

Съдържание

Standart Template Library	2
Контейнери	2
Клас "pair"	2
Resizable Array a.k.a. Vector: C++ STL <vector> (Java ArrayList).....	3
Клас "string"	7
Клас "bitset"	8
Linked List: C++ STL <list> (Java LinkedList)	9
Stack: C++ STL <stack> (Java Stack).....	10
Queue: C++ STL <queue> (Java Queue)	12
Double ended queue: STL < deque >	14
Heap: C++ STL <queue>: priority_queue (Java PriorityQueue)	15
Итератори	19
Класове "set" и "multiset"	20
Класове "map" и "multimap"	22
Алгоритми	24
Сортиране	24
Промяна на последователности	26
Търсене в не сортирани последователности	27
Търсене в сортирани последователности	29
Генериране на пермутации	31
Някои често ползвани функции	31
Използвана литература	32

Standart Template Library

STL е съкращение от Standart Template Library. Той е част от стандартната C++ библиотека и множество алгоритми, структури от данни и някои инструменти, които значително могат да ускорят писането на състезателни задачи.

Контейнери

За "структури данни" в STL се ползва името контейнери. Това име идва поради специфичността на структурите данни, които са имплементирани в STL - това са само такива структури данни, които съдържат елементи (подредени по някакъв начин) и дават достъп до тях.

Клас "pair"

Съхранява двойка обекти от произволен тип. Често в различни задачи се налага да дефинираме собствени структури. Като например

```
struct point { double x, y; };
struct edge { int from, to, cost; };
struct product { string name; int amount; };
```

В случаите, в които структурата ни ще се състои от 2 полета, можем да избегнем дефинирането ѝ като използваме **pair**. Ето как би изглеждало това:

```
pair <double, double> point;
pair <string, int> product;
```

Разбира се може да използваме **pair** и когато структурата се състои от повече от 2 полета, но това може да усложни по-нататъшния ни код. Ето как би изглеждала структурата edge:

```
pair< pair<int, int>, int> edge;
```

Класът **pair** предоставя две член променливи с публичен достъп, съответно именувани **first** и **second**. Пример за използване на **pair** е:

```
pair <string, int> product;
product.first = "alabala";
product.second = 54;
printf("%s %d\n", product.first.c_str(), product.second);
```

Удобен начин за конструиране на **pair** е функцията **make_pair**. Използвайки я, горния код би изглеждал така:

```
pair <string, int> product = make_pair("alabala", 54);
printf("%s %d\n", product.first.c_str(), product.second);
```

Важно е да се отбележи, че в **pair** са предефинирани операторите **<**, **==**, **>**, както и операторът за присвояване **=**. Сравнението се извършва първо по полето **first** и ако стойностите в него са равни - по полето **second**. Т.е. ако имаме:

```
pair <string,int> a = make_pair("b", 3);
pair <string,int> b = make_pair("a", 1);
pair <string,int> c = make_pair("b", 3);
pair <string,int> d = make_pair("a", 2);
```

то изразът $(b < d) \&\& (d < a) \&\& (a == c)$ ще има стойност **true**.

За да можем да използваме **pair** в програмата си трябва да включим библиотеката, в която е дефиниран, както и именуваното пространство **std**:

```
#include <algorithm>
using namespace std;
```

Resizable Array a.k.a. Vector: C++ STL <vector> (Java ArrayList)

STL-ска имплементация на **динамичен масив**. Един обект **vector** е подобен на масив по това, че осигурява произволен достъп до елементите поставени в поредица. За разлика от традиционния масив един обект **vector** (по време на работа) може да променя размерите си динамично, така че да поддържа произволен брой елементи. Един обект **vector** може бързо да вмъкне или отстрани елементи от края на неговата последователност, но вмъкването или отстраняването на друга позиция не е толкова ефикасно. Това е така, защото обектът **vector** трябва да премести позициите на елементите, за да настани новия елемент или да затвори мястото, оставено от отстраненият елемент. Достъпът до елементите на вектор се осъществява чрез итератори. Дефиницията на шаблона за клас **vector** се съдържа във файла "vector" (#include <vector>).

Конструиране на **vector**:

```
vector<int> first; // empty vector of ints
vector<int> second (4); // four ints
vector<int> second (4,100); // four ints with value 100
vector<int> third (second.begin(), second.end()); // iterating through second
vector<int> fourth (third); // a copy of third
```

Да разгледаме пример, за конструиране на **vector**, използвайки първия конструктор:

```
vector <int> a;
for (int i = 0; i < n; i++)
{
    scanf("%d", &a[i]);
    a.push_back(tmp);
}
for (int i = 0; i < n; i++)
    printf("%d ", a[i]);
```

Както и с втория конструктор:

```
vector<int> a(n);
for (int i = 0; i < n; i++)
    scanf("%d", &a[i]);
for (int i = 0; i < n; i++)
    printf("%d ", a[i]);
```

По-добре е да се използва втория вариант, понеже още в самото начало ще бъде заделена необходимата памет и няма да има допълнителни заделяния при извикване на **push_back**, както е случая при първия вариант. Това е така, понеже процесът по алокиране и заделяне на динамична памет е забележимо бавен, а тази операция се изпълнява всеки път, когато се променя размера на вектора. От това програмата ни става доста по-бавна.

Въобщие, ако в даден момент знаем точния брой на елементите, които нашият вектор ще трябва да съдържа, добра практика е да използваме конструктор със задаване на този брой или да викаме метода **resize**, ако векторът е вече конструиран.

Шаблонът за клас **vector** дефинира пълно множество от оператори в това число и оператора за сравняване. Една програма може да определи дали два вектора са равни и кой е по-голям или по малък от друг. За равни вектори се смятат 2 вектора с равен брой елементи и еднакви елементи.

