ლაბ 6: ბინარული გროვა

განსახილველი საკითხები:

- გროვასთან დაკავშირებული STL -ის ზოგიერთი ალგორითმი
- მაგალითი >>>
- სავარჯიშოები >>>

გროვასთან დაკავშირებული STL -ის ზოგიერთი ალგორითმი

ბინარული გროვის შესაქმნელად და მასზე მანიპულირებისათვის, STL გვამარაგებს შემდეგი ალგორითმებით:

```
template<typename RandomAccessIterator>
void push_heap(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>
void push_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<typename RandomAccessIterator>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>
void pop_heap(RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<typename RandomAccessIterator>
void make_heap(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>
void make_heap (RandomAccessIterator first, RandomAccessIterator last, Compare comp);

template<typename RandomAccessIterator>
void sort_heap(RandomAccessIterator first, RandomAccessIterator last);

template<typename RandomAccessIterator, typename Compare>
void sort_heap (RandomAccessIterator first, RandomAccessIterator last, Compare comp);
```

ალგორითმების აღწერა:

- თუ ფრაგმენტი `[first, last-1)` არის გროვა, `push_heap` გადაადგილებს `[first, last)`-ის ელემენტებს და ამ ფრაგმენტს გარდაქმნის გროვად. ამ დროს ჩვენ ვამბობთ, რომ `(last-1)`-ე ელემენტი შეყვანილ იქნა გროვაში;
- თუ `[first,last)` ფრაგმენტი არის გროვა, მაშინ `pop_heap` ადგილებს გაუცვლის `first` და `(last-1)` ადგილებზე მოთავსებულ მნიშვნელობებს და შემდეგ ელემენტებს ისე გადაადგილებს `[first,last-1)`-ში, რომ ეს ფრაგმენტი ისევ გროვად იქცევა. ამ დროს ვამბობთ, რომ `(last-1)`-ე ელემეტი გამოყვანილ იქნა გროვიდან;
- `make_heap` გადაადგილებს `[first,last)` ფრაგმენტის ელემენტებს და ამ ფრაგმენტს აქცევს გროვად;
- `sort_heap` დაახარისხებს `[first,last)` ფრაგმენტში შექმნილი გროვის ელემენტებს.

სისწრაფე: ვთქვათ, N არის `[first,last)` ფრაგმენტის ელემენების მიერ მითითებული მნიშვნელობების რაოდენობა. მაშინ, `push_heap` და `pop_heap` სრულდება ლოგარითმულ დროში; `make_heap` სრულდება წრფივ დროში, ხოლო `sort_heap` სრულდება N logN დროში.

<<< მაგალითი. განვიხილოთ შემდეგი პროგრამა, რომელიც იყენებს ამ ფუნქციებს.

```
#include<iostream>
```

```
#include<algorithm>
#include<vector>
#include <iterator>
using namespace std;
int main()
{
	vector<int> v {100, 630, 134, 24, -34, 384, 34, 100};
	ostream_iterator<int> out(cout, " ");
	copy(v.begin(), v.end(), out);
	cout << endl;

	for (int i = 2; i <= v.size(); i++)
		push_heap(v.begin(), v.begin() + i);
	cout << "This is heap:\n";
	copy(v.begin(), v.end(), out);
	cout << endl;

	for (int i = v.size(); i >= 2; i--)
		pop_heap(v.begin(), v.begin() + i);
	cout << "This is sorted container:" << endl;
	copy(v.begin(), v.end(), out);
	cout << endl;

	random_shuffle(v.begin(), v.end());
	cout << "This is shuffled container:" << endl;
	copy(v.begin(), v.end(), out);
	cout << << endl;

	make_heap(v.begin(), v.end());
	cout << "This is heap:\n";
	copy(v.begin(), v.end(), out);
	cout << endl;

	sort_heap(v.begin(), v.end());
	cout << "This is sorted container:" << endl;
	copy(v.begin(), v.end(), out);
	cout << endl;
}
```

შედეგს აქვს სახე:

```
100 630 134 24 -34 384 34 100
This is heap:
630 100 384 100 -34 134 34 24
This is sorted container:
-34 24 34 100 100 134 384 630
This is shuffled container:
100 24 384 34 -34 134 630 100
This is heap:
630 100 384 34 -34 134 100 24
This is sorted container:
-34 24 34 100 100 134 384 630
Press any key to continue . . .
```

თუ იგივე ამოცანაში კონტეინერად გამოვიყენებთ მასივს, გვექნება:

```
#include<iostream>
#include<algorithm>
#include <iterator>
using namespace std;
int main()
{
	double v[] {100, 630, 134, 24, -34, 384, 34, 100};
	int size = sizeof(v) / sizeof(v[0]);
	ostream_iterator<int> out(cout, " ");
	copy(v, v+size, out);
	cout << endl << endl;

	for (int i = 2; i <= size; i++)
		push_heap(v + 0, v + i);
	cout << "This is heap:\n";
	copy(v + 0, v + size, out);
	cout << endl << endl;

	for (int i = size; i >= 2; i--)
		pop_heap(v, v + i);
	cout << "This is sorted container:" << endl;
	copy(v, v+size, out);
	cout << endl << endl;

	random_shuffle(v, v + size);
	cout << "This is shuffled container:" << endl;
	copy(v, v+size, out);
	cout << endl << endl;

	make_heap(v, v + size);
	cout << "This is heap:\n";
	copy(v, v + size, out);
	cout << endl << endl;

	sort_heap(v, v+size);
	cout << "This is sorted container:" << endl;
	copy(v, v + size, out);
	cout << endl << endl;
}
```

<<< სავარჯიშოები

1. შექმენით მსგავსი პროგრამა, რომელიც შექმნის გროვას

   a) სიმბოლოებისგან
   b) სტრინგების;გან
   c) ნამდვილი რიცხვებისგან

და ჩაატარებს ანალოგიურ მანიპულაციებს.

2. შექმენით მსგავსი პროგრამა ვულკანების გროვის შესაქმნელად.

3. გაარჩიეთ შემდეგი პროგრამა, რომელიც გარკვეული სახის წყვილებისგან ქმნის გროვას:

```cpp
#include<iostream>
#include<vector>
#include<algorithm>
#include<string>
using namespace std;

bool comparePairs(const pair<string, int>& a, const pair<string, int>& b)
{
	return (a.first < b.first);
}

int main()
{
	vector<pair<string, int>> v
	{
		pair<string, int>("Giorgi", 87),
		pair<string, int>("Anni", 97),
		pair<string, int>("Ket", 91),
		pair<string, int>("Lala", 90),
		pair<string, int>("Abraam", 93),
		pair<string, int>("Gogi", 77)
	};
	make_heap(v.begin(), v.end());
	sort_heap(v.begin(), v.end());
	for (int i = 0; i < v.size(); ++i)
		cout << v[i].first << "   " << v[i].second << endl;
}
```

შედეგს აქვს სახე:

```
Abraam   93
Anni   97
Giorgi   87
Gogi   77
Ket   91
Lala   90
Press any key to continue . . .
```

4. წინა ამოცანის პროგრამაში გამოიყენეთ ბინარული პრედიკატი, შემდეგ შეცვალეთ ბინარული პრედიკატი სხვადასხვანაირად და გააანალიზეთ მიღებული შედეგები.