Common data structures: STL

Prof. Rhadamés Carmona

Content

- vector
- set
- multiset
- unordered_set (hash)
- map
- multimap
- list, queue, stack

std::vector

- Dynamic array.
- We can add/remove elements at the end in O*(1) – constant time amortized.
- Vectors usually occupy more space than common arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted.
- The total amount of allocated memory can be queried using capacity() function. It is commonly larger than size().
- The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements.

Vector

```
#include <vector>
std::vector<int> a; // empty vector of integers
std::vector<person> b; // empty vector of person
std::vector<int> c = { 1, 10, 14, -1}; // vector with 4 integers
std::vector<int> d(c); // copy constructor is called…
std::vector<int> e = d; // it first creates an empty vector, then copy

using namespace std;
vector<int> v;          // we do not need to write std::  anymore
```

std::vector

- Most common operations

begin() – Returns an iterator pointing to the first element in the vector

end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector

size() – Returns the number of elements in the vector

capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.

resize(n) – Resizes the container so that it contains ‘n’ elements.

empty() – Returns whether the container is empty.

reserve() – Requests that the vector capacity be at least enough to contain n elements.

std::vector

- Most common operations

data() – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

reference operator [i] – Returns a reference to the element at position ‘i’ in the vector

push_back() – It push the elements into a vector from the back

pop_back() – It is used to pop or remove elements from a vector from the back

insert() – It inserts new elements before the element at the specified position

erase() – It is used to remove elements from a container from the specified position or range.

swap() – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.

clear() – It is used to remove all the elements of the vector container

std::vector

Common use

```
vector<int> v,w;
v.push_back(5);
for (int i=0; i<v.size(); i++)          v.push_back(5);
v.pop.back();
for (vector<int>::iterator i=v.begin(); i!=v.end(); i++) *i = 7;
for (auto i=v.begin(); i!=v.end(); i++) *i = 7;
v.erase(v.begin());
v.swap(w);
#include <algorithm>
sort(v.begin(), v.end()); // sort the array, using integer < operator
```

std::vector

Sorting a vector of person

```
class person {
        private:
                string name;
                string address;
        public:
        bool operator < (const person &p) const {
                return name < p.name;
        }
};
vector<person> v; ….
sort(v.begin(), v.end()); // sort the array, using person < operator
```

std::set

- Sets are containers that store unique elements following a specific order. Sets are usually implemented as red-black trees

  begin: return iterator to beginning O(1)

  end: return iterator to end O(1)

  empty: test whether container is empty O(1)

  size: return container size O(1)

  clear: clears the contents O(n)

  insert: inserts elements O(log n)

  erase: erases elements O(log n)

  find: finds element with specific key O(log n)

std::set

- Sets are containers that store unique elements following a specific order.

```
set<int> s;
s.insert(3);    // s = {3}
s.insert(2);    // s = {2,3}
s.insert(2);    // s = {2,3}
for (auto i=s.begin(); i!=s.end(); i++) // it should print 2 and 3
        cout << *i << endl;
s.erase(3);    // s = {2}
```

std::set

- May we create a set of points? I do not want to duplicate points

```
class point {
public:
        int x,y;
        point(int x, int y) {
                this->x = x; this->y = y;
        }
        bool operator < (const point &p) const {
                return x < p.x || (x==p.x && y<p.y);
        }
};
```

std::set

- May we create a set of points? I do not want to duplicate points

```
set<point> s;
s.insert(point(1,2)); // s = { (1,2) }
s.insert(point(0,2)); // s = { (0,2), (1,2) }
s.insert(point(1,2)); // s = { (0,2), (1,2) }
s.erase(point(1,2)); // s = { (0,2) }
auto i = s.find(point(0,2));
If (i != s.end())
        cout << “we found the point 0,2 in the set” << endl;
```

std::multiset

- Store elements, following a specific order, but elements can have equivalent values
- multiset::count(k): count the number of elements equivalent to k

```
multiset<int> a;
for (int i = 0; i < 3; i++) {
        a.insert(i); a.insert(i);
}
cout << "multi set elements:\n";
for (auto i = a.begin(); i != a.end(); i++)
                cout << " " << *i << endl;   // 0 0 1 1 2 2
cout << "removing 1" << endl;
a.erase(1);
for (auto i = a.begin(); i != a.end(); i++)
                cout << " " << *i << endl;    // 0 0 2 2
```

std::unordered_set

- It is a hash table
- Unique keys
- Insertion is O*(1), amortized constant time
- You know if one element exists or not in O*(1)
- Faster than set, but … unsorted.
- Useful if you want to have unique elements, or if you want to know if one element exists or not.
- Methods as std::set: insert, begin, end, clear, size, find, …

std::unordered_set

```
unordered_set<int> a;
a.insert(3);
a.insert(80);
a.insert(8);
a.insert(17);
a.insert(8);
cout << "unordered set elements:\n";
for (auto i = a.begin(); i != a.end(); i++)
        cout << " " << *i << endl;  // {3,8,80,17}
auto pos = a.find(80);
if (pos != a.end())
        cout << "we found " << *pos  << endl;
```

std::unordered_multiset

- Allow duplicated keys

std::map

- Collection of key-values pairs (associative containe)
- Keys are unique

begin() – Returns an iterator to the first element in the map
end() – Returns an iterator to the theoretical element that follows last element in the map
size() – Returns the number of elements in the map
max_size() – Returns the maximum number of elements that the map can hold
empty() – Returns whether the map is empty
pair insert(keyvalue, mapvalue) – Adds a new element to the map
erase(iterator position) – Removes the element at the position pointed by the iterator
erase(keyvalue)– Removes the key value from the map
clear() – Removes all the elements from the map

T& operator[](const Key& key): Returns a reference to the value that is mapped to a key, performing an insertion if such key does not already exist.

std::map

```
map <int, string> m;
m.insert(pair<int, string>(3, “hi”));
m[5] = “hello”;
m[5] = “updating the value of a key”;
auto i = m.find(5);
if (i == m.end())
        cout << “ 5 is not found” << endl;
else
        cout << “we found 5” << endl;
```

std::multimap

- Collection of key-values pairs
- Allow duplicated keys
- Same key can be related to different values

May we use map<key, vector<value>> as a possible implementation of multimap?

Notice that value can be a container

Problems to solve with STL

- 14944 Santa1 - Reindeer Games https://www.spoj.com/problems/SANTA1/
- HOMO - Homo or Hetero https://www.spoj.com/problems/HOMO/
- TSORT - Turbo Sort https://www.spoj.com/problems/TSORT/
- BYTESE2 - The Great Ball https://www.spoj.com/problems/BYTESE2/
- SBANK - Sorting Bank Accounts https://www.spoj.com/problems/SBANK/
- 6345 AMR12G - The Glittering Caves of Aglarond https://www.spoj.com/problems/AMR12G/