STL. Greedy

Ioniță Alexandru

October 13, 2020

Faculty of Computer Science Iași

Outline i

STL

- string
- pair
- vector
- stack
- queue
- deque
- priority_queue
- set
- map
- Keep in mind!

Outline ii

Greedy

Activity Selection Problem

Proposed Problems

STL

string

Declaration and basic operators

```
string a, b, c(4, 'a'); // a, b are empty strings
                        // c is "aaaa"

a = "Hello";          // a is "Hello"
b = a + "␣2019";      // b is "Hello 2019"

a[0] = 'h';           // a in "hello 2019"
                      // b is "Hello 2019"
c = "bye␣bye";

if(a > b) {           // this is true (in ascii 'h' > 'H')
  cout << c.size();   // prints 7
}

a = "Oh,␣" + a;       // a is now "Oh, hello 2019"
```

string

other functions

```
string a, b, c;         // a, b, c are empty strings

a = "Hello,␣2019!";
b = a.substr(1, 3);     // b is "ell"
b += "a!"               // b is "ella!"

int found = a.find("2019");  // found is 7
if (found != std::string::npos)
  cout << "first␣'2019'␣found␣at:␣" << found << '\n';

int found = a.find("2020");  // found is string::npos
if (found == std::string::npos)
  cout << "Not␣found";         // prints "Not found"
```

pair

```cpp
pair<int, string> p;
p.first = 12;
p.second = "PC";

cout << (p > make_pair(12, "AC"));
    // prints 1 (the statement is true)

pair<int, pair<int, string> > p2;
```

vector

An array, dynamically allocated.

```cpp
std::vector<int> v;

v.push_back(12);              // v is: [12]
v.push_back(75);              // v is: [12, 75]
v.push_back(15);              // v is: [12, 75, 15]

cout << v.size();             // prints 3

sort(v.begin(), v.end());     // v is: [12, 15, 75]
cout << v[0] << '␣' << v[2];  // prints "12 75"

v.pop_back();                 // v is [12, 15]
```

stack

Last in, first out.

A collection of elements, which supports 3 operations:

- push(x) - adds element 'x' to the stack
- top() - returns the last inserted element
- pop() - removes the last inserted element

```
std::stack<int> mystack;

mystack.push(12);          // the stack now is: [12]
mystack.push(15);          // the stack now is: [15, 12]
mystack.push(75);          // the stack now is: [75, 15,
    12]

cout << mystack.top();     //  prints 75

mystack.pop();             // the queue now is: [15, 12]

cout << myqueue.top();     // prints 15
```

queue

First in, first out.

A collection of elements, which supports 3 operations:

- push(x) - adds element ’x’ to the stack
- front() - returns the first inserted element
- pop() - removes the first inserted element

```
std::queue<int> myqueue;

myqueue.push(12);              // the queue now is: [12]
myqueue.push(15);              // the queue now is: [12, 15]
myqueue.push(75);              // the queue now is: [12, 15,
    75]

cout << myqueue.front();       // prints 12

myqueue.pop();                 // the queue now is: [15, 75]

cout << myqueue.front();       // prints 15
```

deque

double ended queue

A collection of elements, which supports 6 operations:

- push_front(x) - adds element 'x' to the front
- push_back(x) - adds element 'x' to the back
- front() - returns the element from the front
- back() - returns the element from the back
- pop_front() - removes the element from the front
- pop_back() - removes the element from the back

```
std::deque<int> d;

d.push_front(1);
d.push_front(2);
d.push_front(3);    // d is [3, 2, 1]
cout << d[2];       // prints 2

d.pop_back();       // d is [3, 2]
d.back()            // is 2
```

priority_queue

- push(x) - adds element 'x' to the stack
- top() - returns the element with the highest priority
- pop() - removes the first inserted element

```
std::priority_queue<int> pq;

pq.push(33);
pq.push(95);
pq.push(10);
pq.push(27);

cout << pq.top();      // prints 95
pq.pop();              // removes 95

while(!pq.empty()) {
  cout << pq.top() << '␣';
  pq.pop();
}                     // prints "33 27 10"
```

priority_queue Custom Comparator

```
bool my_comparator(int a, int b) {
  return a > b;
}
...
priority_queue<int, vector<int>, function<bool(int,int)> >
        pq(my_comparator);

pq.push(33);  pq.push(95);  pq.push(10);  pq.push(27);

cout << pq.top();     // prints 10
pq.pop();             // removes 10

while(!pq.empty()) {
  cout << pq.top() << '␣';
  pq.pop();
}                  // prints ”27 33 95”
```