Съвет: не ползвайте вектори за щяло и нещяло! Когато можете да ползвате обикновен масив - ползвайте обикновен масив. Понякога програма, която ползва вектори, може да е *няколко пъти* по-бавна от аналогична програма със стандартни масиви. Често без STL бихме ползвали повече памет, но какво от това? Ако програмата се събира в ограниченията по памет, то няма проблем.

```
#include <stdio.h>
#include <vector>
using namespace std;

vector<int> a (10, -1); // 10 ints with value -1
vector<int> b (20); // 20 ints

void print()
{
    for(int i = 0; i < (int)a.size(); i++)
        printf("%d ", a[i]);
    printf("\n");
}

int main()
{
    for(int i = 0; i < 20; i++)
        b[i] = i;

    print(); // -1 -1 -1 -1 -1 -1 -1 -1 -1 -1

    //~~~~ ASSIGN ~~~
    //Assigns new content to the vector object, dropping all the elements
```

```

//contained in the vector before the call and replacing them by those
//specified by the parameters:
a.assign (5, 0); // a repetition 5 times of value 0
print(); // 0 0 0 0 0

int arr[] = {1, 2, 3, 4, 5, 6, 7};
a.assign (arr + 3, arr + 6); // assigning from array.
print(); // 4 5 6

a.assign (b.begin() + 5, b.end() - 10); // assigning from other vector.
print(); // 5 6 7 8 9

//~~~~ BACK ~~~
//A reference to the last element in the vector.
while (a.back() != 0)
{
    a.push_back(a.back() - 1);
}
print(); //5 6 7 8 9 8 7 6 5 4 3 2 1 0

//~~~~ BEGIN ~~~
//Returns an iterator referring to the first element in the vector.
vector<int>::iterator begin = a.begin();
printf("%d\n", *begin); // 5

//~~~~ END ~~~
//Returns an iterator referring to the past-the-end element
for(; begin < a.end(); begin++)
    printf("%d ", *begin); //5 6 7 8 9 8 7 6 5 4 3 2 1 0

//~~~~ CAPACITY ~~~
//Return size of allocated storage capacity
printf("\ncapacity: %d\n", a.capacity());

//~~~~ CLEAR ~~~
//Clear content
//All the elements of the vector are dropped: their destructors are
//called, and then they are removed from the vector container,
//leaving the container with a size of 0.
a.clear();
print(); //

//~~~~ EMPTY ~~~
//Returns whether the vector container is empty, i.e. its size is 0.
printf(a.empty() ? "Empty.\n" : "Not empty\n"); //Empty

for(int i = 0; i < 10; i++)
    a.push_back(i + 1);
print(); // 1 2 3 4 5 6 7 8 9 10

//~~~~ ERASE ~~~
//Removes from the vector container either a single element (position)
or a range of elements ([first,last)).

// erase the 6th element

```

```

a.erase (a.begin() + 5);
print(); // 1 2 3 4 5 7 8 9 10

// erase the first 3 elements:
a.erase (a.begin(), a.begin() + 3);
print(); // 4 5 7 8 9 10

//~~~~ FRONT ~~~
//Returns a reference to the first element in the vector container.
a.front() -= a.back();
printf("%d\n", a.front()); // -6

//~~~~ INSERT ~~~
//Extendin vector by inserting new elements before the element at
position
a.insert(a.begin() + 2, 44);
print(); // -6 5 44 7 8 9 10

//~~~~ MAX SIZE ~~~
//Returns the maximum number of elements that the vector container can
hold
printf("%d\n", a.max_size()); //1073741823

//~~~~ POP_BACK ~~~
//Removes the last element in the vector, effectively reducing the
//vector size by one and invalidating all iterators and references to
it.
a.pop_back();
a.pop_back();
print(); // -6 5 44 7 8

//~~~~ PUSH_BACK ~~~
//Adds a new element at the end of the vector, after its current last
//element. The content of this new element is initialized to a copy of
x.
a.push_back(-3);
print(); // -6 5 44 7 8 -3

return 0;
}

```

Нека имаме граф с $N \leq 100000$ върха и $M \leq 500000$ **насочени** ребра, като всяко ребро има определена цена (или в общия случай наричано "тегло") – число с плаваща запетая. Добро представяне на този граф е списък на съседите. Следният програмен фрагмент демонстрира как можем да имплементираме представяне на графа чрез списък на съседите посредством употребата на **vector**.

```

vector < vector < pair<int, float> > > Graph(N);
for (int i = 0; i < M; i++)
{
    int from, to;
    float cost;
    cin>>from>>to>>cost;
    Graph[from].push_back(make_pair(to, cost));
}

```

```
}
```

Клас "string"

Класът **string**, дефиниран в библиотеката <string>, е един от най-силните елементи на STL. Той предоставя удобен начин за работа със символни низове, като ни спестява много усилия. По същество е **vector**, където типът е предварително зададен (**char**). С нея могат да се представят стрингове. Логично. Но като цяло има няколко особени предимства пред нормалния `char*`. Първо, размерът му може да бъде взет с константна сложност чрез метода `.size()`. Второ, не се нуждае от терминираща нула. Трето, позволява лесно разширяване или смаляване (както при вектор). Четвърто, копирането му е много по-лесно. Пето, предефинирани са няколко допълнителни оператора - например `operator <`, `operator +`, `operator +=`. С тях може да се сравняват и съединяват стрингове сравнително по-лесно, отколкото при `char*`.

Недостатъците са, че отново, както и при векторите, е малко по-бавен от нормален масив от `char`-ове. Има допълнителен метод `.c_str()`, който връща `const` указател към стринга, за да може да бъде печатан като обикновен C-style стринг.

Започваме с това, че в него са предефинирани операторите `=`, `+=` и `[]`. Благодарение на това следният код е абсолютно коректен:

```
string a = "alabala";
cout<<a<<endl;
a += "_suffix";
cout<<a<<endl;
cout<<a.size()<<endl;
cout<<a[3]<<endl;
```

Важно е да се отбележи, че символите в **string** са индексирани от 0! Можем да сравняваме два стринга с `<`, `==`, `>`, като казваме че `a < b`, ако `a` е лексикографски преди `b`. В следния пример са показани други полезни член функции на класа **string**.

```
string a = "alabala!!!";
printf("%d\n", a.empty());
printf("%d\n", a.length()); // дължината на a

a.erase(4, 3); // трие три символа от a, започвайки от позиция 3
printf("%s\n", a.c_str());

a.erase(5); // трие всички символи в a, от позиция 5 нататък
printf("%s\n", a.c_str());
cout<<a.find("lab")<<endl; // връща индекса на първото срещане на "lab" в a
// Ако няма такова - връща
константата string::npos
a.insert(4, "ala"); // вкарва низа "ala" в a на позиция 4
```

```
printf("%s\n", a.c_str());
a.replace(3, 2, "ffff"); // заменя 2 символа в a, започващи от позиция 3, с
низа "ffff"

printf("%s\n", a.c_str());
printf("%s\n", a.substr(1, 3).c_str());

getline(cin, a); // четете ред от cin
printf("%s\n", a.c_str());
cin>>a; // четете дума от cin

printf("%s\n", a.c_str());
```

Клас "bitset"

