ლაბ 9: map და set კონტეინერის გამოყენების მაგალითები

განსახილველი საკითხები:

- set<> - ის ელემენტების შედარება და ძებნა
- მინიმალური და მაქსიმალური ელემენტის განსაზღვრა set კონტეინერში >>>
- მეჩხერი ვექტორის წარმოდგენის შესახებ >>>
- სავარჯიშოები >>>

set<> - ის ელემენტების შედარება და ძებნა

განვიხილოთ შემდეგი პროგრამა:

```
#include<iostream>
#include<set>
#include<string>

using namespace std;

struct cmpBySecond {
	bool operator()(const pair<string, int> & a, const pair<string, int>& b) const {
		return a.second < b.second;
	}
};

int main()
{
	//set<pair<string, int>, cmpBySecond> s;
	set<pair<string, int>> s;
	s.insert(make_pair("Salome",97));
	s.insert(make_pair("Aram", 89));
	s.insert(make_pair("Kate", 97));
	s.insert(make_pair("Kate", 88));
	for (auto m:s)	cout << m.first << "  " <<m.second << endl;
}
```

თუ გამოვბეჭდავთ შედეგებს, დავრწმუნდებით რომ წყვილების ელემენტები ერთმანეთს დარდება ლექსიკოგრაფიულად. თუ მეორე სტრიქონს მოვაქცევთ კომენტარში, ხოლო პირველს გავანთავისუფლებთ, დავწმუნდებით, რომ ამ შემთხვევაში პროგრამა ერთმანეთისგან ვერ არჩევს ისეთ წყვილებს, რომელთა მეორე კომპონენტები ტოლია. ეს იმიტომ, რომ ელემენტების შედარება ხდება მეორე კომპონენტით. შესაბამისად, არც ("Kate", 97) არის მეტი ("Salome",97)-ზე და არც პირიქით. ამის გამო ისინი ითვლებიან ტოლად. შედეგად, სიმრავლე დუბლირებებს გამორიცხავს და მეორე დუბლიკატს არ ჩასვამს სიმრავლეში.

როგორ მოვძებნოთ ელემენტი „რთულ“ სიმრავლეში? თუ შედარება მეორე კომონენტებით ხდება, მაშინ უნდა შევქმნათ საჭირო მნიშვნელობის მქონე წყვილი და მოვძებნოთ. პირველ კომპონენტს მნისვნელობა არ აქვს, რადგან ყველა წყვილი, რომლის მეორე კომპონენტები ტოლია, აგრეთვე ერთმანეთის ტოლად ითვლება>

შემდეგი პროგრამაში:

```
int main()
{
	set<pair<string, int>, cmpBySecond> s;
	s.insert(make_pair("Salome",97));
	s.insert(make_pair("Aram", 89));
	s.insert(make_pair("Kate", 97));
	s.insert(make_pair("Kate", 88));
	for (auto m:s)	cout << m.first << "  " <<m.second << endl;
	auto i = s.find(make_pair("", 88));
	if (i != s.end())
		cout << endl << "Found (" << i->first << ',' << i->second << ')' << endl;
}
```

ვხედავთ თუ რა შედეგი მოსდევს (`"", 88`) წყვილის ძებნას: პროგრამა ვ=გვეუბნება რომ იპოვა მისი ტოლი: (`"Kate", 88`).

<<< მინიმალური და მაქსიმალური ელემენტის განსაზღვრა set კონტეინერში

განვიხილოთ შემდეგი მარტივი პროგრამა:

```
#include<iostream>
#include<list>
#include<set>

using namespace std;

int main()
{
	list<int> lst = { 1, 433, 342, 13, 31, -32, 5, -43 };
	set<int> s(lst.begin(),lst.end());
	cout << "Minimum value in the tree: " << *s.begin() << endl;
	cout << "Maximum value in the tree: " << *s.rbegin() << endl;
}
```

რომლის შედეგია:

```
Minimum value in the tree: -43
Maximum value in the tree: 433
Press any key to continue . . .
```

ეს გზა გაცილებით სწრაფია, ვიდრე C++ ენის სტანდარტული საშუალებების გამოყენება, რომლებიც მოკლედ არის განხილული ბოლო სავარჯიშოში.

მეჩხერი ვექტორის წარმოდგენის შესახებ. ჩვენი მიზანია, ორი განსხვავებული გზით ვიანგარიშოთ რიცხვთა ორი კრებულის სკალარული ნამრავლი. უნდა წინასწარ აღვნიშნოთ, რომ მეჩხერ მონაცემებთან სამუშაოდ კერძოდ ამ მიზნით შემუშავებული განსხვავებული ფორმატები (მონაცემთა სტრუქტურები) არსებობს. სტანდარტული კონტეინერები არ მიეკუთვნება მათ რიცხვს. თუმცა, მულტი-სიმრავლე საშუალებას გვაძლევს რომ ასეთი მონაცემები შედარებით ეკონომიურად განვათავსოთ მეხსიერებაში.

ქვემოთ, პირველი პროგრამა სკალარულ ნამრავლს ითვლის „უხეში ძალის“ გამოყენებით, ანუ პრიმიტიულად: რიცხვთა კრებულების შესანახად ვიყენებთ ვექტორის სტრუქტურას, და ერთი განმეორების შეტყობინებით იწერება მათი ნამრავლი.

