void** Hash::Out(string \_name) {  
 cout << \_name << ": ";  
 **for**(**auto** i: table){  
 cout << i << ' ';  
 }  
 cout << endl;  
}  
  
**void** Hash::Out\_id(string \_name) {  
 cout << \_name;  
 **for**(**auto** i: (\***this**).keys){  
 cout << i << ' ';  
 }  
 cout << endl;  
}  
  
Hash Hash::**operator**&(Hash B) {  
 Hash R(\***this**);  
 **for** (**auto** i: R.table) {  
 **if**(B.table.find(i) == B.table.**end**()){  
 R.table.erase(i);  
 }  
 }  
 R.keys.clear();  
 **for**(**auto** i: R.table){  
 R.keys.push\_back(i);  
 }  
 **return** R;  
}  
  
  
Hash Hash::**operator**|(Hash B) {  
 Hash R(\***this**);  
 **for**(**auto** i: B.table){  
 **if**(R.table.find(i) == R.table.**end**()){  
 R.table.insert(i);  
 }  
 }  
 R.keys.clear();  
 **for**(**auto** i: R.table){  
 R.keys.push\_back(i);  
 }  
 **return** R;  
}  
  
  
Hash Hash::**operator**/(Hash B) {  
 Hash R(\***this**);  
 **for** (**auto** i: R.table) {  
 **if**(B.table.find(i) != B.table.**end**()){  
 R.table.erase(i);  
 }  
 }  
 R.keys.clear();  
 **for**(**auto** i: R.table){  
 R.keys.push\_back(i);  
 }  
 **return** R;  
}  
  
Hash Hash::**operator**+(Hash B) {  
 Hash R = (\***this**) | B;  
 Hash A = (\***this**) & B;  
 R = R / A;  
 R.keys.clear();  
 **for**(**auto** i: R.table){  
 R.keys.push\_back(i);  
 }  
 **return** R;  
}  
  
Hash Hash::MERGE(Hash B) {  
 Hash R(\***this**);  
 R.table.insert(B.table.begin(), B.table.**end**());  
 R.keys.insert(R.keys.**end**(), B.keys.begin(), B.keys.**end**());  
 sort(R.keys.begin(), R.keys.**end**());  
 **return** R;  
}  
  
Hash Hash::CONCAT(Hash B) {  
 Hash R(\***this**);  
 R.table.insert(B.table.begin(), B.table.**end**());  
 R.keys.insert(R.keys.**end**(), B.keys.begin(), B.keys.**end**());  
 **return** R;  
}  
  
Hash Hash::SUBST(Hash B, **int** k) {  
 Hash R(\***this**);  
 R.table.insert(B.table.begin(), B.table.**end**());  
 R.keys.insert(R.keys.begin() + k, B.keys.begin(), B.keys.**end**());  
 **return** R;  
}