Класа **bitset** предоставя много ефикасен начин за съхранение на битове (1, 0, true или false).

```
#include <iostream>
#include <string>
#include <bitset>
using namespace std;

int main ()
{
    bitset<4> first (string("1001")); // initialize from string
    bitset<4> second (string("0011")); // initialize from string

    cout << (first^=second) << endl; // 1010 (XOR,assign)
    cout << (first&=second) << endl; // 0010 (AND,assign)
    cout << (first|=second) << endl; // 0011 (OR,assign)

    cout << (first<<=2) << endl; // 1100 (SHL,assign)
    cout << (first>>=1) << endl; // 0110 (SHR,assign)

    cout << (~second) << endl; // 1100 (NOT)
    cout << (second<<1) << endl; // 0110 (SHL)
    cout << (second>>1) << endl; // 0001 (SHR)

    cout << (first==second) << endl; // false (0110==0011)
    cout << (first!=second) << endl; // true (0110!=0011)

    cout << (first&second) << endl; // 0010
    cout << (first|second) << endl; // 0111
    cout << (first^second) << endl; // 0101

    cout<< first.to_ulong() << endl; // 6
    cout<< second.to_ulong() << endl; // 3

    bitset<10> third (155); // initialize from long
    cout<< third.to_string() << endl; // 0010011011

    cout<< third.count() << endl; // number of 1
    cout<< third.size() - third.count() << endl; // number of 0
```



```

    // flips only the bit at position pos
    cout << third.flip(2) << endl;      // 0010011111

    //changes all 0s for 1s and all 1s for 0s.
    cout << third.flip() << endl;      // 1101100000

    //convert to unsigned long integer
    cout << third.to_ulong() << endl;

    //resets all the bits in the bitset (sets al bits to 0).
    third.reset();
    cout << third.to_string() << endl; // 0000000000

    //sets (to 1) all the bits in the bitset.
    cout << third.set() << endl;      // 1111111111

    //The parameterized version, stores val as the
    //new value for bit at position pos.
    cout << third.set(2,0) << endl;    // 1111111011
    cout << third.set(2) << endl;      // 1111111111

    return 0;
}

```

Linked List: C++ STL <list> (Java LinkedList)

Един обект **list** е подобен на вектор или дек, с изключение на това че списъците не осигуряват произволен достъп. Все пак един обект **list** е ефикасен при поставянето на елементи във, или отстраняването на елементи от произволно място в последователност. Подобно на вектор или дек един обект **list** може да променя размерите си динамично при необходимост. Достъп до елементите може да се осъществи също и чрез итератори.

```

#include <stdio.h>
#include <list>
using namespace std;

void print(list<int> l)
{
    for (list<int>::iterator it = l.begin(); it != l.end(); it++)
        printf("%d ", *it);
    printf("\n");
}

int main ()
{
    // constructors used in the same order as described above:
    list<int> first;                      // empty list of ints
    list<int> second (4,100);             // four ints with value 100
    list<int> third (second.begin(),second.end()); // iterating through second
    list<int> fourth (third);             // a copy of third

    // the iterator constructor can also be used to construct from arrays:

```

```

int myints[] = {16, 2, 77, 29};
list<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
print(fifth); //16 2 77 29

fifth.push_front(1);
fifth.push_back(100);
print(fifth); //1 16 2 77 29 100

list<int>::iterator it = fifth.begin(); // points to first element
it++; // points to second element

fifth.insert(it, 33);
print(fifth); //1 33 16 2 77 29 100

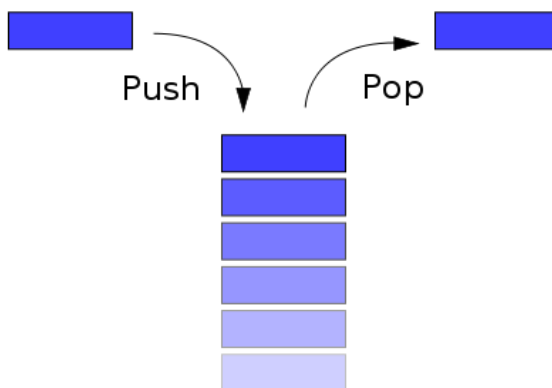
it++; // points to third element
fifth.insert(it, 2, 20);
print(fifth); //1 33 16 20 20 2 77 29 100

return 0;
}

```

Stack: C++ STL <stack> (Java Stack)

Стекът е последователност, която изпълнява операция тип "първи влязъл, последен излязъл" (LIFO), върху елементите си.



Класът **stack**, дефиниран в библиотеката **<stack>**. Както знаем, стекът е структура от данни, в която можем само да модифицираме елемента, стоящ на върха, и никой друг. Затова логично можем да предположим, че в **stack** нямаме предефиниран оператор [], понеже не можем да достъпваме произволни елементи.

Член функциите, които ще ни се наложи да употребяваме най често, са:

- **push** - вкарва елемент на върха на стека
- **pop** - премахва най-горния елемент от стека (ако има такъв)
- **top** - взема най-горния елемент от стека

Важно е да разберем, че макар че **vector** предоставя цялата функционалност на **stack**, когато наистина се нуждаем от структурата от данни стек, е по-добре да използваме класа **stack**. Това е така, първо защото **stack** работи около 2 пъти по-бързо от **vector**, понеже не заделя големи блокове памет наведнъж, и второ защото ограничавайки функционалността само до тази, която ни е необходима, се улеснява използването и се намаля риска да допуснем имплементационна грешка.

Пример за използването на **stack** в реална задача е следния:

Нека искаме да направим обхождане в дълбочина на графа. Ако реализираме обхождането чрез рекурсия, рискуваме при определен тип граф, рекурсията да стане твърде дълбока и да се получи препълване на стека. Затова можем да реализираме обхождането итеративно, като използваме класа **stack**.

(Между впрочем, трябва да се отбележи, че всяка една рекурсивна функция може да бъде реализирана итеративно!)

За решаването на задачата на върха на стека ще пазим двойката: (върх от графа, в който се намираме ; номер на следващия съсед на този върх, който трябва да посетим). Ще започнем обхождането на графа от върха с номер 0.

```
#include "stdio.h"
#include <stack>
#include <vector>
using namespace std;

#define N 1000000

vector < vector < pair <int, float> > > Graph(N);

void dfs()
{
    vector<bool> viz(N);
    viz[0] = true;
    stack < pair <int, int> > dfs;
    dfs.push(make_pair(0, 0));

    while (!dfs.empty())
    {
        const pair <int, int> top = dfs.top();
        dfs.pop();
        const int vertex = top.first;
        const int next = top.second;
        if (!next)
            printf("Coming into vertex %d\n", vertex);

        if (next < (int)Graph[vertex].size())
        {
            dfs.push(make_pair(vertex, next + 1));
            const int toVisit = Graph[vertex][next].first;
            if (!viz[ toVisit ])
```

```

        {
            viz [ toVisit ] = true;
            dfs.push(make_pair(toVisit,0));
        }
    }
}

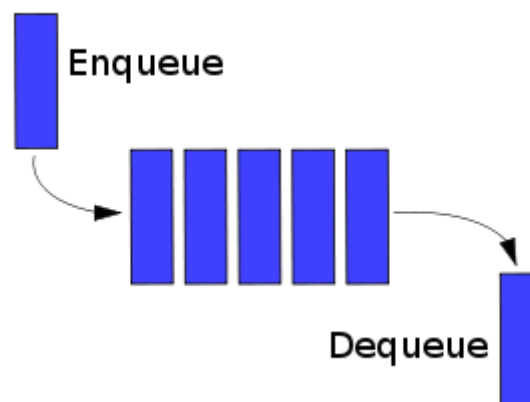
void read()
{
    int from, to;
    float cost;
    for (int i = 0; i < M; i++)
    {
        scanf("%d %d %f\n", &from, &to, &cost);
        Graph[from].push_back(make_pair(to, cost));
    }
}

int main()
{
    read();
    dfs();
    return 0;
}

