Estructuras de Datos No Lineales

by Alan Martinez

[ΓΩ5]

Diferencia entre Lineal y No Lineal

Las estructuras de datos no lineales son aquellas en las que los elementos no se organizan en una secuencia lineal o en una única dimensión. En su lugar, los elementos pueden tener múltiples relaciones entre sí, permitiendo representaciones más complejas y eficientes de datos. Ejemplos comunes incluyen árboles, grafos, mapas (diccionarios) y sets (conjuntos).

Caracteristicas

- **Relaciones Complejas:** Los elementos pueden estar conectados de múltiples maneras, permitiendo representar jerarquías o relaciones entre conjuntos de datos.
- **Acceso No Secuencial:** A diferencia de las estructuras lineales, donde los elementos se acceden en un orden específico, en las no lineales se pueden acceder a los elementos sin seguir un orden predefinido.
- **Almacenamiento Dinámico:** Muchas estructuras no lineales pueden crecer y reducirse dinámicamente, lo que permite una gestión eficiente de la memoria.
- **Optimización de Búsqueda:** Pueden ser más eficientes para búsquedas y operaciones específicas, como la búsqueda de elementos únicos (en sets) o la recuperación de pares clave-valor (en mapas).
- **Diversidad de Implementación:** Pueden ser implementadas de diversas maneras, como listas enlazadas, tablas hash, árboles binarios, etc., cada una con sus propias ventajas y desventajas.

Set (STL)

Los sets son un tipo de contenedor asociativo en el que cada elemento debe ser **único**, ya que el valor del elemento lo identifica. Los valores se almacenan en un orden específico, generalmente de manera ascendente.

La clase `std::set` forma parte de la Biblioteca Estándar de Plantillas (STL) de C++ y está definida dentro del archivo de cabecera `<set>`. Internamente, se implementa utilizando un árbol binario de búsqueda balanceado, lo que garantiza que las operaciones sean eficientes.

> Complejidad en el Tiempo: **O(log N)** para inserción, borrado y búsqueda, donde N es el número de elementos en el set.

Implementación

```
set<char> a;
a.insert('G');
a.insert('F');
a.insert('G');
for (auto& str : a) {
    cout << str << ' ';
}
cout << endl;
```

Funciones Básicas (operador punto `.`) de sets

- `begin()` – Retorna un iterador al primer elemento del set. → O(1)
- `end()` – Retorna un iterador al siguiente elemento teórico después del último. → O(1)
- `size()` – Retorna el número de elementos en el set. → O(1)
- `max_size()` – Retorna el número máximo de elementos que puede contener el set. → O(1)
- `empty()` – Retorna 1 si el set está vacío, 0 si tiene al menos un elemento. → O(1)
- `count(val)` – Retorna el número de elementos que coinciden con `val` (0 o 1 en un set). → O(log N)
- `insert(val)` – Inserta un elemento en el set. → O(log N)
- `erase(val)` – Elimina un elemento específico. → O(log N)
- `clear()` – Elimina todos los elementos del set. → O(N)
- `emplace(args...)` – Inserta un nuevo elemento construido en su lugar. → O(log N)
- `swap()` – Intercambia el contenido de dos sets. → O(1)

Unordered Set (STL)

Un `unordered_set` es un contenedor asociativo que almacena elementos de manera no ordenada, utilizando una **tabla hash** para gestionar las claves. Esto significa que los elementos no se almacenan en ningún orden en particular, y las claves se asignan a posiciones en la tabla mediante una función hash. A diferencia de los sets (`std::set`), que garantizan el orden de los elementos, en un `unordered_set` el orden de inserción es irrelevante.

- **Complejidad promedio**: Las operaciones como inserción, búsqueda y eliminación tienen un tiempo constante **Θ(1)** en promedio, lo que hace que `unordered_set` sea extremadamente eficiente para grandes conjuntos de datos.
- **Peor caso**: En el peor de los casos, si muchas claves se asignan al mismo índice (colisiones en la tabla hash), el rendimiento puede degradarse a **O(n)**, donde `n` es el número de elementos en el set. Esto depende de la calidad de la función hash interna.
- **Claves únicas**: Al igual que en los sets, los elementos en un `unordered_set` deben ser únicos.

La clase `std::unordered_set` está definida en el archivo de cabecera `<unordered_set>`.

Implementación

```
std::unordered_set<std::string> stringSet;
stringSet.insert("hoy");
stringSet.insert("es");
stringSet.insert("lunes");

std::string key = "martes";

// Buscar si la clave está en el unordered_set
if (stringSet.find(key) == stringSet.end())
    std::cout << key << " no encontrado" << std::endl;
else
    std::cout << "Encontrado " << key << std::endl;

// Mostrar todos los elementos
std::cout << "\nTodos los elementos: ";
for (const auto& elem : stringSet)
    std::cout << elem << std::endl;
```

Comparación entre Set y Unordered Set

Set	Unordered Set
Mantiene las claves en orden.	Almacena las claves en cualquier orden (desordenado).
Implementado como un árbol balanceado.	Implementado con tablas hash.
Operaciones básicas tienen complejidad O(log n).	Operaciones básicas tienen complejidad Θ(1) y O(n)

> Usa `unordered_set` cuando trabajes con grandes volúmenes de datos y la eficiencia en las operaciones sea crucial, mientras que `set` es más adecuado para conjuntos pequeños y ordenados donde mantener el orden de los elementos es importante.