ეს გზა არაა საუკეთესო, რადგან ჩვენს შემთხვევაში საქმე გვაქვს გაიშვიათებულ, ანუ გამეჩხერებულ კრებულთან, რაც ნიშნავს, რომ ნულოვანი ელემენტების რაოდენობა ერთი რიგით მეტია არანულოვანების რაოდენობაზე.

```
#include<iostream>
#include<vector>

using namespace std;
int main()
{
	cout << "computing an inner  product of tuples as vectors"<< endl;
	const int N =600000; // length of tuples x and y
	const int S = 10;  // sparseness factor

	cout << "Initializing..." << endl;
	vector<double> x(N),y(N);
	int k;
	for(k=0; 3*k*S < N; k++)
		x[3*k*S] = 1.0;
	for(k=0; 5*k*S < N; k++)
		y[5*k*S] = 1.0;

	cout << "Computing inner product by brute force:" << endl;
```

```
	double sum(0.0);
	for(k=0; k < N; k++)
		sum += x[k]*y[k];
	cout <<"sum = " << sum << endl;
	return 0;
}
```

ახლა, იგივე ამოცანა გადავჭრათ ასახვის სტრუქტურის გამოყენებით. შევქმნათ ორი ობიექტი, თითოეულში ვინახავთ შესაბამისი კრებულის მხოლოდ არანულოვან კომპონენტებს. გასაღებად ვიღებთ კომპონენტის ინდექსს (ნულიდან დაწყებული).

```
#include<iostream>
#include<map>

using namespace std;
int main()
{
	cout << "computing an inner  product of tuples as maps"<< endl;
	const int N =600000; // length of tuples x and y
	const int S = 10;  // sparseness factor

	cout << "Initializing..." << endl;
	map<int,double> x, y;
	int k;
	for(k=0; 3*k*S < N; k++)
		x[3*k*S] = 1.0;
	for(k=0; 5*k*S < N; k++)
		y[5*k*S] = 1.0;

	cout << "Computing inner product taking advantage of sparsenesxs:" << endl;
	double sum(0.0);
	map<int,double>::iterator ix(x.begin()),iy;
	while(ix != x.end())
	{
		int i = ix -> first;
		iy = y.find(i);
		if(iy != y.end())
			sum += ix->second * iy->second;
		ix++;
	}

	cout <<"sum = " << sum << endl;
	return 0;
}
```

შევადაროთ ერთმანეთს ამ პროგრამების შესრულებისთვის საჭირო ოპერაციების რაოდენობები.

ბოლო პროგრამა ასრულებს მხოლოდ 4000 შეკრებას და ამდენივე გამრავლებას, მაშინ როცა წინა შემთხვევაში თითოეულს ჭირდებოდა 600 000. შემდეგი ცხრილი გვიჩვენებს ინდექსაციის, იტერატორების ოპერაციების და ძებნის ბიჯებს **find** ალგორითმში.

გამოთვლები ეფუძნება იმ ფაქტს, რომ x-ში შენახულია 600 000 / 30 = 20 000 მონაცემი, ხოლო y-ში შენახულია 12 000 მონაცემი. ერთი **find** ალგორითმი y-ზე ასრულებს საშუალოდ log(12 000) = 13.5 ოპერაციას.

ოპერაციები	ვექტორის შემთხვევა	ასახვის შემთხვევა
გამრავლება	600 000	4 000
შეკრება	600 000	4 000
k++ ინკრემენტი	600 000	
ix++ იტერირება		20 000

`x[k]` მნიშვნელობის აღება	600 000	
`y[k]` მნიშვნელობის აღება	600 000	
`*ix` მნიშვნელობის აღება		20 000
ძებნის ბიჯები **`find`** -ში		270 000 (13.5 * 20 000)
`*iy` მნიშვნელობის აღება		12 000

<<< სავარჯიშოები

1. პირველი ამოცანა გადაჭერით სხვადასხვა გაიშვიათების ფაქტორისთვის (ანუ ცვალეთ

```
const int S = 10;  // sparseness factor).
```

შეგიძლიათ თუ არა მოიფიქროთ ალტერნატიული გზა მოცემული პრობლემის გადასაჭრელად?

2. გადააკეთეთ ზემოთ მოყვანილი პროგრამები ისე, რომ ორივე შემთხვევაში დაიბეჭდოს პროგრამის შესრულებისათვის საჭირო დრო.

3. აქვს თუ არა მნიშვნელობა, როგორი თანმიმდევრობით ვიღებთ თანამამრავლებს

```
sum += ix->second * iy->second;
```

შეტყობინებაში. სხვა სიტყვებით, `while(ix != x.end())` განმეორებაში აქვს თუ არა მნიშვნელობა რომელ კონტეინერს შემოვივლით?

4. განვიხილოთ შემდეგი მარტივი პროგრამა და მისი შესაძლო შედეგი:

```
#include <algorithm>
#include <iostream>
#include <string>

int main()
{
    std::cout << "larger of 1 and 9999: " << std::max(1, 9999) << '\n'
          << "larger of 'a', and 'b': " << std::max('a', 'b') << '\n'
          << "longest of \"foo\", \"bar\", and \"hello\": " <<
          std::max({ "foo", "bar", "hello" },
          [](const std::string& s1, const std::string& s2) {
          return s1.size() < s2.size();
    }) << '\n';
    std::cout << "larger of 32,213,31,23,3,21,3,-21,3,251: " <<
          std::max({ 32, 213, 31, 23, 3, 21, 3, -21, 3, 251 }) << '\n';
}
```

```
larger of 1 and 9999: 9999
larger of 'a', and 'b': b
longest of "foo", "bar", and "hello": hello
larger of 32,213,31,23,3,21,3,-21,3,251: 251
Press any key to continue . . .
```