```

Queue: C++ STL <queue> (Java Queue)

Опашката е структура от данни която реализира операции тип "първи влязъл, първи излязъл" (FIFO) върху елементите си. Тоест елементите в опашката се вмъкват от единия край и излизат от другия.



Класа **queue** дефиниран в библиотеката **<queue>**. Ще споменем член функциите, които ще ни се наложи да използваме най-често. Това са

- **push** - слага елемент на края на опашката

- **pop** - премахва първия елемент от опашката
- **front** - взема първия елемент от опашката

Следният пример илюстрира използването на **queue** за обхождане в широчина на нашия граф:

```
#include "stdio.h"
#include <queue>
#include <vector>
using namespace std;

#define N 1000000

vector < vector < pair<int, float> > > Graph(N);

void bfs()
{
    vector<bool> viz(N);
    viz[0] = true;
    queue <int> bfs;
    bfs.push(0);
    while (!bfs.empty())
    {
        const int vertex = bfs.front();
        printf("Coming into vertex %d\n", vertex);
        bfs.pop();
        for (int i = 0; i < (int)Graph[vertex].size(); i++)
        {
            const int toVisit = Graph[vertex][i].first;
            if (!viz[toVisit])
            {
                viz[toVisit] = true;
                bfs.push(toVisit);
            }
        }
    }
}

void read()
{
    int from, to;
    float cost;
    for (int i = 0; i < M; i++)
    {
        scanf("%d %d %f\n", &from, &to, &cost);
        Graph[from].push_back(make_pair(to, cost));
    }
}

int main()
{
    read();
    bfs();
    return 0;
}
```

Double ended queue: STL <deque>

Друг контейнер, който ще споменем, е класа **deque** дефиниран в библиотеката <deque>. Дека е структура от данни подобна на опашката, с тази разлика че можем за константно време да добавяме и премахваме елементи и в двата ѝ края. Най-често използваните функции са:

- **push_back** - слага елемент на края на дека
- **push_front** - слага елемент в началото на дека
- **pop_back** - премахва последния елемент от дека
- **pop_front** - премахва първия елемент от дека
- **front** - взема първия елемент от дека
- **back** - взема последния елемент от дека

```
#include <deque>
#include <stdio.h>
using namespace std;

void print(deque<int> &d)
{
    for (int i = 0; i < d.size(); i++)
        printf("%d ", d[i]);
    printf("\n");
}

int main()
{
    deque<int> first; // empty deque of ints
    deque<int> second (4,100); // four ints with value 100
    deque<int> third (second.begin(), second.end()); // iterating through
second
    deque<int> fourth (third); // a copy of third

    // the iterator constructor can also be used to construct from arrays:
    int myints[] = {16, 2, 77, 29};
    deque<int> fifth (myints, myints + sizeof(myints) / sizeof(int) );
    print(fifth); //16 2 77 29

    fifth.push_front(1);
    fifth.push_back(100);

    print(fifth); //1 16 2 77 29 100

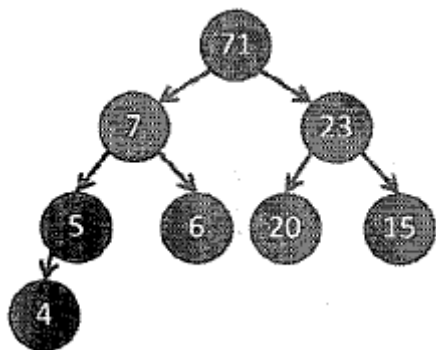
    fifth.front() += fifth.back();
    print(fifth); //101 16 2 77 29 100
    return 0;
}
```

Heap: C++ STL <queue>: priority_queue (Java PriorityQueue)

Много полезна структура данни. В нея можете да добавяте елементи със сложност $O(\log(N))$ и да взимате максималния елемент с $O(1)$. Също така можете да премахвате максималния елемент със сложност $O(\log(N))$. Често можете да я срещнете и като *heap* в английската литература.

Опашка с приоритет е структура за данни, която извлича елементи от последователност според приоритета им. Приоритетът е основан на поставяната функция за сравнение (наречена "предикат"). Например, ако използвате предварително дефинирания предикат `std::less<>`, винаги когато добавяте или отстранявате стойност от опашка с приоритет, контейнерите се подреждат в низходящ ред. Това задава на елемента с най-голяма стойност най-висок приоритет.

Контейнерът **priority_queue** е дефиниран в библиотеката **<queue>** (също както класът **queue**) и предоставя възможности за добавяне на елемент и за вземане и изтриване на "най-горния" (с най-голяма стойност) елемент. По същество **priority_queue** е контейнер-адаптер – т.е. имплементацията е реализирана върху някакъв друг контейнер – по подразбиране това е **vector**, но може и да е някой друг. Още по-точно имплементацията е "random access container" поддържан като пирамида. Това гарантира сложност $\Theta(\log N)$ при добавяне и премахване на елемент.



Имаме голям брой конкуриращи се обекти, всеки от който има дадена важност (приоритет, с който трябва да го разгледаме). В процеса на работа могат да се появяват нови обекти със свой собствен приоритет (потенциално много нисък или много висок), които трябва да бъдат обработени. От нас се иска бързо да можем да:

- Добавяме нов обект с даден приоритет;
- Намираме обекта с най-голям приоритет;
- Премахваме обекта с най-голям приоритет.

