GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ESTRUCTURA DE DATOS CONJUNTO Y DICCIONARIO (SET Y MAP)

---

1. Introducción

Otras de las estructuras básicas que debe dominar un concursante están las estructuras de datos Conjuntos (*Set*) y Diccionarios (*Map*). A ellas les vamos a dedicar la siguiente guía de aprendizaje.

2. Conocimientos previos

2.1. Estructura de Datos

Una estructura de datos es una forma de organizar un conjunto de datos elementales con el objetivo de facilitar su manipulación. Un dato elemental es la mínima información que se tiene en un sistema. Una estructura de datos define la organización e interrelación de estos y un conjunto de operaciones que se pueden realizar sobre ellos. Cada estructura ofrece ventajas y desventajas en relación a la simplicidad y eficiencia para la realización de cada operación. De está forma, la elección de la estructura de datos apropiada para cada problema depende de factores como la frecuencia y el orden en que se realiza cada operación sobre los datos.

2.2. Estructuras Dinámicas No Lineales

Estas estructuras se caracterizan por que el acceso y la modificación ya no son constantes, es necesario especificar la complejidad de cada función de ahora en adelante. Entre las estructuras dinámicas no lineales están:

- Conjunto
- Diccionario
- Cola con Prioridad

2.3. Árbol rojo-negro

Un árbol rojo-negro es un tipo especial de árbol binario usado para organizar información compuesta por datos comparables. En estos árboles las hojas no son relevantes y no contienen datos. Estos árboles además de los requisitos impuestos propios de un árbol binario de búsqueda convencionales debe cumplir las siguientes reglas:

1. Todo nodo es o bien rojo o bien negro.
2. La raíz es negra.
3. Todas las hojas (NULL) son negras.
4. Todo nodo rojo debe tener dos nodos hijos negros.
5. Cada camino desde un nodo dado a sus hojas descendientes contiene el mismo número de nodos negros.

2.4. Árbol binario balanceado

Un árbol binario de búsqueda auto-balanceado o equilibrado es un árbol binario de búsqueda que intenta mantener su altura, o el número de niveles de nodos bajo la raíz, tan pequeño como sea posible en todo momento automaticamente.

2.5. Hashing

Es una técnica donde la idea básica es generar un valor hash es que sirva como una representación compacta de la cadena de entrada.

2.6. Iteradores

Estos son punteros que se utilizan para acceder a los elementos de una estructura de datos, un iterador soporta el operador ++, lo cual significa que podemos usarlo en sentencias de iteración (for,while).

3. Desarrollo

3.1. Conjunto (*Set*)

Conjunto es un contenedor asociativo que contiene un conjunto ordenado de objetos únicos. La ordenación se realiza utilizando la función de comparación. Los conjuntos se implementan normalmente como árboles rojo-negro, por lo que sus funciones principales (búsqueda, eliminación e inserción) tienen complejidad logarítmica O($log_2 N$). El acceso a está estructura se debe hacer con iteradores, o a ciertos datos por medio de funciones. Para la creación de un conjunto se debe mandar el tipo de dato siempre y cuando tenga un comparador por defecto, los datos primitivos si llevan comparadores definidos por defecto. Si se desearía crear un conjunto de otro tipo de datos que no sean primitivos del lenguaje se debe implementar una función booleana o entera que acepta dos variables del tipo de dato deseado y las compares según determinados criterios.

Tanto C++ como Java contiene dos implementaciones de esta estructura:

- Una estructura de datos conjuntos cuya implementación esta basada en un árbol binario balanceado cuyas operaciones tienen una complejidad O($\log N$). En C++ está el set mientras en Java $TreeSet$ responde a esta implementación.
- Una estructura de datos conjuntos cuya implementación esta basada en *hashing* cuyas operaciones tienen una complejidad o trabajan O(1) como promedio. En C++ está el $unordered_set$ mientras en Java $HashSet$ responde a esta implementación.

La elección de cual implementación utilizar a menudo depende del uso que deseas darle a la estructura, si te es beneficioso mantener el orden la implementación basada en árbol binario balanceado te será útil además de proveerte de un grupo de funcionalidades no disponibles en la implementación basada en *hashing*. Aunque a favor de esta implementación se puede decir que es mucho más eficiente que la otra.

