STL

la biblioteca estándar de C++

Estructuras de datos

Estructuras de datos

Una estructura de datos es un objeto que contiene y organiza datos, con el fin de soportar ciertas operaciones de forma eficiente. Nos interesa saber la complejidad de cada operación para ayudarnos a decidir cuál estructura usar en cada situación.

Aparte, en un problema dado, podemos darnos cuenta que hay varias estructuras que soportan las operaciones necesarias con complejidad igual o similar. En estos casos, tratamos de elegir la que sea más rápida en términos absolutos (lo podríamos medir con un cronómetro). Al valor que cuantifica esta diferencia lo llamamos “factor constante”.

Arrays dinámicos - vector y string

A diferencia de los arrays que conocen de C, es posible asignar y modificar la longitud de un array dinámico en tiempo de ejecución. En C++ se llama vector. Vector nos permite agregar y eliminar elementos del final en O(1).

```
vector<int> v;    // v es un array vacío
v.resize(5, 7);   // v = {7, 7, 7, 7, 7}
v[4] = 15;        // v = {7, 7, 7, 7, 15}
v.push_back(42);  // v = {7, 7, 7, 7, 15, 42}
v.pop_back();     // v = {7, 7, 7, 7, 15}
```

String es muy similar a vector<char>, pero también puede iniciarse con un literal, y leerse e imprimirse por consola.

```
string s = “hola”;  // inicio s con la cadena ‘hola’
cout << s;          // imprime ‘hola’
```

Conjuntos - set y unordered_set

Un conjunto tiene dos operaciones: guardar valores, borrar valores y chequear si un valor es parte del conjunto. (esto sirve, por ejemplo, si tenemos dos formas de generar valores, y queremos ver cuáles valores son generados por ambas). En C++ tenemos set y unordered_set, que soportan ambas operaciones.

```
set<int> s;
s.insert(37);
s.insert(42);
cout << s.count(42) << '\n'; // imprime 1
s.erase(42);
cout << s.count(42) << '\n'; // imprime 0
cout << s.count(37) << '\n'; // imprime 1
```

Aparte, set mantiene los elementos ordenados de menor a mayor.

Arrays asociativos - map y unordered_map

De la misma forma que usando arrays asociamos un índice (un número entero) con el valor que se encuentra en la posición correspondiente, usando arrays asociativos podemos asociar otros tipos de datos, como cadenas. En C++, se llaman map y unordered_map.

Adicionalmente, map nos permite iterar por los índices de menor a mayor.

```
map<string, int> edades;
edades[“pepe”] = 42;
edades[“schujman”] = 2000;

// imprime 42
cout << edades[“pepe”] << ‘\n’;
```

Bolsas - queue, stack y priority_queue

Una bolsa nos deja insertar valores y luego extraerlos en un orden determinado. Este orden depende del tipo de bolsa que usemos.

queue: se extrae el valor más viejo, funciona como la cola del super

stack: se extrae el valor más nuevo, funciona como una pila de platos

priority_queue: se extrae el valor más grande

Observación: se pueden implementar todas las operaciones de priority_queue usando set, pero con un factor constante bastante peor.

Bolsas - queue, stack y priority_queue

Aunque son similares conceptualmente, sus las funciones que implementan sus operaciones tienen distintos nombres

estructura	queue<int> q;	stack<int> s;	priority_queue<int> pq;
insertar	q.push(x)	s.push(x)	pq.push(x)
inspeccionar	q.front()	s.top()	pq.top()
extraer	q.pop()	s.pop()	pq.pop()

Bolsas - queue, stack y priority_queue

```
queue<int> q;

q.push(1);
q.push(3);
q.push(2);

// imprime 1
cout << q.front() << ‘\n’;

q.pop();

// imprime 3
cout << q.front() << ‘\n’;
```

```
stack<int> s;

s.push(1);
s.push(3);
s.push(2);

// imprime 2
cout << s.top() << ‘\n’;

s.pop();

// imprime 3
cout << s.top() << ‘\n’;
```

```
priority_queue<int> p;

p.push(1);
p.push(3);
p.push(2);

// imprime 3
cout << p.top() << ‘\n’;

p.pop();

// imprime 2
cout << p.top() << ‘\n’;
```