Пример за използване на **priority_queue** (числата се извеждат в низходящ ред):

```
priority_queue<int> a;  
a.push(3);  
a.push(5);  
a.push(1);
```

```
while (!a.empty())
{
    printf("%d\n", a.top());
    a.pop();
}
```

Разбира се, можем да сравняваме елементите в **priority_queue** с наша логика. Това може да стане по следния начин:

```
#include "stdio.h"
#include <queue>
#include <vector>
using namespace std;

struct cmp
{
    bool operator() (const int &a, const int &b) const {
        return a > b;
    }
};

int main()
{
    priority_queue<int, vector<int>, cmp> a;
    a.push(3);
    a.push(5);
    a.push(1);
    while (!a.empty())
    {
        printf("%d\n", a.top());
        a.pop();
    }
    return 0;
}
```

В горния пример числата се извеждат във възходящ ред. **cmp** извършва сравнението на елементите като смисъла на връщаната bool стойност от оператора е дали елемента a е по-малък от елемента b.

С **vector <int>** задаваме контейнера, на който ще се базира приоритетната опашка. Като основен недостатък на **priority_queue** можем да споменем липсата на функционалност за промяна на стойностите на елементите.

Т.е. ако в даден момент в приоритетната опашка имаме елементи А, В и С със стойности съответно 1, 3 и 5, не можем да сменим стойността на С от 5 на 2, и това да се отрази в опашката. Едно възможно решение на този проблем е следното:

Когато ни се наложи смяна на стойността на даден елемент, всичко което правим е да добавим този елемент с новата му стойност в опашката. По този начин в опашката можем да получим повече от 1 запис за един и същи елемент. Когато ни се наложи да вземем най-големия елемент

от опашката, първо проверяваме дали стойността му е актуална – т.е. последната добавена в опашката за този елемент. Ако не е, то премахваме този най-голям елемент и вземаме следващия.

Недостатък на горното решение е, че се увеличава необходимата памет и намалява (макар и не много съществено) производителността на приоритетната опашка. Все пак в повечето задачи този недостатък няма да е от голямо значение и можем да си го позволим.

Пример за използване на горното решение е реализирането на алгоритъма на Дейкстра със сложност $\Theta(M \cdot \log N)$.

```
#include <stdio.h>
#include <queue>
using namespace std;

#define MAX 100
#define MP(x, y) make_pair((x), (y))

int E, V, dist[MAX], parent[MAX];
bool vis[MAX];
vector < pair <int,int > > edges[MAX];

void printPath(int s, int j)
{
    if (parent[j] != s)
        printPath(s, parent[j]);
    printf("%d ", j + 1);
}

void printResult(int s)
{
    for (int i = 0; i < V; i++)
    {
        if (dist[i] != INT_MAX && i != s)
        {
            printf("%d -> %d (%d): ", s + 1, i + 1, dist[i]);
            printf("%d ", s + 1);
            printPath(s, i);
            printf("\n");
        }
    }
}

void read()
{
    int u, v, c;
    scanf("%d %d", &V, &E);
    for(int i = 0; i < E; i++)
    {
        scanf("%d %d %d", &u, &v, &c);
        u--; v--;

        edges[u].push_back(MP(v, c));
        //edges[v].push_back(MP(u, c)); //if the graph is b-directional
    }
}
```

```
    }
}

void dijkstra(int s)
{
    int i, u, v, w;
    dist[s] = 0;
    parent[s] = -1;
    priority_queue<pair<int,int> > pq;
    pq.push(MP(-dist[s], s));

    while (!pq.empty())
    {
        u = pq.top().second;
        pq.pop();
        if (vis[u])
            continue;

        vis[u] = true;
        for(i = 0; i < edges[u].size(); i++)
        {
            v = edges[u][i].first;
            w = edges[u][i].second;

            if (dist[u] + w < dist[v])
            {
                dist[v] = dist[u] + w;
                parent[v] = u;
                pq.push(MP(-dist[v], v));
            }
        }
    }
}

void init()
{
    for(int i = 0; i < V; i++)
    {
        dist[i] = INT_MAX;
        vis[i] = false;
    }
}

void main()
{
    int startV = 1;
    read();
    init();
    dijkstra(startV);
    printResult(startV);
}
```

Итератори

За да преминем към важните контейнери **set**, **multiset**, **map** и **multimap**, ще трябва първо да разгледаме какво представляват итераторите и как се използват.

Контейнерите се делят на два вида според това дали можем да обхождаме елементите им. От разгледаните до сега контейнери можем да обхождаме всички елементи единствено на **vector**, а на останалите – не.

Контейнерите, чиито елементи могат да бъдат обходени (**итерирани**), предоставят удобен механизъм за това – **итератори**.

Итераторите в STL представляват **позиции на елементи в различни STL контейнери**. Тъй като итераторите винаги са асоциирани със специфичен тип контейнер, декларирането на итератор става използвайки контейнера, към който те са асоциирани.

Пример:

```
vector<double>::iterator values_iter;  
  
vector<double>::const_iterator const_values_iter;
```

Итераторите се делят на два вида, според това дали елементите могат да бъдат модифицирани чрез тях – константни и не константни (съответно не даващи и даващи право да се модифицират елементите).

Итераторите се делят на три вида според това какви възможности за итериране предлагат:

- **Forward** – възможна е итерация само в една посока, без връщане назад
- **Bidirectional** – възможна е итерация и в двете посоки
- **Random access** – възможна е итерация и в двете посоки със прескачане на елементи.

Следната таблица илюстрира възможностите, които предотставя всеки един от тези три типа:

Оператор	Описание	Forward	Bidirectional	Random access
!=, ==	Сравнение на итератори	да	да	да
++	Итериране 1 позиция напред	да	да	да
--	Итериране 1 позиция назад		да	да
+=, -=, +, -	Итериране на произволен брой			да

Класът **vector**, който разгледахме по-горе, предоставя **Random access** итератори. Той предоставя методи **begin()** и **end()**, които връщат итератори съответно към първия елемент и края на вектора (т.е. позицията след края на вектора).

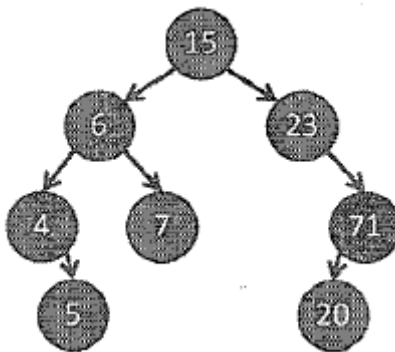
Достъпването на елемента, който "стои зад итератора" става като използваме итератора като указател към този елемент. Пример:

```
vector < pair <int,int> > A(3);
for (vector <pair <int,int> >::iterator it = A.begin(); it != A.end(); it++)
    cout<<it->first<<' '<<it->second<<endl;
```

Класове "set" и "multiset"

Класът **set**, дефиниран в библиотеката **<set>**, представлява множество от елементи, никой два от които не са равни. Елементите са сортирани в нарастващ ред.