En las implementaciones en esta guía utilizaremos la implementación basada en árbol binario balanceado pero utilizar el otro modelo de implementación no requiere gran cambio, en C++ sustituir set por $unordered_set$ mientras en Java sería cambiar $TreeSet$ por $HashSet$. Lo que si es importante recalcar que al hacer este cambio no vamos a poder utilizar algunas funcionales sobre todo aquellas que tienen que ver con el orden de como se organizan los elementos dentro de la estructura.

3.1.1. C++

Para la utilización de está estructura es necesario añadir en la cabecera del programa **#include <set>**, gracias a esto ya podemos utilizar la estructura de otro modo no.

- **set::empty():** Está función retorna verdad si el conjunto vacio y retorna falso en otro caso.
- **set::size():** Está función retorna cuantos elementos tiene el conjunto.
- **set::clear():** Está función elimina todos los elemtos del conjunto logrando así que el tamaño (.size()) sea 0.
- **set::insert(pos):** Está función inserta un elemento al conjunto si este ya existe lo ignorara, valor debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es O($\log_2 N$).
- **set::erase(inicio, fin):** Está función elimina elemtos desde una posición inicial (inicio) hasta otra posición (fin), inicio y fin deben ser iteradores. La complejidad es lineal.
- **set::erase(valor):** Está función busca el elemnto y lo elimina, **valor** debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es O($\log_2 N$).
- **set::count(valor):** Está función retorna la cantidad de elementos coincidentes con valor lo cual está entre 0 y 1 ya que no existen elementos dobles o repetidos, valor debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es O($log_2 N$).
- **set::find(valor):** Está función retorna el iterador apuntando al elemento valor si existe y si no retorna .end(), valor debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es O($log_2 N$).
- **set::lower_bound(valor):** Está función retorna un iterador apuntando al primer elemento no menos que valor, valor debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es O($log_2 N$).
- **set::upper_bound(valor):** Está función retorna un iterador apuntando al primer elemento mas grande que valor, valor debe ser un tipo de dato aceptado por el conjunto. La complejidad de está función es O($log_2 N$).

Multiconjunto: En el caso específico de C++ existe adicionalemente la estructura $multiset$ cuyo funcionamiento, implementación y operaciones es idéntico al set con la única variación que esta estructura si permite la duplicación de valores.

3.1.2. Java

Para la utilización de está estructura es necesario añadir en la cabecera del programa **import java.util.*;**, gracias a esto ya podemos utilizar la estructura de otro modo no. Los métodos disponibles para la clase set son los siguientes:

- **add(E elemento):** Agrega un elemento al conjunto.
- **clear():** Borra el conjunto.
- **addAll(coleccion):** Agrega toda una colección.
- **removeAll(coleccion):** Elimina todos los elementos de la colección que existan en el conjunto.
- **remove(Object objeto):** Elimina el objeto del conjunto. Devuelve true si ha sido eliminado
- **isEmpty():** Devuelve verdadero si el conjunto esta vacío.
- **contains(E elemento):** Devuelve verdadero si el conjunto contiene el elemento.
- **size():** Devuelve el número de elementos del conjunto.

3.2. Diccionarios (*Map*)

Diccionario o mapa es un contenedor asociativo que contiene pares de llaves y valores, las llave son únicas más no así los valores. La ordenación se realiza utilizando la función de comparación. Las aplicaciones se implementan normalmente como árboles rojo-negro, por lo que sus funciones principales (búsqueda, eliminación e inserción) tienen complejidad logarítmica O($log_2 N$). Para la creación de un mapa se debe mandar el tipo de dato de las llaves y el tipo de dato de los valores, siempre y cuando el tipo de dato de las llaves tenga un comparador por defecto, los datos primitivos sí llevan comparadores definidos por defecto. Si se desearía crear un mapa de otro tipo de datos que no sean primitivos del lenguaje se debe implementar una función booleana o entera que acepta dos variables del tipo de dato deseado y las compares según determinados criterios. Un mapa tiene la ventaja de acceder a los valores directamente mediante los corchetes [llave] y llaves, como si fuera un vector. Si se accede a un elemento mediante la llave y el corchete que no existe, se crea automáticamente asignando 0's como valores. También se puede acceder a las llaves y valores mediante iteradores.