- push(x) - adds element 'x' to the stack
- top() - returns the element with the highest priority
- pop() - removes the first inserted element

set

A set of (sorted) elements. Each of them apperas exactly once.

```
set<int> s;

s.insert(1);      // s contains {1}
s.insert(2);      // s contains {1, 2}
s.insert(3);      // s contains {1, 2, 3}
s.insert(2);      // s contains {1, 2, 3}

s.erase(2);       // s is {1, 3}

set<int>::iterator it = s.find(3); // What?
// iterators are 'pointers' for STL structures
// the find method works in O(log(N)),
// where N is the dimension of the set

for (set<int>::iterator it = s.begin(); it != s.end(); it++)
  cout << *it << '␣';
// prints: 1 2 3

cout << *s.begin(); // prints 1
```

set

Other interesting usages

```
set <string> s;

s.insert("Salut");        // s contains {"Salut"}
s.insert("Hello");        // s contains {"Hello", "Salut"}
s.insert("Salut");        // s contains {"Hello", "Salut"}

if(s.find("Salut") != s.end()) // true
  cout << "Yes";
  // the find method works in O(log(N) * M),
  // where N is the dimension of the set
  // and M is the length of the longest string to be
     compared
  // it is doing log(N) comparisons
```

set

multiset

```
multiset <string> s;  // same as set, but supports
                      //multiple elements with the same
                         value

s.insert("Salut");      // s contains {"Salut"}
s.insert("Hello");      // s contains {"Hello", "Salut"}
s.insert("Salut");      // s contains {"Hello", "Salut", "
    Salut"}

cout << s.count("Salut"); // prints 2
  // count works in O(log(N) * M) as well
```

set

set of pair/struct

```
set <pair<string, int>> s;
s.insert(make_pair("Hello", 155));

struct mystruct {
  ...
  bool operator<(mystruct a) const {
    ...
  }
};

set <mystruct> ss;
ss.insert(...);
```

set

set of pair/struct

```
set <pair<string , int>> s;
s.insert(make_pair("Hello", 155));

struct foo{
  int key;
};
inline bool operator<(const foo& lhs, const foo& rhs){
  return lhs.key < rhs.key;
}

set <foo> sss;
sss.insert(...);
```

set

lower_bound/upper_bound

```
std::set<int> s;
std::set<int>::iterator itlow,itup;

for(i=1; i<9; i++) s.insert(i*10); // 10 20 30 40 50 60 70 80

itlow=s.lower_bound (30);           //       ^
itup=s.upper_bound (60);            //                   ^

myset.erase(itlow,itup);                // 10 20 70 80
```

map

Maps are associative containers that store elements formed by a combination of a key value and a mapped value, following a specific order.

Elements are stored as pairs (key, value), in sorted order based on keys

```
map<int, int> m;
m[6] = 5;         // m contains {(6, 5)}
m[5] = 1;         // m contains {(5, 1), (6, 5)}
m[5] = 2;         // m contains {(5, 2), (6, 5)}
m[1000009] = 1;   // m contains {(5, 2), (6, 5), (1000009, 1)}

cout << m.begin()->first;     // prints 5
cout << m.begin()->second;    // prints 2

m.erase(1000009); // m contains {(5, 2), (6, 5)}

cout << m[62];                // prints 0
        // creates the entry (62, 0) in the map
        // m contains {(5, 2), (6, 5), (62, 0)}
```

map

```
map<int, int> m;
m[6] = 5;       // m contains {(6, 5)}
m[5] = 2;       // m contains {(5, 1), (6, 5)}
m[5] = 2;       // m contains {(5, 2), (6, 5)}
m[1000009] = 2; // m contains {(5, 2), (6, 5), (1000009, 1)}

cout << m.begin()->first;    // prints 5
cout << m.begin()->second;   // prints 2

m.erase(1000009); // m contains {(5, 2), (6, 5)}

cout << m[62];               // prints 0
        // creates the entry (62, 0) in the map
        // m contains {(5, 2), (6, 5), (62, 0)}

m[7]++; // creates the entry (7, 0)
        // and then adds 1 to m[7]
        // m is now: {(5, 2), (6, 5), (7, 1), (62, 0)}
```

map

```
map<string, int> m;
m["Hello"] = 5;
m["Salut"] = 200;

cout << m["Who?"];
    // prints 0, and creates the entry ("Who?", 0)

map<int, vector<string> > mvs;
    // many combinations possible

multimap<int, int>
    // can have the same key multiple times

unordered_map<int, int>
    // not sorted by keys, slightly faster
```

complexities set&map

- insert/create new element - O(log(N))
- find element - O(log(N))
- erase element - O(log(N))
  where N is size of the set/map

complexities vector

- insert/create new element at the end - O(1)
- find element - O(N)
- erase element - O(N)

Keep in mind!

There are only a few name of functions that need to be momorised:

- push/push_back/push_front/insert
- pop/pop_back/pop_front
- top/front
- size
- find
- lower_bound/upper_bound

You must not memorize what every function does for a certain data structe! You should look up in the documentation for this.

Greedy

Activity Selection Problem

https://www.infoarena.ro/problema/int

Proposed Problems

Easy/Classical

CF 855A

CF 918B

CF 44A

Medium

int infoarena

Timus - 1112

heapuri infoarena

interclasari - infoarena

CF 903C

CF 1084B

Hard

timus - 1306

CF 1007A

timus 1604