Структурата от данни, върху която е изграден **set**, е **червено-черно балансирано дърво**. Така имаме гаранция че добавянето, търсенето и премахването на елемент стават със сложност $O(\log N)$, където **N** е броя на елементите в **set**-а.



Итераторите, предоставени от **set**, са **bidirectional**. Добавянето на елемент става чрез член функцията **insert()**, търсенето – чрез **find()** и изтриването – чрез **erase()**.

Тъй като **set** изисква наредба на елементите, за да работи с дефинирани от вас структури или класове, те трябва да имат предефинирани оператор **<** (за вкарване) и оператор **==** (за търсене). Също както в **priority_queue**, можем да зададем наша логика за сравнение на елементите в **set**-а.

Пример за употреба на **set**:

```
#include <stdio.h>
#include <set>
#include <string>
using namespace std;

struct ltstr
{
    bool operator()(const string &s1, const string &s2) const{
        return strcmp(s1.c_str(), s2.c_str()) > 0;
    }
};
```

```
void iterateOverSet(set<string> s)
{
    for (set<string>::iterator it = s.begin(); it != s.end(); it++)
        printf("%s ", ((string)*it).c_str());
    printf("\n");
}

void iterateOverSet(set<string, ltstr> s)
{
    for (set<string, ltstr>::iterator it = s.begin(); it != s.end(); it++)
        printf("%s ", ((string)*it).c_str());
    printf("\n");
}

int main()
{
    set<string> A; // използваме нормалната подредба на string
    set<string, ltstr> B; // използваме наша подредба

    A.insert("ala");
    A.insert("bala");
    A.insert("ala");

    iterateOverSet(A); // ala bala

    B.insert("ala");
    B.insert("bala");
    B.insert("ala");

    iterateOverSet(B); // bala ala

    set<string>::iterator it = A.find("koko");
    if(it != A.end())
        A.erase(it);
    A.erase(A.find("ala"));

    iterateOverSet(A); // bala

    return 0;
}
```

Разликата между класовете **set** и **multiset**, е че **multiset** може да съдържа повече от един елемент с еднакъв ключ. Метода **count()** дава възможност за преброяването на елементите с даден ключ. Пример за употреба на **multiset**:

```
multiset<string> C;
C.insert("ala");
C.insert("bala");
C.insert("ala");
printf("%d\n", C.count("ala")); //2
```

Ето някои полезни функции, които работят върху всякакви сортирани последователности:

- **set_union** – обединява две сортирани последователности
- **set_intersection** – сечение на две сортирани последователности
- **set_difference** - разлика на две сортирани последователности
- **set_symmetric_difference** – симетрична разлика на две сортирани последователности

Следва пример за използването на тези функции с **multiset**:

```
#include <iostream>
#include <algorithm>
#include <set>
using namespace std;

int main()
{
    const int N = 10;
    int a[N] = {4, 1, 1, 1, 1, 1, 0, 5, 1, 0};
    int b[N] = {4, 4, 2, 4, 2, 4, 0, 1, 5, 5};

    multiset<int> A(a, a + N);
    multiset<int> B(b, b + N);
    multiset<int> C;

    cout << "Set A: ";
    copy(A.begin(), A.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
    cout << "Set B: ";
    copy(B.begin(), B.end(), ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Union: ";
    set_union(A.begin(), A.end(), B.begin(), B.end(),
              ostream_iterator<int>(cout, " "));
    cout << endl;

    cout << "Intersection: ";
    set_intersection(A.begin(), A.end(), B.begin(), B.end(),
                     ostream_iterator<int>(cout, " "));
    cout << endl;

    set_difference(A.begin(), A.end(), B.begin(), B.end(),
                   inserter(C, C.begin()));
    cout << "Set C (difference of A and B): ";
    copy(C.begin(), C.end(), ostream_iterator<int>(cout, " "));
    cout << endl;
}
```

Класове “map” и “multimap”

Map дава *асоциативен масив*, тоест масив от двойки, като по даден ключ (*key*) ви връща стойност (*value*). Класовете **map** и **multimap**, дефинирани в библиотеката **<map>**, дават възможност за

съпоставяне между ключ и стойности. Разликата между тях е, че в **map** на един ключ може да бъде съпоставена най-много една стойност, докато в **multimap** – много.

Добавянето на елемент става чрез метода **insert()**, на който се подава двойката (ключ; стойност).
Пример:

```
map/*multimap*/ <string, int> months;
months.insert(make_pair("January", 31));
```

В **map** за удобство е предефиниран оператора **[]**, което прави достъпването на елементи възможно по следния начин:

```
map/*!!!не работи при multimap!!!*/ <string, int> months;
months["January"] = 31;
printf("%d\n", months["January"]);
```

Макар този оператор да е доста удобен за използване, с него трябва да се внимава много. Проблемът идва от това, че когато по този начин правим опит за достъп на елемент, който не е част от **map**-а, то в **map**-а се добавя елемент със същия ключ и стойност по подразбиране. Пример:

```
map <string, int> months;
months["January"] = 31;
cout<<months.size()<<endl; // размерът е 1
cout<<months["invalid month"]<<endl; //!!!!добавя елемента (invalid month; 0)
cout<<months.size()<<endl; // размера е 2
```

Търсенето и изтриването на елементи става по същия начин, както в **set**, така и в **multiset** – чрез **find()** и **erase()**. Итераторите са **bidirectional**, като всеки итератор е указател към двойката (ключ; стойност).

Също както в **set**, **multiset** и **priority_queue**, може да се дефинира структура, която сравнява ключовете.

За удобство можем да използваме **typedef** за да избегнем многократното писане на нотацията на **map**. Пример за итерация със структура за сравнение и **typedef**:

```
#include <stdio.h>
#include <map>
#include <string>
using namespace std;

struct ltstr
{
    bool operator()(const string &s1, const string &s2) const{
        return strcmp(s1.c_str(), s2.c_str()) > 0;
    }
};

typedef map<string, int, ltstr> myMap;
```

```
void iterateOverMap(myMap s)
{
    for (myMap::iterator it = s.begin(); it != s.end(); it++)
        printf("%s %d\n", ((string)it->first).c_str(), it->second);
    printf("\n");
}

int main()
{
    myMap months;

    months["january"] = 31;
    months["february"] = 28;
    months["march"] = 31;
    months["april"] = 30;
    months["may"] = 31;
    months["june"] = 30;
    months["july"] = 31;
    months["august"] = 31;
    months["september"] = 30;
    months["october"] = 31;
    months["november"] = 30;
    months["december"] = 31;

    iterateOverMap(months);

    return 0;
}
```

Алгоритми

Сортиране

Функцията **sort**, дефинирана в библиотеката **<algorithm>**, е удобна за използване и по същество представлява имплементация на алгоритъма **quick sort**. Параметрите, които подаваме, са указател към началото на редицата, която сортираме, и указател към елемента след края на редицата.

