Estructuras de Datos

Román Castellarin

Estructuras

- Array
- Vector
- Stack
- Queue
- Priority Queue
- Deque
- Linked List
- Set
- Map
- Unordered Set
- Unordered Map

Introducción

Las **estructuras de datos** nos presentan formas de organizar, almacenar y consultar datos.

Cada una de ellas tiene sus ventajas y desventajas.

Por lo tanto, al diseñar un algoritmo es importante elegir aquélla que permita realizar **eficientemente** las operaciones en las cuales estamos interesados.

Biblioteca Estándar C++

Existe un puñado de **estructuras básicas** que todo programador debe conocer y manejar.

Por suerte, la **biblioteca estándar** de C++ trae en ella muchas de estas estructuras ya implementadas.

Para la *programación competitiva*, por lo general es sólo necesario saber lo suficiente para **seleccionar** y **utilizar** la estructura adecuada.

Estructuras contiguas

Arrays

Tan básico que muchas veces ni se lo considera una estructura de datos.
Tienen un tamaño fijo y acceso aleatorio constante.

A menos que un valor sea explicitado, son default-inicializados.

```
float A[100] = {};    // 100 floats inicializados a 0.0
int B[10][50];        // 500 ints inicializados al azar
```

Vector

Es una suerte de array con tamaño dinámico. `#include<vector>`

```
vector<int> A; // vector de ints vacio
vector<int> A(5); // vector de 5 ints inicializados a 0 (default)
vector<int> A(5, 7); // vector de 5 ints inicializados a 7
```

Vector

Algunas de sus operaciones son:

`.size()`	devuelve el tamaño (cant. de elementos)
`.empty()`	devuelve true sii está vacío
`.clear()`	vacía la estructura

Estas son operaciones comunes a la mayoría de las estructuras de la biblioteca estándar de C++.

Vector

Algunas de las operaciones propias de vector son:

`[i]`	accede al elemento i-ésimo	O(1)
`.push_back(x)`	inserta x al final	O(1) amort.
`.pop_back()`	elimina el último elemento	O(1) amort.
`.resize(n)`	redimensiona, si n > size() entonces rellena con x	
`.resize(n, x)`	o el valor por defecto	O(| size() - n |) amort.

colecciones tipo bolsa

Stack

Es una pila (LIFO). `#include <stack>`

```
stack<char> S; // pila de chars
S.push( ’A’ );
S.push( ’B’ );
S.push( ’C’ );
cout << S.size() << ’ ’ << S.top() << endl; // 3 C
S.pop();
cout << S.size() << ’ ’ << S.top() << endl ; // 2 B
```

Stack

Operaciones

`.push(x)`	coloca x en el tope	O(1)
`.top()`	retorna el tope	O(1)
`.pop()`	desapila el tope	O(1)

Las pilas no son muy utilizadas ya que pueden ser reemplazadas por un vector, o por recursión en algunos algoritmos.

Queue

Es una cola (FIFO). `#include <queue>`

```
queue<char> Q; // cola de chars
Q.push( ’A’ );
Q.push( ’B’ );
Q.push( ’C’ );
cout << Q.size() << ’ ’ << Q.front() << endl; // 3 A
Q.pop();
cout << Q.size() << ’ ’ << Q.front() << endl ; // 2 B
```

Queue

Operaciones

`.push(x)`	coloca x al final	O(1)
`.front()`	retorna el primer elemento	O(1)
`.pop()`	elimina el primer elemento	O(1)

Las colas son **extremadamente** comunes

Priority Queue

Es una bolsa donde se extrae el elemento de mayor prioridad.

La prioridad está dada por `operator<`.

```
#include <queue>

priority_queue<int> Q; // cola de prioridad de ints
Q.push( 50 );
Q.push( 100 );
Q.push( 20 );
cout << Q.size() << ' ' << Q.top() << endl; // 3 100
Q.pop();
cout << Q.size() << ' ' << Q.top() << endl ; // 2 50
```

Priority Queue

Operaciones

`.push(x)`	coloca x adentro de la bolsa	O(log n)
`.top()`	retorna el mayor elemento	O(1)
`.pop()`	elimina el mayor elemento	O(log n)

Las colas de prioridad se usan en muchos algoritmos comunes como Dijkstra, Huffman, sweep line, etc..

