Estructuras de Datos I

Román Castellarin

2016

1. Introducción

En este apunte se van a **introducir** un puñado de **estructuras de datos** básicas que todo programador debe conocer y manejar. Las estructuras de datos representan formas de organizar, almacenar y consultar datos. Cada una tiene sus ventajas y sus desventajas. Por lo tanto, al diseñar un algoritmo, es importante elegir aquélla que permita realizar **eficientemente** las operaciones en las cuales estamos interesados.

Adicionalmente, vamos a analizar la **biblioteca estándar de C++ (STL)**, donde estas estructuras ya están implementadas, para no tener que reinventar la rueda.

Notemos que para la *programación competitiva*, por lo general sólo es necesario saber lo suficiente para poder **seleccionar** y **utilizar** la estructura correcta; aunque comprender el funcionamiento interno es importante para poder implementar estructuras de datos que no vienen prearmadas, o realizar modificaciones a las ya existentes.

1.1. Aclaraciones previas:

- Este apunte supone un conocimento mínimo previo sobre las funciones base del lenguaje, en particular el concepto de bibliotecas, classes/structs, funciones miembro (también llamadas métodos), punteros, y sobrecarga de operadores.
- Ciertas estructuras y métodos están disponibles sólo a partir de la versión 11^{ra} de C++. Compilar con `-std=c++11`. El juez online de OIA Poli e IOI hacen esto automáticamente.
- **Siempre consultar la documentación, hasta aprendérsela de memoria**: (*www.cplusplus.com/reference/stl*).

2. Estructuras de Datos

2.1. Array

Los arrays son una manera tan elemental y natural de organizar datos, que muchas veces ni se lo considera una estructura de datos. Se puede pensar como una matriz de datos almacenados secuencialmente, cuyos elementos se acceden mediante índices. La cantidad de dimensiones de un array es la cantidad de índices que hacen falta para indetificar un elemento, por lo general, no necesitaremos arrays con más de 3 dimensiones.

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

Importante:

- Los elementos de un array pueden ser accedidos y modificados en tiempo constante.

- Una vez declarado, no se puede cambiar el tamaño de un array.
- Si no se lo inicializa, se rellena con el constructor por defecto (generalmente basura).

2.2. Vector (`#include<vector>`) ¡importante!

Los vectores no son nada más ni nada menos que arrays dinámicos. Se los prefiere cuando su tamaño es desconocido al momento de compilar, y para ahorrar memoria. Sus elementos se acceden con `operator[⋆]` en $O(1)$ como en los arrays.

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

Importante:

- Insertar o borrar un elemento por el final con `.push_back(⋆)` y `.pop_back()` resp. es $O(1)$, efectivamente modificando el tamaño del vector en 1.
- Su tamaño se puede cambiar con `.resize(⋆)` en $O(n)$.

Bolsas

Las bolsas son estructuras de datos que nos permiten hacer tres operaciones fundamentales:

I `insertar(⋆)`: agrega el elemento $\star$ a la bolsa.

II `consultar()`: devuelve un elemento particular de la bolsa, sin eliminarlo de ella.

III `borrar()`: elimina el mismo elemento al que se refiere consultar.

Veremos ahora tres tipos de bolsas muy importantes en la programación:

2.3. Stack (pila) (`#include<stack>`)

Las pilas son **bolsas** cuyos elementos podemos imaginar que se almacenan uno arriba de otro, apilados. Se las suele denominar **LIFO** (el último elem. en entrar es el primero en salir). Sus tres operaciones fundamentales se llaman:

I `.push(⋆)`: coloca el elemento $\star$ en el tope de la pila.

II `.top()`: devuelve el valor del elemento en el tope de la pila.

III `.pop()`: elimina el elemento del tope de la pila.

Todas sus operaciones son $O(1)$, aunque pueden ser emuladas igual de eficientemente con un `vector`, utilizando `.push_back(⋆)`, `.back()`, y `.pop_back()` respectivamente.

```
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
```

$\Diamond$: Podés ver una visualización de esta estructura en: `visualgo.net/list` , pestaña stack.

2.4. Queue (cola) (`#include<queue>`) ¡importante!

Las colas son **bolsas** cuyos elementos podemos pensar que ingresan por el el final y salen por delante. Se las suele denominar **FIFO** (primero en entrar, primero en salir).