Tanto C++ como Java contiene dos implementaciones de esta estructura:

- Una estructura de datos conjuntos cuya implementación esta basada en un árbol binario balanceado cuyas operaciones tienen una complejidad O($\log N$). En C++ está el map mientras en Java $TreeMap$ responde a esta implementación.
- Una estructura de datos conjuntos cuya implementación esta basada en *hashing* cuyas operaciones tienen una complejidad o trabajan O(1) como promedio. En C++ está el $unordered_map$ mientras en Java $HashMap$ responde a esta implementación.

La elección de cual implementación utilizar a menudo depende del uso que deseas darle a la estructura, si te es beneficioso mantener el orden la implementación basada en árbol binario balanceado te será útil además de proveerte de un grupo de funcionalidades no disponibles en la implementación basada en *hashing*. Aunque a favor de esta implementación se puede decir que es mucho más eficiente que la otra.

En las implementaciones en esta guía utilizaremos la implementación basada en árbol binario balanceado pero utilizar el otro modelo de implementación no requiere gran cambio, en C++ sustituir map por $unordered_map$ mientras en Java sería cambiar $TreeMap$ por $HashMap$. Lo que si es importante recalcar que al hacer este cambio no vamos a poder utilizar algunas funcionales sobre todo aquellas que tienen que ver con el orden de como se organizan los elementos dentro de la estructura.

3.2.1. C++

Para la utilización de está estructura es necesario añadir en la cabecera del programa **#include <map>**, gracias a esto ya podemos utilizar la estructura de otro modo no.

- **map::empty():** Está función retorna verdad si el diccionario está vacío y retorna falso en otro caso.
- **map::size():** Está función retorna cuantos elementos tiene el diccionario.
- **map::clear():** Está función elimina todos los elemtos de la aplicación logrando así que el tamaño (.size()) sea 0
- **map::insert(make_pair(llave, valor)):** Está función inserta un par de elementos la llave y el valor designado a la llave, llave y valor debe ser un tipo de dato aceptado por el diccionario. La complejidad de está función es O($log_2 N$).
- **map::erase(inicio, fin):** Está función elimina elemtos desde una posición inicial (inicio) hasta otra posición (fin), inicio y fin deben ser iteradores. La complejidad es lineal
- **map::erase(llave):** Está función busca el elemnto y lo elimina, llave debe ser un tipo de dato aceptado por la aplicación. La complejidad de está función es O($log_2 N$).
- **map::count(llave):** Está función retorna la cantidad de llaves coincidentes con llave lo cual está entre 0 y 1 ya que no existen llaves dobles o repetidos, llave debe ser un tipo de dato aceptado por la aplicación. La complejidad de está función es O($\log_2 N$).
- **map::find(llave):** Está función retorna el iterador apuntando a la llave si existe y si no retorna .end(), llave debe ser un tipo de dato aceptado por el diccionario. La complejidad de está función es O($log_2 N$).
- **map::lower_bound(llave):** Está función retorna un iterador apuntando a la primera llave no menos que llave, llave debe ser un tipo de dato aceptado por el diccionario. La complejidad de está función es O($log_2 N$).
- **map::upper_bound(llave):** Está función retorna un iterador apuntando a la primera llave

mas grande que llave, llave debe ser un tipo de dato aceptado por el diccionario. La complejidad de está función es O($log_2 N$).

Multidiccionario: En el caso específico de C++ existe adicionalemente la estructura $multimap$ cuyo funcionamiento, implementación y operaciones es idéntico al map con la única variación que esta estructura si la existencia de varios valores para una misma clave o llave.

3.2.2. Java

Para la utilización de está estructura es necesario añadir en la cabecera del programa **import java.util.*;**, gracias a esto ya podemos utilizar la estructura de otro modo no. Los métodos disponibles para la clase set son los siguientes:

- **put(Clave,Valor):** Agrega un elemento al diccionario.
- **get(Clave):** Devuelve el elemento del diccionario con la clave especificada.
- **clear():** Borra el conjunto Map.
- **remove(Object objeto):** Elimina el objeto del diccionario con la clave especificada. Devuelve verdadero si es eliminado.
- **isEmpty():** Devuelve verdadero si el diccionario esta vacío.
- **containsValue(Object elemento):** Devuelve verdadero si el diccionario contiene el valor.
- **containsKey(Object elemento):** Devuelve verdadero si el diccionario contiene la clave.
- **size():** Devuelve el numero de elementos del diccionario.

