ლაბ 13 დაუხარისხებელი სიმრავლე და ასახვა

განსახილველი საკითხები:

- **ამოცანა 1** /კონტეინერის bucket ინტერფეისის გამოყენების მაგალითი/
- **ამოცანა 2** / bucket ინტერფეისის გამოყენების სხვა შემთხვევა/ >>>
- **ამოცანა 3** / `unordered_set` ტიპის ობიექტისთვის ჰეშ-ფუნქციის გადაწოდება/ >>>
- **სავარჯიშოები** >>>

ამოცანა 1 /კონტეინერის bucket ინტერფეისის გამოყენების მაგალითი/.

შევქმნათ დაუხარისხებელი სიმრავლის ობიექტი, შევავსოთ და დავბეჭდოთ მის შესახებ შეძლებისდაგვარად სრული ცნობები. დავბეჭდოთ აგრეთვე თვით სიმრავლე, მისი „ვედროების“ (ინგლისურად buckets) შიგთავსის მითითებით. შემდეგ, დავამატოთ ელემენტები და კვლავ დავბეჭდოთ მსგავსი ცნობები განახლებული სიმრავლის შესახებ.

ამოხსნა. შესაბამისი პროგრამა შედგება თავსართი ფაილისგან და პროგრამა დრაივერისგან.

```
//buckets.hpp
#include<iostream>
#include<iomanip>
#include<utility>
#include<iterator>
#include<typeinfo>

//generic output for pairs (map elements)
template <typename T1, typename T2>
std::ostream& operator << (std::ostream& strm, const std::pair<T1, T2>&p)
{
	return strm << "[" << p.first << "," << p.second << "]";
}

template <typename T>
void printHashTableState(const T& cont)
{
	//basic layout data:
	std::cout << "size:              " << cont.size() << "\n";
	std::cout << "buckets:           " << cont.bucket_count() << "\n";
	std::cout << "load_factor:       " << cont.load_factor() << "\n";
	std::cout << "max_load_factor:   " << cont.max_load_factor() << "\n";

	//iterator category:
	if (typeid(typename std::iterator_traits<typename T::iterator::iterator_category>)
		== typeid(std::bidirectional_iterator_tag))
	{
		std::cout << "chaining style: doubly-linked" << "\n";
	}
	else
	{
		std::cout << "chaining style: singly-linked" << "\n";
	}

	//elements per bucket:
	std::cout << "data: " << "\n";
	for (auto idx = 0; idx != cont.bucket_count(); ++idx)
	{
		std::cout << " b[" << std::setw(2) << idx << "]: ";
		for (auto pos = cont.begin(idx); pos != cont.end(idx); ++pos)
		{
			std::cout << *pos << " ";
		}
```

```
            std::cout << "\n";
        }
        std::cout << std::endl;
}
```

პროგრამა დრაივერი:

```
#include <unordered_set>
#include <iostream>
#include "buckets.hpp"

int main()
{
        //create and initialize unordered set:
        std::unordered_set<int> intset = { 1, 2, 3, 5, 11, 13, 17, 19 };
        printHashTableState(intset);

        //insert some additional values (might cause rehash)
        intset.insert({ -7, 17, 33, 4 });
        printHashTableState(intset);
}
```

<<< **ამოცანა 2** / bucket ინტერფეისის გამოყენების სხვა შემთხვევა/. ახლა, შევქმნათ დაუხარისხებელი ასახვის ობიექტი, რომელიც შეინახავს ორი სტრინგისგან შედგენილ წყვილებს. პროგრამა გამოიყენებს ჩვენს მიერ შექმნილ თავსართ ფაილს, რომელშიც გადატვირთულია წყვილების გამოტანის განზოგადებული (ტემპლიტიანი) ფუნქცია.

```
#include <unordered_map>
#include <string>
#include <iostream>
#include<utility>
#include "buckets.hpp"

using namespace std;

int main()
{
        //create and initialize unordered set:
        unordered_multimap<string,string> dict = {
                {"day","tag"},
                { "strange", "fremd" },
                { "car", "Auto" },
                { "smart", "elegant" },
                { "trait", "Markmal" },
                { "strange", "seltsam" },
        };
        printHashTableState(dict);

        //insert some additional values (might cause rehash)
        dict.insert({ {"smart", "raffinient"},
                      {"smart", "klug"},
                      { "clever", "raffinient" }
                    });
        printHashTableState(dict);

        dict.max_load_factor(0.7);
        printHashTableState(dict);
}
```

<<< **ამოცანა 3** / unordered_set ტიპის ობიექტისთვის ჰეშ-ფუნქციის გადაწყდება/. შევცვალოთ პროგრამა დრაივერი შემდეგნაირად:

```
#include <unordered_set>
#include <string>
#include <iostream>
#include<utility>
#include "buckets.hpp"

using namespace std;

int main()
{
	//create and initialize unordered set:
	unordered_set<string> s = {
		{ "day"},
		{ "strange" },
		{ "car" },
		{ "Auto" },
		{ "smart"},
		{ "trait" },
		{ "strange"},
	};
	printHashTableState(s);

	//insert some additional values (might cause rehash)
	s.insert({ {"raffinient"},{"klug"}, { "clever" } });
	printHashTableState(s);

	s.max_load_factor(0.7);
	printHashTableState(s);
}
```

და შეცვალოთ იგი, - კონტეინერს განაცხადში გადავაწოდოთ მომხმარებლის მიერ შექმნილი ჰეშ-ფუნქცია.

