Manual 2 - 2do Torneo de Programación Competitiva

Lions R.C.

Junio 2019

Contents

1 Estructuras de datos sencillos **2**
- 1.1 Vectores . . . 2
- 1.2 Pares . . . 5
- 1.3 Mapas . . . 6
- 1.4 Sets . . . 8

2 Complejidad de tiempo **8**

3 Complejidad de memoria **8**

4 Busquedas **8**
- 4.1 Busqueda lineal . . . 8
- 4.2 Busqueda binaria . . . 8
- 4.3 Busqueda alfabetica . . . 8

5 Ordenamientos **8**
- 5.1 Ordenamiento de selección . . . 8
- 5.2 Ordenamiento de inserción . . . 8
- 5.3 Ordenamiento por mezcla . . . 8

6 Matemáticas en C++ **8**
- 6.1 Libreria cmath . . . 8
- 6.2 Desventajas . . . 8

1 Estructuras de datos sencillos

Muchas veces, nos encontramos en medio de un problema que ocupa el uso de algo mas flexible que un arreglo. Si queremos borrar datos de un arreglo, tendriamos que desplazar todos los demás datos que estan enfrente hacia atras, y nuestros resultados se vuelven mas ineficientes o más complicados.

Una solución es utilizar otros estructuras de datos, que almacenan y manejan datos de distintas maneras. En esta sección, solo se explicaran algunas de las muchas estructuras de datos, y la siguiente semana se darán a conocer los demás.

1.1 Vectores

Un vector no es nada más que un arreglo dinámico. Esto significa que el vector no tiene un tamaño fijo y puedes agregar y quitar elementos sin problema.

Para crear un vector, es necesario agregar una libreria especifico a esta estructura, llamado **vector**.

Para incluir esta libreria, se debe de escribir **#include <vector>** en las primeras lineas de tu programa.

Listing 1: Vectores

```
#include <iostream>
#include <vector>

using namespace std;

int main() {

}
```

Como un vector se parece a un arreglo, debes definir el tipo de dato que se almacenará a la hora de declararlo. Se puede declarar el vector con el sintaxis **vector<dato> nombre;** Como se puede observar, el vector no requiere que le des un tamaño predeterminado.

Listing 2: Vectores

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> enteros;
}
```

Aqui se ha declarado un vector de enteros, y para agregarle elementos a este vector, se debe escribir el nombre del vector seguido por un punto y la función **push_back()** con el valor del entero entre los paréntesis.

Listing 3: Agregando valores

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> enteros;
    enteros.push_back(1);
    enteros.push_back(4);
    enteros.push_back(9);
}
```

Aqui, nuestro vector tendrá los valores de 1, 4 y 9 guardados.

Para ver o modificar el valor en algún indice, se puede utilizar el mismo sintaxis de un arreglo. Si en el ejemplo queremos cambiar el 9 a 7, podemos modificarlo y ver sus cambios con el siguiente código:

Listing 4: Modificando valores

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> enteros;
    enteros.push_back(1);
    enteros.push_back(4);
    enteros.push_back(9);
    for(int i = 0; i < 3; i++) {
        cout << enteros[i] << endl;
    }
    enteros[2] = 7;
```

```
    for(int i = 0; i < 3; i++) {
        cout << enteros[i] << endl;
    }
}
```

Primero se imprimiran los valores de 1, 4 y 9, luego se verán los valores 1, 4 y 7 en la consola.

Existen mas funciones de los vectores que son utiles, como **insert ()** que inserta elementos en ciertos indices, **erase** que elimina ciertos elementos, **clear ()** que borra todos los datos en un vector, **pop_back()** que elimina el último valor y finalmente **size ()**, que indica el tamaño de un vector.

Listing 5: Jugando con vectores

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    vector<int> enteros;
    for(int i = 0; i < 10; i++) {
        enteros.push_back(i);
    }
    enteros.insert(enteros.begin() + 5, 25);
    enteros.erase(enteros.begin() + 7);
    cout << "Cantidad: " << enteros.size() << endl;
    for(int i = 0; i < enteros.size(); i++) {
        cout << enteros[i] << endl;
    }
    enteros.clear();
    cout << "Cantidad: " << enteros.size() << endl;
}
```

Liga al código

Como se puede ver en el código de arriba, se crea un vector de enteros y se llena de los valores de 0 a 9, luego se inserta el valor 25 en el índice 5 y se elimina el valor en el índice 7. Luego se imprime el tamaño del arreglo (12 en ese momento), se imprimen todos los valores, se limpia el vector y finalmente se imprime el tamaño final (cero porque se limpió).

Se debe notar que para las funciones **insert** y **erase**, se ocupa llamar a la función begin para ese vector y luego se debe sumar el índice a ese valor. Esta suma luego determina el lugar en la memoria donde esta guardado el valor con ese índice.