4. Implementación

4.1. C++

4.1.1. Conjunto (*Set*)

```
#include <iostream>
#include <set>
#include <vector>
using namespace std ;

bool comvec(vector<int> v1 ,vector<int> v2){//Esta es el comparador
   return v1.size()<v2.size();
}

int main() {
   //Creacion de un conjunto con 'vector<int>' como Tipo de Dato
   set<vector<int>, bool(*)( vector<int>, vector<int>)> conjvec (comvec);
```

```
set<int> conj ;
//Adicionamos al conjunto los elementos
conj.insert(69); conj.insert(80); conj.insert(77);
conj.insert(82); conj.insert(75); conj.insert(81);
conj.insert(78);
if(conj.empty()){cout<<" El conjunto esta vacio\n" ;}
else{
   cout<<" El conjunto lleva " ;
   cout<<conj.size()<<" elementos\n " ;
}//Salida : El conjunto lleva 7 elementos
for(set<int>::iterator it = conj.begin(); it!=conj.end(); it++){
   cout <<*it<<" ";
}
cout<<"\n" ;//Resultado: 69 75 77 80 81 82

if(conj.count(77)==1) cout<<" 77 esta en el conjunto\n";
else cout<<"77 no esta en el conjunto\n";
//Resultado: 77 esta en el conjunto

if(conj.find(100)!=conj.end()) cout<<"100 esta en el conjunto\n";
else cout<<" 100 no esta en el conjunto\n";
//Resultado: 100 no esta en el conjunto

if(conj.lower_bound(80)!=conj.upper_bound(80))
   cout<<" 80 esta en el conjunto\n";
else cout<<" 80 no esta en el conjunto\n" ;
//Resultado: 80 esta en el conjunto

if(conj.lower_bound(70)==conj.upper_bound(70))
   cout<<" 70 no esta en el conjunto\n";
else cout<<" 70 esta en el conjunto \n" ;
//Resultado: 70 no esta en el conjunto

set<int> c1, c2;
vector<int> unionc, difc, interc;

c1.insert(1);c1.insert(2);c1.insert(3);c1.insert(4);
c1.insert(5);c1.insert(6);c1.insert(7);

c2.insert(4);c2.insert(3);c2.insert(6);c2.insert(1);
c2.insert(9);c2.insert(10);c2.insert(7);

set_union(c1.begin(),c1.end(),c2.begin(),c2.end(),back_inserter(unionc));
set_difference(c1.begin(),c1.end(),c2.begin(),c2.end(),back_inserter(difc))
   ;
set_union(c1.begin(),c1.end(),c2.begin(),c2.end(),back_inserter(interc));

cout<<"Union :";for(auto x : unionc) cout<<x<<" "; cout<<endl;
cout<<"Diferencia :";for(auto x : difc) cout<<x<<" "; cout<<endl;
cout<<"Interseccion :";for(auto x : interc)cout<<x<<" ";cout<<endl;
```

```
   //Union :1 2 3 4 5 6 7 9 10
   //Diferencia :2 5
   //Interseccion :1 2 3 4 5 6 7 9 10

   return 0;
}
```

4.1.2. Diccionario (*Map*)

```
#include <iostream>
#include <map>
#include <vector>
using namespace std ;
bool comvec(vector<int> v1 , vector <int> v2 ){//Esta es el comparador
   return v1.size() < v2.size();
}

int main (){
   //Creacion de un diccionario con 'vector<int>'
   //como Tipo de Dato de las llaves
   map<vector<int>,int,bool(*)( vector<int> , vector<int>)> aplvec(comvec);

   map<char,int> apl;
   apl.insert(make_pair('a',13)); apl.insert(make_pair('b',98));
   cout<<apl['a']<<"\n"; cout<<apl['b']<<"\n";
   //Notese que no existe apl['c] por lo que se creara y pondra como valor 0
   cout<<apl['c']<<"\n";

   //Acceso a las llaves y valores mediante iteradores
   for(map<char,int>::iterator it=apl.begin();it!= apl.end();it++){
      cout<<it->first<<" "<<it->second<<"\n";
   }//Resultado: a 13 b 98 c 0

   if(apl.empty()){ cout<<" La aplicacion esta vacia\n";}
   else {
      cout<<" La aplicacion lleva ";
      cout<<apl.size()<<" elementos\n";
   }//Salida : La aplicacion lleva 3 elementos

   if(apl.count('a')==1) cout<<" 'a ' esta en el diccionario\n";
   else cout<<" 'a ' no esta en el diccionario\n";
   //Resultado: 'a' esta en el diccionario

   if(apl.find('d')!=apl.end()) cout<<" 'd ' esta en el diccionario\n";
   else cout<<" 'd ' no esta en el diccionario\n";
   //Resultado: 'd' no esta en el diccionario

   if(apl.lower_bound('c')!=apl.upper_bound('c'))
```

```
      cout<<" 'c ' esta en el diccionario\n";
   else
      cout<<" 'c ' no esta en el diccionario\n";
   //Resultado: 'c' esta en el diccionario

   if(apl.lower_bound('Z')==apl.upper_bound('Z'))
      cout<<" 'Z ' no esta en el diccionario\n" ;
   else cout<<" 'Z ' esta en el diccionario\n";
   //Resultado: 70 no esta en el diccionario
   return 0;
```

