Estructuras de Datos Básicas

Pilas, Colas, Set, Map e Iteradores

¿Por qué son importantes?

- Permiten resolver problemas de forma **rápida y eficiente**
- Ayudan a organizar y manejar información

Pilas (`stack`)

- **LIFO**: Último en entrar, primero en salir
- Operaciones principales:
  - `push()` → agregar
  - `pop()` → quitar
  - `top()` → acceder al tope
  - `empty()` → verificar si está vacía

```cpp
stack<int> pila;
pila.push(7);
pila.push(5);
pila.push(10);
cout << pila.top() << endl; // imprime 10
pila.pop();
cout << pila.top() << endl; // imprime 5
```

Colas (`queue`)

- **FIFO**: Primero en entrar, primero en salir
- Operaciones principales:
  - `push()`
  - `pop()`
  - `front()`
  - `back()`
  - `empty()`

```cpp
queue<int> cola;
cola.push(1);
cola.push(2);
cout << cola.front() << endl; // imprime 1
cola.pop();
cout << cola.front() << endl; // imprime 2
```

Doble Cola (`deque`)

- Permite insertar y eliminar por **ambos extremos** en **O(1)**
- Combina propiedades de `stack` y `queue`
- Operaciones:
  - `push_back()`, `push_front()`, `pop_back()`, `pop_front()`
  - Acceso por índice: `dq[i]`

```cpp
deque<int> dq;
dq.push_back(1);
dq.push_front(2);
cout << dq.front() << endl;  // imprime 2
cout << dq.back() << endl;   // imprime 1
```

Cola de Prioridad (`priority_queue`)

- Siempre devuelve el **mayor** o **menor** elemento.
- Operaciones:
  - `push()`, `pop()`, `top()`, `empty()`
- Complejidad: **O(log n)**

Usos comunes:

- Seleccionar el mayor o el más importante

```cpp
// Priority queue default (devuelve mayor)
priority_queue<int> pq1;
pq1.push(3);
pq1.push(8);
pq1.push(1);
cout << pq1.top() << endl; // imprime 8
pq1.pop();
cout << pq1.top() << endl; // imprime 3

// Priority queue que devuelve menor
priority_queue<int, vector<int>, greater<int>> pq2;
pq2.push(3);
pq2.push(8);
pq2.push(1);
cout << pq2.top() << endl; // imprime 1
pq2.pop();
cout << pq2.top() << endl; // imprime 3
```

Ejercicio – Matching de Paréntesis

Te dan una cadena que contiene solo los caracteres `'('` y `')'`.

Debes determinar si los paréntesis están **correctamente balanceados**.

Ejemplos:

Input	Output
`( ( ) ( ) )`	YES
`) ( ) (`	NO
`( ( ( ) ) )`	YES
`( ( )`	NO

Ejercicio – Matching de Paréntesis

```cpp
#include<bits/stdc++.h>
using namespace std;

int main() {
    string s; cin >> s;

    stack<char> st;
    bool balanceado = true;

    for (char c : s) {
        if (c == '(') {
            st.push(c);
        } else if (c == ')') {
            if (st.empty()) { // no se puede matchear
                balanceado = false;
                break;
            }
            st.pop();
        }
    }
    if (!st.empty()) { // si quedan sin matchear no está balanceado
        balanceado = false;
    }

    if (balanceado) {
        cout << "YES" << endl;
    } else {
        cout << "NO" << endl;
    }

    return 0;
}
```

Set (`set`)

- Conjunto **ordenado** y sin repetidos
- Operaciones:
  - `insert()`
  - `erase()`
  - `count()` → devuelve 0 o 1 dependiendo de si está o no.
  - `find()` → devuelve un iterador al elemento x si está, o end() si no está.
  - `lower_bound(x)` → iterador al **primer elemento ≥ x**, o `end()` si no hay.
  - `upper_bound(x)` → iterador al **primer elemento > x**, o `end()` si no hay.
- Complejidad: **O(log n)**
- Existe también `multiset` que hace lo mismo, pero permite elementos repetidos.

```cpp
set<int> s;
s.insert(4);
s.insert(2);
s.insert(4); // no se repite

// usando count: devuelve 1 si está, 0 si no
if (s.count(2))
    cout << "Está el 2" << endl;

// usando find: devuelve iterador al elemento, o s.end() si no está
auto it = s.find(2);
if (it != s.end())
    cout << "Está el 2, valor: " << *it << endl; //con *it se accede al elemento

s.erase(2); // eliminar el 2

if (!s.count(2)) //revisar que ya no está
    cout << "El 2 ya no está" << endl;
```