Algunas estructuras lineales

Deque

Pronunciado "deck", es similar al vector pero se puede insertar y borrar eficientemente en ambos extremos, a costa de que sus elementos no están contiguos en memoria. `#include <deque>`

```
deque<int> Q ; // cola doble de ints
Q.push_back ( 50 );
Q.push_back ( 100 );
Q.push_front ( 20 );
cout << Q.size () << ’ ’ << Q.front() << ’ ’ << Q[1] << endl; // 3 20 50
Q.pop_front();
cout << Q.size () << ’ ’ << Q.back () << ’ ’ << Q[1] << endl; // 2 100 100
```

Deque

Algunas sus operaciones son:

`[i]`	accede al elemento i-ésimo	O(1)
`.push_back(x)`	inserta x al final	O(1)
`.push_front(x)`	inserta x al inicio	O(1)
`.pop_back()`	elimina el último elemento	O(1)
`.pop_front()`	elimina el primer elemento	O(1)

A pesar de que esta estructura parece ganarle a todas las anteriores excepto priority_queue, tiene constante computacional muy alta

Linked List

Listas doblemente enlazadas.

```
#include <list>

list<int> L; // lista de ints
L.push_front( 40 );
L.push_back( 50 );
auto p = L.begin(); // begin() y end() son O(1)
L.push_front( 20 ); // no invalida p
L.push_front( 10 ); // no invalida p
L.insert( p , 30 ); // O(1) porque sabemos la posición exacta
for( auto &x : L )
     cout << x << ' '; // 10 20 30 40 50
```

Linked List

Algunas sus operaciones son:

`.push_back(x)`	inserta x al final	O(1)
`.push_front(x)`	inserta x al inicio	O(1)
`.pop_back()`	elimina el último elemento	O(1)
`.pop_front()`	elimina el primer elemento	O(1)

Para iterar la lista, podemos obtener punteros bidireccionales con begin() y end()

Linked List

Además,...

`.insert(p, x)`	inserta x antes de p	O(1)
`.erase(p)`	elimina el elemento en p	O(1)
`.merge(`L_2`)`	mergea L_2 en L	O(N + N_2)
`.sort()`	ordena la lista	O(N log N)

Esta estructura no se usa casi nunca a menos que se requiera insertar elementos *en el medio* dada la posición.

Estructuras no lineales

Set

Se lo puede pensar como una colección **ordenada** de elementos **únicos**.

A pesar de su nombre, no soporta eficientemente las operaciones comunes de conjuntos (unión, intersección, diferencia, etc...).

Al igual que las colas de prioridad, necesitan que exista un orden estricto definido entre sus elementos (esto no es verdad en matemática).

En un set no puede haber dos elementos equivalentes
(a y b se consideran equivalentes cuando $a \not< b$ y $b \not< a$).

Set

```
#include <set>

set<int> S; // conjunto de ints
S.insert( 4 );
S.insert( 8 );
cout << S.size() << endl; // 2
S.insert( 8 ); // no se inserta porque 8 es equivalente a 8
cout << S.size() << endl; // 2 todavia
cout << *S.lower_bound(4) << ’ ’<< *S.upper_bound(4) << endl; // 4 8
```

Set

Algunas sus operaciones son:

`.insert(x)`	inserta x	O(log n)
`.erase(x)`	elimina x	O(log n)
`.count(x)`	verifica si x está presente en la estructura	O(log n)
`.lower_bound(x)`	retorna un puntero al primer elemento mayor o igual a x	O(log n)
`.upper_bound(x)`	retorna un puntero al primer elemento estrict. mayor a x	O(log n)

Para iterar el set, podemos obtener punteros bidireccionales con begin() y end(). Recordemos que el set está ordenado, así que el mín elemento de S es *S.begin() y el último es *S.rbegin()

Map

Representan diccionarios.

Pueden ser pensados como sets, en los cuales a cada uno de sus elementos (key) le es asociado un valor (value).

Notemos que los tipos de la claves y los valores no necesariamente deben coincidir, i.e, podemos asociar números a strings.

Son como arrays donde las claves pueden ser cualquier tipo ordenado.

Map

Su operación importante es:

[x]	el valor asociado a x (si no existe lo crea)	O(log n)

Todas las operaciones de set valen (como si se realizasen sobre las claves)

Map

```
#include <map>

...

map<int,char> M; // mapeo de int → char
M[5] = ’A’; // asociamos 5 con ’A’
cout << M.size() << endl; // 1
if( M[7] != ’R’ ) // creamos sin querer una nuevo par clave - valor !
    cout << "ESTO ESTA TAN MAL !" << endl;
cout << M.size() << endl; // 2 ( oops !)
if( M.count(4) and M[4] == ’T’ ) // esto esta bien
    cout << "Pregunto si existe, luego si es lo que busco" << endl ;
```

Tablas de hash

Unordered Set

Se lo puede pensar como un set **no ordenado.**

En lugar de requerir que sus elementos tengan definido `operator<`, se requiere que tengan definido `hash<T>` y `operator==`.

Esto permite operaciones de búsqueda, inserción y borrado en O(1), a costa de perder las operaciones que dependen del orden como lower_bound, upper_bound, etc...

Unordered Map

Se lo puede pensar como un map cuyas claves **no están ordenadas.**

Unordered map es a map lo que unordered set es a set.

Si conocemos la cantidad de objetos que usaremos podemos aprovechar para reservar espacio a priori (.reserve, .max_load_factor).