Extra - for en rango

Algunas de las estructuras que vimos nos permiten hacer un for que pasa por todos sus elementos.

```
set<int> s = {4, 8, 5, 0, 9};
// imprime ‘0 4 5 8 9’
for (int x : s) {
     cout << x << ‘ ‘;
}

map<string, int> m;
m[“BBB”] = 42;
m[“AAA”] = 37;
// imprime ‘AAA 37 -- BBB 42 --’
for (auto p : m) {
     cout << p.first << ‘ ‘ << p.second << “ -- “;
}
```

Algoritmos

Iteradores

Los iteradores son valores que representan una posición dentro de una estructura de datos. Para la mayoría de las que vimos, podemos obtener el principio y el final usando .begin() y .end()

Dado un iterador podemos obtener el valor que le corresponde con un asterisco y hacer que apunte una posición más adelante usando el operador ++.

El iterador .end() apunta a la posición justo después del último elemento.

```
vector<int> v = {16, 42, 37};
auto it = v.begin();
cout << *it << '\n'; // imprime 16
++it;
cout << *it << '\n'; // imprime 42
```

Ordenamiento - sort

El más común de los algoritmos es sort, que toma dos iteradores y ordena el rango de valores que se encuentra entre ellos de menor a mayor.

```
vector<int> v = { 7, 4, 9, 0 };
sort(v.begin(), v.end());
// ahora queda  v = { 0, 4, 7, 9 }
```

Búsqueda binaria - lower_bound y upper_bound

Dado un rango ordenado, upper_bound encuentra el primer valor que es mayor al que se busca

Dado un rango ordenado, lower_bound encuentra la primera aparición del valor que se busca. Si no existe, encuentra lo mismo que upper_bound.

Internamente usan búsqueda binaria!

```
auto v = {1, 2, 3, 4, 5, 6, 7};
auto it1 = lower_bound(v.begin(), v.end(), 4);  // {1, 2, 3, 4, 5, 6, 7}
auto it2 = upper_bound(v.begin(), v.end(), 4);  // {1, 2, 3, 4, 5, 6, 7}
auto it3 = upper_bound(v.begin(), v.end(), 7);  // {1, 2, 3, 4, 5, 6, 7} .
```

Permutaciones - reverse, rotate y partition

reverse da vuelta un rango

rotate intercambia un prefijo de un rango con el sufijo correspondiente

partitio toma un rango y un predicado, y pone todos los valores que dan true al principio del array. Devuelve un iterador a después del último elemento que dio true.

```
vector<int> v = {6, 5, 4, 3, 2, 1};
reverse(v.begin(), v.end());                    // v queda        {1, 2, 3, 4, 5, 6}
rotate(v.begin(), v.begin() + 2, v.end());      // v queda        {3, 4, 5, 6, 1, 2}
auto it = partition(v.begin(), v.end(), es_par); // v puede quedar {4, 6, 2, 3, 5, 1}
```

Varios - fill, unique, remove_if

fill rellena un rango con un valor.

remove_if es similar a partition. pero pone los que dan false al principio, y el resto queda indeterminado.

unique es similar a remove_if pero condensa los grupos de elementos iguales consecutivos.

```
vector<int> v = {1, 2, 2, 3, 5, 5, 2, 2, 2};
auto it1 = unique(v.begin(), v.end());          // v = {1, 2, 3, 5, 2, ?, ?, ?, ?}
auto it2 = remove_if(v.begin(), it1, es_par);   // v = {1, 3, ?, ?, ?, ?, ?, ?, ?}
fill(v.begin(), it2, 0);                        // v = {0, 0, ?, ?, ?, ?, ?, ?, ?}
```

Problemas

https://cses.fi/problemset/task/1621

https://codeforces.com/problemset/problem/4/C

https://codeforces.com/problemset/problem/519/B

https://codeforces.com/problemset/problem/368/B

Bibliografía

Apunte de estructuras de Román: https://raw.githubusercontent.com/SebastianMestre/taller-oia/master/Apuntes/EstructurasEnCpp.pdf

cpp reference: https://en.cppreference.com/w/cpp/algorithm
https://en.cppreference.com/w/cpp/container

oia-wiki: http://wiki.oia.unsam.edu.ar/cpp-avanzado/stl