I `.push(⋆)`: coloca el elemento ⋆ al final de la cola.

II `.front()`: devuelve el valor del primer elemento de la cola (el de más adelante).

III `.pop()`: elimina el primer elemento de la cola.

Todas sus operaciones son $O(1)$ y por eso no pueden ser emuladas eficientemente con un `vector` ya que en este último, `.pop_front()` es $O(n)$.

```
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
```

♢ : Podés ver una visualización de esta estructura en: `visualgo.net/list` , pestaña queue.

2.5. Priority Queue (cola de prioridad o *heap*) (`#include<queue>`) ¡importante!

Las colas de prioridad son **bolsas** que nada tienen en común con las colas comunes y corrientes que acabamos de ver, de hecho, se parecen más a las pilas. Las colas de prioridad necesitan que exista un orden estricto definido entre sus elementos (`operator<`), así, el elemento a consultar es siempre el más grande definido por dicho orden.

I `.push(⋆)`: coloca el elemento ⋆ adentro de la cola de prioridad.

II `.top()`: devuelve el elemento más grande dentro de la cola de prioridad.

III `.pop()`: elimina el elemento más grande dentro de la cola de prioridad.

Todas estas operaciones son $O(log\ n)$.

```
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
```

♢ : Podés ver una visualización de esta estructura en: `visualgo.net/heap`.

Otras estructuras lineales...

2.6. Deque (cola doble) (`#include<deque>`)

Una cola doblemente terminada, o cola de doble punta, es una estructura de datos que permite realizar las mismas operaciones que una cola convencional, pero en **ambos** extremos. Adicionalmente, permite el acceso a sus elementos en $O(1)$ mediante `operator[⋆]`. Sus métodos son:

I `.push_back(⋆)`: coloca el elemento ⋆ al final de la cola.

II `.push_front(⋆)`: coloca $\star$ al inicio de la cola.

III `.front()`: devuelve el valor del primer elemento de la cola (el de más adelante).

IV `.back()`: devuelve el valor del último elemento de la cola (el de más atrás).

V `.pop_front()`: elimina el primer elemento de la cola.

VI `.pop_back()`: elimina el último elemento de la cola.

Parece que nos hemos encontrado con una estructura que domina a todas las demás, a excepción de la *cola de prioridad*. Sin embargo, las deques se usan poco, ya que a pesar de que sus operaciones son todas $O(1)$, su **constante computacional** es bastante **alta**.

```
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
```

♢ : Podés ver una visualización de esta estructura en: `visualgo.net/list` , pestaña deque.

2.7. Linked List (lista enlazada) (`#include<list>`)

Las listas enlazadas almacenan sus elementos de manera no contigua, enlazándolos, es decir que por cada elemento guardan una referencia al anterior y al posterior (si existen). Esto inmediatamente nos provee de una operación que no habíamos visto antes: se puede insertar eficientemente un elemento en cualquier posición **conocida** de la lista en $O(1)$, ya que sólo hay que modificar las referencias de dos elementos y no modificar toda la estructura.

¿A qué nos referimos con conocida?, como los elementos no se guardan contiguamente, no es posible fácilmente calcular la dirección en memoria del n-ésimo elemento, sino que se debe empezar del principio, y avanzar n posiciones, es decir, que acceder a un elemento cuesta $O(n)$, razón por la cual **casi nunca** se emplean en programación competitiva, y se prefiere `vector` en su lugar.

```
list<int> L;              // lista de ints
L.push_front( 40 );
L.push_back( 50 );
list<int>::iterator p = L.begin();      // nota 1
L.push_front( 20 );                     // nota 2
L.push_front( 10 );                     // nota 2
L.insert( p, 30 );        // O(1) porque sabemos la posicion exacta
for(p = L.begin(); p != L.end(); ++p)   // ver apendice de iteradores
        cout<<*p<<' ';    // 10 20 30 40 50
```

♢ : Podés ver una visualización de esta estructura en: `visualgo.net/list` , pestaña doubly-linked list. **nota 1:** Hallar la posición del primer o último elemento es $O(1)$.

nota 2: La documentación dice "Iterator validity: no changes", que significa que esta operación **no invalida** las posiciones previamente guardadas, (en nuestro caso, p).