Знаем, че в средния случай, този алгоритъм има сложност $\Theta(N \cdot \log N)$, а в най-лошия - $\Theta(N^2)$. Знаем също така, че най-лошият случай се получава, когато елементите, които сортираме, са подредени първоначално в обратен ред. Въпреки това ние можем да очакваме *sort* да работи със сложност $\Theta(N \cdot \log N)$ във всеки един случай, дори когато подаваме елементи, които са подредени в обратен ред.

```
int a[5] = { 4, 2, 4, 1, 5 };
sort(a, a + 5);
for(int i = 0; i < 5; i++)
    printf("%d ", a[i]); // 1 2 4 4 5
printf("\n");
```


Функцията **sort** нормално използва оператора **<**, за да определи дали един елемент е по-малък от друг, в процеса на сортирането. Вместо да използваме естествената подредба на елементите, които сравняваме, можем и **да подадем като трети параметър на *sort* указател към функция**, която имплементира някаква наша логика за сравнение. Контрактът на тази функция е да връща резултат *true*, ако първият ѝ аргумент е по-малък от втория.

```
#include <stdio.h>
#include <algorithm>
using namespace std;

bool cmp(const int &a, const int &b) { return a > b; }

int main()
{
    sort(a, a + 5, cmp);
    for(int i = 0; i < 5; i++)
        printf("%d ", a[i]); // 1 2 4 4 5
    printf("\n");

    return 0;
}
```

Важно е да се отбележи, че в **pair** са предефинирани операторите **<**, **==**, **>**, както и операторът за присвояване **=**. Сравнението се извършва първо по полето **first** и ако стойностите в него са равни - по полето **second**. Това е демонстрирано в следващия пример, където първоначално сортираме вектора без да му подаваме сравняваща функция. За следващото сортиране използваме версията на функцията **sort**, която като трети параметър приема указател към функция, която имплементира наша специфична логика за сравнение. В нашия случай това е функцията **cmpVector**, в която сравняваме **pair** обектите **единствено по второто им поле**:

```
#include <stdio.h>
#include <algorithm>
#include <vector>
using namespace std;

bool cmpVector(const pair<int, double> a, const pair<int, double> b)
{
    return a.second != b.second ?
        a.second - b.second < 0 :
        a.first - b.first < 0;
}

vector< pair <int, double > > v;

void printVector()
{
    vector< pair <int, double > > :: iterator it;
    for(it = v.begin(); it != v.end(); it++)
        printf("%d %.7f\n", it->first, it->second);
    printf("\n");
}
```

```
int main()
{
    v.push_back(make_pair(3, 0.000001));
    v.push_back(make_pair(2, 3.1415));
    v.push_back(make_pair(1, 2.1));
    v.push_back(make_pair(1, 1.2));
    v.push_back(make_pair(2, 4.999999));
    v.push_back(make_pair(2, 5));

    sort(v.begin(), v.end());
    printVector();

    //1 1.2000000
    //1 2.1000000
    //2 3.1415000
    //2 4.9999999
    //2 5.0000000
    //3 0.0000010

    sort(v.begin(), v.end(), cmpVector);
    printVector();

    //3 0.0000010
    //1 1.2000000
    //1 2.1000000
    //2 3.1415000
    //2 4.9999999
    //2 5.0000000

    return 0;
}
```

Промяна на последователности

- ***swap_ranges*** - разменя елементите в цели последователности. Първата последователност подадена на функцията трябва да е по-малка, защото тя определя големината на разменящите се елементи.
- ***replace*** - заменя всеки елемент в последователността, който е еднакъв на подадената стойност или елемент.
- ***replace_if*** - същото като *replace*, но има и предикат, който определя при какво условие да се заменят.
- ***fill*** - запълва последователност с някаква стойност.
- ***fill_n*** - запълва с някаква стойност първите N елемента от началото на последователността.
- ***remove*** - премахва елементите в последователността равни на подадения елемент.

- ***remove_if*** - Като ***remove***, но премахва елементите, ако изпълняват някакво условие (предикат).
- ***unique*** е особен алгоритъм - той **НЕ** премахва повтарящите се елементи. В действителност контейнера може да е масив и от него не може да се трие.

Ако в подадената му редица има последователни повтарящи се елементи, от тях остава само първия, а другите се преместват в края на редицата. Това означава, че ако подадената редица е сортирана, в началото ѝ ще останат само уникалните елементи, а в края ще бъдат преместени повторенията. ***unique*** връща указател към новия края на уникалната редица - т.е. мястото, което разделя уникалните от повтарящите се елементи.

- ***reverse*** - обръща областта огледално спрямо средата ѝ.
- ***rotate*** - завърта наляво елементите в дадена областта. Избира се един елемент, който да иде наляво и той става първи в областта, елемента след него става втори, а елементите се "залепят" след последния елемент.