4.2. Java

4.2.1. Conjunto (*Set*)

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Set<String> conjA = new TreeSet<String>();
      conjA.add("aaa"); conjA.add("bbb"); conjA.add("aaa");
      conjA.add("ccc"); conjA.add("ddd");

      Set<String> conjB = new TreeSet<String>();
      conjB.add("aaa"); conjB.add("bbb"); conjB.add("bbb");
      conjB.add("xxx"); conjB.add("yyy");

      //hallar conjB interseccion conjA
      Set<String> conjC = new TreeSet<String>();
      conjC.addAll(conjA);
      // para intersectar A y B
      // hacemos C=A-B y luego A-C
      conjC.removeAll(conjB);
      conjA.removeAll(conjC);
      //listar
      Iterator<String> iter = conjA.iterator();
      while (iter.hasNext()) System.out.println(iter.next());
   }
}
```

4.2.2. Diccionario (*Map*)

```
import java.util.*;

public class Main {
   public static void main(String[] args) {
      Map<String,String> preferencias=new TreeMap<String,String>();
```

```
        preferencias.put("color","rojo"); preferencias.put("ancho","640");
        preferencias.put("alto","480");
        //listar todo
        System.out.println(preferencias);
        // Quitar color
        preferencias.remove("color");
        // cambiar una entrada
        preferencias.put("ancho","1024");
        // recuperar un valor
        System.out.println("Alto= " + preferencias.get("alto"));
        // iterar por todos los elementos
        for(Map.Entry<String,String> datos:preferencias.entrySet()){
           String clave = datos.getKey();
           String valor = datos.getValue();
           System.out.println("clave=" + clave + ", valor=" + valor);
        }
    }
}
```

5. Aplicaciones

Siempre es una buena idea usar las estructuras de datos que nos provee las bibliotecas estandar del lenguaje de programación que usemos ya que nos ahorra tiempo de implementación y trabajamos con algo que no produce errores al no ser un mal uso por nuestra parte.

En el caso de la estructura conjunto es muy útil para cuando tenemos o necesitamos una colección sin valores duplicados y tener los valores ordenados. Además con dicha estructura podemos simular las operaciones que comunmente se aplican en teoría de conjunto (unión, intersección, diferencia) de manera muy eficiente.

En el caso del diccionario lo podemos manipular como un arreglo cuya indexación no tiene que ser necesariamente númerica y comenzar por cero o uno, lo cual nos permite hacer una mejor optimización del uso de la memoria ya que con un arreglo pudiera existir posiciones que nunca utilizamos. Lo podemos utilizar para el algoritmo de *Counting Sort* en el caso que el intervalo de los números a ordenar es más grande de lo que permite dicho algoritmo. Es un mecanismo que nos permite llevar la frecuencia de ocurrecia de cierto valor dentro de una colección. Una forma rápida de obtener una determinada información a partir de una clave.

6. Complejidad

La complejidad de las operaciones de estás estructuras como hemos visto están en rango logarítmico O($\log N$) o son constantes O(1). Solo determinadas operaciones como la eliminación en rango llegan a ser lineales O(N).

7. Ejercicios propuestos

A continuación una lista de ejercicios que se puede resolver utilizando algunas o las dos estructuras de datos abordadas en esta guía:

- UVA - 11849 - CD
- UVA - 10887 - Concatenation of Languages
- UVA - 12049 - Just Prune The List
- UVA - 13148 - A Giveaway
- UVA - 902 - Password Search
- UVA - 11348 - Exhibition
- MOG - D - Database of clients