2.8. Estructuras asociativas

2.9. Set (conjunto) (`#include<set>`) ¡importante!

Los conjuntos de elementos, al igual que las colas de prioridad, necesitan que exista un orden estricto definido entre sus elementos. En un set no puede haber dos elementos equivalentes (a y b se consideran equivalentes cuando $a \not< b$ y $b \not< a$). Se lo puede pensar como una colección ordenada de elementos únicos. Algunas operaciones importantes:

I `.insert(⋆)`: inserta $\star$ adentro del set si y solo no existía un elemento equivalente dentro.

II `.erase(⋆)`: elimina el elemento equivalente a $\star$ si lo había.

III `.count(⋆)`: devuelve la cantidad de elementos equivalentes a $\star$ (0 ó 1 por operación I).

IV `.lower_bound(⋆)`: devuelve un puntero al primer elemento que no precede a $\star$.

V `.upper_bound(⋆)`: devuelve un puntero al primer elemento que sucede a $\star$.

Todas estas operaciones son $O(log\ n)$.

```
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 equiv. a 8
cout<<S.size()<<endl;       // 2 todavia
cout<< *S.lower_bound(4) <<' '<< *S.upper_bound(4) <<endl; // 4 8
```

$\Diamond$: Podés ver una visualización de esta estructura en: `visualgo.net/bst`.

2.10. Map (mapeo) (`#include<map>`) ¡importante!

Los mapeos pueden ser pensados como conjuntos de elementos, en los cuales a cada uno de sus elementos **(key)** le asocio un valor **(value)**, notemos que los tipos de la claves y los valores no necesariamente deben coincidir, es decir, podemos asociar números a strings, o peras a olmos.

Asímismo, los conjuntos de elementos pueden ser pensados como mapeos donde la clave y el valor son el mismo, por esa razón, las mismas restricciones de orden se mantienen para las claves (no es necesario para los valores).

I `.[⋆]`: accede al elemento con clave $\star$, **o lo crea si no existe** con el valor por defecto.

II Las operaciones de `set` también valen, es como si se realizaran sobre las claves.

De nuevo, todas estas operaciones son $O(log\ n)$.

```
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;
```

2.11. ... ¿más estructuras?

A pesar de haber hecho un muy breve resumen de diez estructuras de datos básicas, aún hay muchísimas más por ver. No hay nada por qué asustarse, prontamente van a estar manejando las estructuras vistas en este apunte de manera totalmente natural. Justo a tiempo para el apunte de Estructuras de Datos II, donde deberemos implementar a mano nuestras propias estructuras ya que la biblioteca estándar no las trae incorporadas.

3. Punteros e Iteradores *(repaso)*

Recordemos qué era un puntero: un dispositivo que sirve para apuntar.
En C++ un **puntero** es simplemente una variable que almacena la dirección de memoria de un objeto. Accedemos al contenido de un puntero mediante `operator*` y a la dirección de una variable mediante `operator&`. Así, si `p` es un puntero que apunta al objeto `r`, entonces `r` es `*p`, mientras que `p` es `&r`. En la mayoría de los punteros, las siguientes operaciones aritméticas valen:

- p puntero, n entero, entonces (p+n) es puntero y apunta n elementos adelante de p.
- p puntero, n entero, entonces (p-n) es puntero y apunta n elementos detrás de p.
- p, q puntero, (p-q) es entero y representa la cantidad de elementos entre q y p.

Venimos trabajando con punteros sin saberlo, ya que el nombre de un array representa un puntero a su primer elemento. Adicionalmente. podemos declarar un puntero como cualquier variable, anteponiéndole un asterisco:

```
int n = 5, *p;
char A[] = {'X', 'Y', 'Z'};
p = &n;                              // p apunta a n
cout<<*(A+2)<<' '<<*p<<endl;         // Z 5
*p = 9;
cout<<n<<endl;                       // 9
```

Con la aparición de las estructuras de datos más complejas, surgió la necesidad de contar con punteros que también funcionen en estas nuevas estructuras. Por ejemplo, una lista doblemente enlazada no almacena sus elementos contiguamente, así que "avanzar n elementos" no se resuelve con una simple suma. Sin embargo, la idea de los iteradores es servir de interfaz para que no importe cómo se resuelvan internamente las operaciones.