Търсене в не сортирани последователности

- ***find*** - намира първото срещане на определена последователност или на 1 елемент в дадена последователност (string в string или число в масив) и връща итератор(от същия тип като стойността на търсеното) към началото на последователността или към самия елемент.
- ***count*** - намира броя на срещанията на определен елемент в дадена последователност. Връща броя на срещания.
- ***min_element*** - намира минималния елемент в дадена последователност.
- ***max_element*** - намира максималния елемент в дадена последователност.
- ***lexicographical_compare*** - проверява коя от двете подадени области е по-голяма в лексикографски (азбучен) ред. Връща **true**, ако първата област е по-голяма и **false**, ако втората е по-голяма.
- ***mismatch*** - намира първото несъответствие между елементите на две последователности. Връща елемент от тип **pair**. Ако няма несъответствие връща **first** да е първия итератор и **second** да е първият2 + (последният1 - първият1). Първата последователност трябва да е с по-малка големина, защото тя е определяща каква част от втората ще бъде сравнена.

```
#include <stdio.h>
#include <algorithm>
#include <iostream>
```

```
#include <vector>
using namespace std;

// a case-insensitive comparison function:
bool mycomp (char c1, char c2)
{
    return tolower(c1) < tolower(c2);
}

bool mypredicate (int i, int j)
{
    return (i == j);
}

int main()
{
    int myints[] = { 10, 20, 30 ,40 };
    int * p;

    //~~~ find ~~~~
    // pointer to array element:
    p = find(myints, myints + 4, 30);
    ++p;
    cout << "The element following 30 is " << *p << endl;
    //The element following 30 is 40

    vector<int> myvector (myints, myints+4);
    vector<int>::iterator it;

    // iterator to vector element:
    it = find (myvector.begin(), myvector.end(), 30);
    ++it;
    cout << "The element following 30 is " << *it << endl;
    //The element following 30 is 40

    //~~~ count ~~~~
    myvector.push_back(20);
    cout << count(myvector.begin(), myvector.end(), 20) << endl; // 2

    //~~~ min/max element ~~~
    cout << *min_element(myvector.begin(), myvector.end()) << endl; // 10
    cout << *max_element(myvector.begin(), myvector.end()) << endl; // 40

    //~~~ lexicographical_compare ~~~
    char first[]="Apple";           // 5 letters
    char second[]="apartment";      // 9 letters

    //Using default comparison (operator<):
    if (lexicographical_compare(first, first + 5, second, second + 9))
        cout << first << " is less than " << second << endl;
    else if (lexicographical_compare(second, second + 9, first, first + 5))
        cout << first << " is greater than " << second << endl;
    else
        cout << first << " and " << second << " are equivalent\n";
}
```

```

//Using mycomp as comparison object:
if (lexicographical_compare(first, first+5, second, second+9, mycomp))
    cout << first << " is less than " << second << endl;
else if (lexicographical_compare(second, second + 9, first, first+
5, mycomp))
    cout << first << " is greater than " << second << endl;
else
    cout << first << " and " << second << " are equivalent\n";

//~~~ mismatch ~~~
myvector.clear();
for (int i = 1; i < 6; i++)
    myvector.push_back (i*10); // myvector: 10 20 30 40 50
int m[] = {10, 20, 80, 320, 1024}; // myints: 10 20 80 320 1024
pair<vector<int>::iterator, int*> mypair;
// using default comparison:
mypair = mismatch (myvector.begin(), myvector.end(), m);
cout << "First mismatching elements: " << *mypair.first;
cout << " and " << *mypair.second << endl;;

mypair.first++; mypair.second++;

// using predicate comparison:
mypair = mismatch (mypair.first, myvector.end(), mypair.second,
mypredicate);
cout << "Second mismatching elements: " << *mypair.first;
cout << " and " << *mypair.second << endl;

return 0;
}

```

Търсене в сортирани последователности

Разгледаните по-долу функции **работят коректно само върху сортирани последователности** и имат логаритмична сложност.

- ***lower_bound*** - намира първия елемент в дадена сортирана последователност, който има стойност по-малка от зададената.
- ***upper_bound*** - същото като *lower_bound*, но намира по-голямата стойност не по-малката.
- ***equal_range*** - двоично търсене, което намира последователността равна на даден елемент, в някаква определена сортирана последователност. Връща тип **pair**, като **first** е равно на итератора, от който започва последователността, а **second** е равно на итератора, в който завършва последователността.
- ***binary_search*** - двоично търсене, което проверява дали елемента е в сортираната последователност. Връща true, ако го има и false, ако го няма.

- **merge** - слива две наредени последователности, запазвайки резултата в наредена трета. Критерия за сравнение може да се промени с функция за сравнение. Последователността, в която се пази резултата се подава като 5 параметър на функцията. **Сложност: Линейна.**
- **includes** - определя дали дадена сортирана последователност съдържа елементите на друга сортирана такава. **Сложност: Линейна.**

```
#include <algorithm>
#include <vector>
using namespace std;

bool mygreater (int i,int j) { return (i > j); }

int main()
{
    int myints[] = {10, 20, 30, 30, 20, 10, 10, 20};
    vector<int> v(myints,myints+8);           // 10 20 30 30 20 10 10 20
    vector<int>::iterator low,up;

    sort (v.begin(), v.end());                // 10 10 10 20 20 20 30 30

    //Returns an iterator pointing to the first element in the sorted range
    //[first,last) which does not compare less than value
    low = lower_bound (v.begin(), v.end(), 20);

    //Returns an iterator pointing to the first element in the sorted range
    //[first,last) which compares greater than value
    up = upper_bound (v.begin(), v.end(), 20);
    // 10 10 10 20 20 20 30 30
    //           ^       ^
    //         low      up
    printf("lower_bound at position %d\n", low - v.begin()); // 3
    printf("upper_bound at position %d\n", up - v.begin()); // 6

    //Returns true if an element in the range [first,last) is equivalent
    //to value, and false otherwise.
    bool found = binary_search(v.begin(), v.end(), 33); //false

    pair<vector<int>::iterator, vector<int>::iterator> bounds;
    bounds = equal_range (v.begin(), v.end(), 20);
    // 10 10 10 20 20 20 30 30
    //           ^       ^
    printf("bounds at positions %d and %d\n",
           bounds.first - v.begin() ,
           bounds.second - v.begin());

    // using "mygreater" as comp:
    sort (v.begin(), v.end(), mygreater);
    bounds = equal_range (v.begin(), v.end(), 20, mygreater);
    // 30 30 20 20 20 10 10 10
    //           ^       ^
    printf("bounds at positions %d and %d\n",
           bounds.first - v.begin() ,
           bounds.second - v.begin());
}
```

```
        return 0;  
    }
```

Генериране на пермутации

```
#include <iostream>  
#include <algorithm>  
using namespace std;  
  
int main ()  
{  
    int p[] = {1,2,3};  
  
    //Rearranges the elements in the range [first, last) into  
    //the lexicographically next greater permutation of elements.  
    //The comparisons of individual elements are performed using  
    //either operator< for the first version, or comp for the second.  
    do  
    {  
        printf("%d %d %d\n", p[0], p[1], p[2]);  
    }  
    while (next_permutation (p, p + 3));  
  
    //1 2 3  
    //1 3 2  
    //2 1 3  
    //2 3 1  
    //3 1 2  
    //3 2 1  
  
    printf("\n");  
    printf("%d %d %d\n", p[0], p[1], p[2]); //1 2 3  
  
    //lexicographically next smaller permutation of elements  
    prev_permutation (p, p + 3);  
    printf("%d %d %d\n", p[0], p[1], p[2]); //3 2 1  
  
    //lexicographically next smaller permutation of elements  
    prev_permutation (p, p + 3);  
    printf("%d %d %d\n", p[0], p[1], p[2]); //3 1 2  
  
    return 0;  
}
```

Някои често ползвани функции

<http://www.informatika.bg/lectures?topic=STDLIB>

Използвана литература

- CS Club на СУ - <http://judge.openfmi.net:9080/mediawiki/index.php>
- C++ References - <http://www.cplusplus.com/reference/stl/>
- Стандартна библиотека, част I - <http://www.informatika.bg/lectures?topic=STL>
- Стандартна библиотека, част II - <http://www.informatika.bg/lectures?topic=STDLIB>