განსხვავებით ფუნქციების შექმნის ადრე განხილული მექანიზმებისგან (ფუნქცია, ბინდერი, ლამბდა), ახლა ჩვენ გამოვიყენებთ ფუნქტორის ცნებას, რომელიც სრულად ორიენტირებულია ობიექტებზე ქმნის კლასს, რომელიც ქმნის ოპერატორებს თავის ობიექტებად.

წინასწარ განვიხილოთ ასეთი საკითხი. ჩვენს მაგალითში სტრინგები წარმოადგენენ გასაღებებს. ამიტომ გვჭირდება ალგორითმი, რომელიც სტრინგს გარდაქმნის უნიშნო მთელად. ეს ალგორითმი გაფორმდება ჰეშ-ფუნქციად. შემდეგ, ნაშთის აღების ოპერატორით კონსტრუქტორი თვითონ გარდაქმნის ჰეშ-მნიშვნელობებს ინდექსებად. მომხმარებელი ამ პროცესს C++ -ში სრულად ვერ აკონტროლებს, რადგან მას არ შეუძლია ზუსტად განუსაზღვროს პროგრამას სიმრავლეში რამდენი ვედრო (სია) იქნება.

განვიხილოთ ერთი ასეთი გავრცელებული ალგორითმი. ვთქვათ მოცემულია გასაღები, რომელიც წარმოადგენს სტრინგს. დავამუშაოთ იგი სიმბოლო-სიმბოლოდ. თავიდან ავიღოთ n = 0. ყოველი განსახილველი სიმბოლოსთვის, n-ის მიმდინარე მნიშვნელობას ვამრავლებთ 8-ზე და ვამატებთ სიმბოლოს კოდს. მაგალითად, გასაღებისთვის key = “and” გამოთვლებს აქვს სახე:

```
n= (0*8) + 97 = 97

n= (97*8) + 110 = 886

n= (886*8) + 100 = 7188
```

ყველა სიმბოლოს დამუშავების შემდეგ, შესაძლოა შეგვრჩეს უარყოფითი რიცხვი (გადავსების გამო), რისთვისაც უნდა ვიყოთ მზად. საბოლოოდ საჭირო ფუნქტორს აქვს სახე:

```
class hFstring
{
public:
	unsigned int operator() (const string& item) const
```

```
	{
		unsigned int prime = 2049982463;
		int n = 0, i;
		for (i = 0; i < item.length(); ++i)
			n = n * 8 + item[i];
		return n>0 ? (n&prime) : (-n%prime);
	}
};
```

ხოლო განაცხადში ემატება ჰეშ-ფუნქცია:

```
unordered_set<string, hFstring> s = {
		{ "day"},
		{ "strange" },
		{ "car" },
		{ "Auto" },
		{ "smart"},
		{ "trait" },
		{ "strange"},
	};
```

პროგრამის გაშვების შემდეგ ვხედავთ, რომ შედეგი იცვლება. თუმცა ცხადია, რომ C++ -ის მიერ გულისხმობის პრინციპით გამოყენებული ჰეშ-ფუნქცია უკეთესად ახდენდა ჰეშ-მნიშვნელობების გაბნევას.

<<< სავარჯიშოები

1. ფუნქტორების საშუალებით შექმენით უნარული პრედიკატები, მაგალითად, უარყოფითი რიცხვის, ან სამის ჯერადი მთელი რიცხვების დასადგენად. გასინჯეთ მისი მოქმედება პროგრამულად. მაგალითად, ბოლო მაგალითში შექმნილი ფუნქტორის გასინჯვა შეიძლება ასე: შევქმნათ ობიექტი (ოპერატორს) განაცხადით:

   ```
   hFstring hf;
   ```

   და დავბეჭდოთ ჰეშ-მნიშვნელობა გასაღებზე `"Algorithm"`, ანუ

   ```
   cout << hf("Algorithm") << endl;
   ```

   მივიღებთ 1343230637 სიდიდეს.

2. ფუნქტორების საშუალებით შექმენით რამდენიმე ბინარული პრედიკატი, რომლებიც ერთმანეთს შეადარებს რომელიმე კლასის ობიექტებს.

3. წინა მეცადინეობის მაგალითში, შექმენით ჰეშ-ფუნქცია მთელი რიცხვების ჰეშირებისთვის და გადააწოდეთ იგი დაუხარისხებელი სიმრავლის ობიექტის განაცხადში.