1.2 Pares

A veces es conveniente guardar cosas en pares, por ejemplo se pueden guardar dos enteros en un par para representar las coordenadas **x** y **y** de un plano, o se puede guardar un caracter con un booleano indicando si es vocal. Se puede crear pares de cualquier tipo de dato y los dos tipos no tienen que ser iguales.

Para crear un par, se debe incluir la libreria **utility** y para declararlo se tiene que escribir **pair<tipo, tipo> nombre;**

Listing 6: Declarando pares

```
#include <iostream>
#include <utility>

using namespace std;

int main() {
    pair<int, float> miPar;
}
```

Para modificar o ver los datos en este par, se debe llamar el elemento **first** para el primer dato (en este caso el entero) y **second** para el segundo dato.

Listing 7: Declarando pares

```
#include <iostream>
#include <utility>

using namespace std;

int main() {
    pair<int, float> miPar;
    miPar.first = 20;
    miPar.second = 4.472136;
}
```

Combinado con arreglos o vectores, pueden servir para una multitud de aplicaciones. Digamos que queremos guardar todos los puntos de un triángulo en un plano para verlos después. Podemos guardar un arreglo de tres pares correspondientes a los tres puntos de ese triángulo:

Listing 8: Arreglo de pares

```
#include <iostream>
#include <utility>

using namespace std;

int main() {
    pair<int, int> triangulo[3];
```

```
    for(int i = 0; i < 3; i++) {
        int x, y;
        cin >> x >> y; //Lee las coordenadas
        pair<int, int> punto;
        punto.first = x;
        punto.second = y;
        triangulo[i] = punto;
    }
}
```

Como se puede ver en el código, primero leemos dos números de la consola tres veces, correspondiendo a las X y las Y de los puntos del triángulo. Luego guardamos estos valores en el par **punto**, donde el primer valor es la X y el segundo valor es la Y. Finalmente, guardamos ese par en el arreglo triángulo.

1.3 Mapas

El mapa es una estructura de datos bastante util debido a que mapea un valor a otro, como indicaría el nombre. Esto te permite asociar algún valor con otro para su busqueda rápida. Puedes declarar un mapa incluyendo la libreria **map** y escribiendo **map<tipo, tipo> nombre;**

Digamos que quieres guardar las edades de cada uno de tus amigos. Una manera de hacer esto es tener un arreglo de pares con el nombre de tu amigo y su edad, y luego para obtener la edad de alguno de ellos tendrías que iterar sobre todos tus amigos hasta encontrar el que quieres. Una alternativa más fácil es usar un mapa que simplemente recibe el nombre de tu amigo y que te da su edad.

Les mostraremos las dos opciones, empezando con la implementación de este problema con un arreglo de pares:

Listing 9: Implementación con pares

```
#include <iostream>
#include <utility>

using namespace std;

int main() {
    int n;
    cin >> n; //Lee el numero de amigos
    pair<string, int> edades[n];
    //Guardar todos los amigos y sus edades
    for(int i = 0; i < n; i++) {
        string nombre;
        int edad;
        cin >> nombre >> edad;
        pair<string, int> amigo;
```

```
        amigo.first = nombre;
        amigo.second = edad;
        edades[i] = amigo;
    }
    //Buscar el amigo "Juan" y desplegar su edad
    for(int i = 0; i < n; i++) {
        if(edades[i].first == "Juan") {
            cout << "Juan tiene " << edades[i].second << endl;
            break;
        }
    }
}
```

Como se puede ver, se guardaron todos los valores en un arreglo de pares y luego se tuvo que hacer una busqueda de todos los amigos hasta encontrar a Juan. Esta es la simplificación con map:

Listing 10: Implementación con map

```
#include <iostream>
#include <map>

using namespace std;

int main() {
    int n;
    cin >> n; //Lee el numero de amigos
    map<string, int> edades;
    for(int i = 0; i < n; i++) {
        string nombre;
        int edad;
        cin >> nombre >> edad;
        edades[nombre] = edad;
    }
    //Buscar el amigo "Juan" y desplegar su edad
    cout << "Juan tiene " << edades["Juan"] << endl;
}
```

Como se puede ver, el primer valor funciona como un estilo de índice que guarda el segundo valor, y se puede utilizar cualquier tipo de variable como este índice.

Pero tambien se debe aclarar que hay dos tipos de mapa, el mapa ordenado y el mapa desordenado. Hasta ahorita, hemos estado utilizando el mapa ordenado o **map**. El mapa desordenado se llama **unordered_map** y para utilizarlo debes incluir la libreria con este mismo nombre.

1.4 Sets

2 Complejidad de tiempo

3 Complejidad de memoria

4 Busquedas

4.1 Busqueda lineal

4.2 Busqueda binaria

4.3 Busqueda alfabetica

5 Ordenamientos

5.1 Ordenamiento de selección

5.2 Ordenamiento de inserción

5.3 Ordenamiento por mezcla

6 Matemáticas en C++

6.1 Libreria cmath

6.2 Desventajas