Finalmente, un **iterador** de una estructura `C` es un objeto `p` que me permite iterar los elementos de `C`, es decir, soporta al menos las siguientes dos operaciones:

- `*p` : acceder al elemento apuntado.
- `p++` : apuntar al siguiente elemento.

No todas las estructuras de este apunte son iterables, pero por lo general si `T` es el tipo de una estructura de la STL, entonces `T::iterator` es el tipo del iterador asociado. Como esto se puede tornar medio denso de escribir, C++11 permite escribir `auto` como tipo cuando éste se puede deducir a partir de su valor inicial. Adicionalmente, las operaciones pueden tener distinta complejidad.

```
set<int> S;
S.insert(4);
set<int>::iterator q = S.begin();
S.insert(2);    // iterator validity: no changes => q valido
```

```
S.insert(3);
S.insert(5);
S.insert(1);
for( auto p = S.begin(); p != S.end(); ++p )
    cout<< *p <<' ';    // 1 2 3 4 5
cout<< *q <<endl;   // 4
```

Por lo general, las estructuras iterables de la STL cuentan con `.begin()` y `.end()`, que son iteradores al inicio (primer elemento) y al final (lo que le sucede al último elemento). Así, todos los elementos de `C` son los apuntados por el rango `[C.begin(); C.end())` cerrado-abierto.

4. Operaciones y complejidades

En la mayoría de las estructuras en las cuales tiene sentido, existen los siguientes métodos:

- `.size()` : devuelve el tamaño de la estructura (cant. de elementos), es $O(1)$.
- `.clear()` : vacía la estructura, (ahora `.size()` devolverá 0), es $O(n)$.

A continuación una tabla con las complejidades de las operaciones más comunes:

Estructura	Push (front)	Push (back)	Insert	Pop (front)	Pop (back)	Erase
Array	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
Vector	$\times$	$O(1)$	$O(n)$	$\times$	$O(1)$	$O(n)$
Stack	$O(1)$	$\times$	$\times$	$O(1)$	$\times$	$\times$
Queue	$\times$	$O(1)$	$\times$	$O(1)$	$\times$	$\times$
Priority Queue	$\times$	$\times$	$O(log\ n)$	$\times$	$\times$	$O(log\ n)$
Deque	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Linked List	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Set	$\times$	$\times$	$O(log\ n)^{*1}$	$\times$	$\times$	$O(log\ n)^{*2}$
Map	$\times$	$\times$	$O(log\ n)^{*1}$	$\times$	$\times$	$O(log\ n)^{*2}$

Estructura	Acceder i-ésimo elemento	Find	LowerBound	UpperBound
Array / Vector / Deque	$O(1)$	$O(n)^{*3}$	$O(n)^{*3}$	$O(n)^{*3}$
Linked List	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Set	$O(n)$	$O(log\ n)$	$O(log\ n)$	$O(log\ n)$
Map	$O(n)$	$O(log\ n)$	$O(log\ n)$	$O(log\ n)$

- 1: $O(1)$ si se indica una pista de la posición.
- 2: $O(1)$ si se indica la posición.
- 3: Se puede en $O(log\ n)$ si está ordenado, mediante búsqueda binaria.

5. Ejercicios

- PROBAR TODAS LAS ESTRUCTURAS DE ESTE APUNTE, Y SUS FUNCIONES.
- Investigar sobre el restante de las funciones disponibles para cada estructura, (buscar en la documentación!).

- Las estructuras de datos traen consigo nuevos algoritmos para manipular los datos. Investigar sobre las funciones provistas en la biblioteca `algorithm`. ¿Es lo mismo usar la función `.count(*)` de `algorithm` que la especializada en `set`? ¿Por qué?
  (consultar `www.cplusplus.com/reference/algorithm`).
- (Avanzado) Investigar sobre una estructura de datos avanzada que está presente en la STL, llamada **tabla hash** e implementada como `unordered_map`. ¿Qué propiedad deben cumplir las claves? ¿Qué complejidad (de tiempo) tienen sus operaciones?
- Volver a leer el apunte y asegurarse que no te olvidaste de leer sobre alguna estructura o función.
- Próximamente ejercicios de verdad.

Referencias

[1] VisuAlgo (visualgo.net), *visualising data structures and algorithms through animation*, Dr.Steven Halim, National University of Singapore