Map (`map`)

- Diccionario **clave → valor** ordenado
- Operaciones:
  - `insert()`
  - `erase()`
  - `find()` → devuelve un iterador al elemento x si está, o end() si no está.
  - `[]` → accede al valor de una clave (y la crea si no existe)
  - `lower_bound(clave)` → iterador al **primer par con clave ≥ clave dada**.
  - `upper_bound(clave)` → iterador al **primer par con clave > clave dada**.
- Complejidad: **O(log n)**

Usos comunes:

- Contar frecuencias
- Asociar información a claves

```cpp
map<string, int> mp;
mp["gato"] = 3;
mp["perro"] = 5;

// acceso con []
cout << mp["gato"] << endl; // imprime 3

// acceso con find
auto it = mp.find("perro");
if (it != mp.end())
    cout << "perro tiene valor: " << it->second << endl;

mp.erase("gato"); // elimina la clave "gato"

if (!mp.count("gato"))
    cout << "gato ya no está" << endl;
```

- Los contenedores como `set` y `map` devuelven **iteradores**.
- Un iterador es como un puntero: se puede **avanzar**, **retroceder** y **acceder al valor** apuntado.
- Operaciones comunes:
  - `++it` / `--it`: avanzar o retroceder
  - `*it`: acceder al valor (clave o par clave→valor)
  - `it != contenedor.end()`: verificar si el iterador está en el contenedor

Ejemplo con `set`:

```cpp
set<int> s = {2, 4, 6};
auto it = s.begin();      // apunta al 2
cout << *it << endl;      // imprime 2
++it;                     // ahora apunta al 4
cout << *it << endl;      // imprime 4
```

Ejemplo con `map`:

```cpp
map<string, int> m = {{"a", 1}, {"b", 2}};
for (auto it = m.begin(); it != m.end(); ++it) {
    cout << it->first << " -> " << it->second << endl;
} //printearía "a - > 1"...
```

Ejercicio – ¿Pasan Todos los Niveles?

En un juego con `n` niveles, dos jugadores pueden pasar ciertos niveles.

Debes determinar si **colaborando** pueden pasar **todos los niveles** del 1 al `n`.

Ejemplos:

Input	Output

`4`
`3 1 2 3`
`2 2 4`
	`I become the guy.`

`4`
`3 1 2 3`
`2 2 3`
	`Oh, my keyboard!`

Ejercicio – ¿Pasan Todos los Niveles?

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n; cin >> n;
    set<int> niveles;

    int p; cin >> p;
    for (int i = 0; i < p; ++i) {
        int x; cin >> x;
        niveles.insert(x);
    }

    int q; cin >> q;
    for (int i = 0; i < q; ++i) {
        int x; cin >> x;
        niveles.insert(x);
    }

    if ((int)niveles.size() == n)
        cout << "I become the guy." << endl;
    else
        cout << "Oh, my keyboard!" << endl;

    return 0;
}
```

Resumen de Complejidades

Estructura	Inserción	Borrado	Búsqueda	Acceso Extremo
`stack` / `queue`	O(1)	O(1)	—	O(1)
`priority_queue`	O(log n)	O(log n)	—	O(1)
`set` / `map`	O(log n)	O(log n)	O(log n)	O(log n)

Consejos para competir!!!

- Antes de acceder a `top()`, `front()`, etc. revisen que la estructura no esté vacía (con `!estructura.empty()`)
- Pueden usar `unordered_map` o `unordered_set` para que las operaciones sean en **O(1)**, recomendado para inputs pequeños,
  porque en el peor caso las operaciones pueden ser **O(n)** por colisiones de hash.
- Si usas tipos como `pair`, `tuple`, o `struct` como clave en `unordered`, debes definir un **hash personalizado**. De lo contrario, el código no compila.
- Operaciones como `find()` son más útiles cuando se guarda más de un dato, como por ejemplo guardar un `pair<int, int>`.

Ejercicio – Palindrome Reorder

Se te da un string. Tu tarea es **reordenar sus letras** de tal forma que se convierta en un **palíndromo** (es decir, que se lea igual hacia adelante y hacia atrás).

Entrada:

Una única línea con un string de longitud `n` compuesto por letras mayúsculas `A–Z`.

Salida:

Imprime un palíndromo que use **todas** las letras del string original.
Si hay **más de una solución válida**, puedes imprimir cualquiera.
Si **no es posible**, imprime `"NO SOLUTION"`.

Link: https://cses.fi/problemset/task/1755