Multisets (STL)

Los multisets son un tipo de contenedores asociativos similares al set, con la excepción de que **múltiples elementos pueden tener los mismos valores**.

La clase `std::multiset` está definida en el archivo de cabecera `<multiset>`.

> Complejidad en el Tiempo: **O(log N)** para la insercion de elementos, acceso a elementos y eliminación de elementos.

Implementación

```
multiset<int> a;
a.insert(10);
a.insert(10);
a.insert(10);

cout << a.count(10) << endl;

a.erase(a.find(10));
cout << a.count(10) << endl;

a.erase(10);
cout << a.count(10) << endl;
```

Mapas

Map (STL)

Los **maps**, hashmaps, o también conocidos como diccionaros en otros lenguajes, son contenedores asociativos que almacenan elementos de manera mapeada. Cada elemento tiene un valor clave y un valor mapeado. Ningún par de valores mapeados puede tener las mismas claves.

`std::map` es la plantilla de clase para contenedores map y está definida dentro del archivo de cabecera `<map>`.

> Para operaciones de inserción, eliminacion y búsqueda, la complejidad en tiempo es **O(log N)**

Implementación

```
map<string, int> contactos;
contactos["Blanca"] = 777777;
contactos["Manuel"] = 888888;
contactos["Jacobo"] = 999999;
cout << contactos.begin()->first << '\n'; // el menor nomnre
cout << contactos.rbegin()->first << '\n'; // el mayor nombre
// todos los nombres y números en orden de nombre:
for (map<string, int>::iterator it = contactos.begin(); it != contactos.end(); it++) {
    cout << it->first << ' ' << it->second << '\n';
}
```

Funciones Básicas (operador punto `.`) de Map

- `begin()` – Devuelve un iterador al primer elemento en el map. → **O(1)**
- `end()` – Devuelve un iterador al elemento teórico que sigue al último elemento en el map. → **O(1)**
- `size()` – Devuelve el número de elementos en el map. → **O(1)** (es constante, ya que el tamaño se almacena como parte del contenedor)
- `max_size()` – Devuelve el número máximo de elementos que el map puede contener. → **O(1)**
- `empty()` – Devuelve si el map está vacío. → **O(1)**
- `pair insert(keyvalue, mapvalue)` – Agrega un nuevo elemento al map. → **O(log n)** (debido a la estructura del árbol balanceado)
- `erase(iterator position)` – Elimina el elemento en la posición apuntada por el iterador. → **O(log n)**
- `erase(const g)` – Elimina la clave-valor ‘g’ del map. → **O(log n)**
- `clear()` – Elimina todos los elementos del map. → **O(n)** (debe recorrer todos los elementos para eliminarlos)

Unordered Map (STL)

Internamente, el `std::unordered_map` está implementado utilizando una **tabla hash**; la clave proporcionada para el mapeo se convierte en **índices** de una tabla hash, por lo que el rendimiento de la estructura de datos depende en gran medida de la función hash utilizada. Sin embargo, en promedio, el costo de búsqueda, inserción y eliminación de la tabla hash es **O(1)**.

El contenedor `std::unordered_map` está definida dentro del archivo de cabecera `<unordered_map>`.

> En el peor de los casos, su complejidad temporal puede pasar de **O(1)** a **O(n)**, especialmente para números primos grandes. En esta situación, se recomienda usar un `map` en su lugar para evitar recibir un error de **TLE**.

Implementación

```
unordered_map<int, string> mapaDesordenado;
mapaDesordenado[1] = "Plan";
mapaDesordenado[2] = "ta";
cout << mapaDesordenado[1] << '\n';
mapaDesordenado[1] = "Mar";
cout << mapaDesordenado[1] << mapaDesordenado[2] << '\n';
cout << mapaDesordenado.size() << '\n';
mapaDesordenado.erase(2);
cout << mapaDesordenado.size() << '\n';
mapaDesordenado[3] = "abc";
if (mapaDesordenado.count(3)) cout << "Contiene el nombre 3\n";
else cout << "No contiene el nombre 3\n";
```

Comparación entre unordered_map y map

Concepto	unordered_map	map
Definición	Un contenedor asociativo que almacena pares clave-valor sin un orden específico.	Un contenedor asociativo que almacena pares clave-valor en orden.
Implementación interna	Utiliza una tabla hash.	Implementado como un árbol balanceado.
Complejidad temporal	En promedio, las operaciones de búsqueda, inserción y eliminación son O(1).	Las operaciones de búsqueda, inserción y eliminación son O(log n).
Peor caso	La complejidad temporal puede llegar a O(n), especialmente con números primos grandes.	Mantiene O(log n) incluso en el peor de los casos.
Recomendación	En casos donde puede ocurrir O(n), se recomienda usar `map` para evitar un error TLE (Tiempo Límite Excedido).	Se prefiere cuando se requiere mantener el orden o evitar colisiones.

Multimap (STL)

Multimap es similar a un map con la diferencia de que múltiples elementos pueden tener las mismas claves. Además, **NO es necesario que el par de clave y valor mapeado sea único en este caso**. Una cosa importante a destacar sobre el multimap es que este **siempre mantiene todas las claves en orden**. Estas propiedades hacen que el multimap sea muy útil en la programación competitiva.

El contenedor `std::multimap` está definida dentro del archivo de cabecera `<multimap>`.

> Complejidad en el Tiempo: **O(log N)** para la insercion de elementos, acceso a elementos y eliminación de elementos.

Implementación

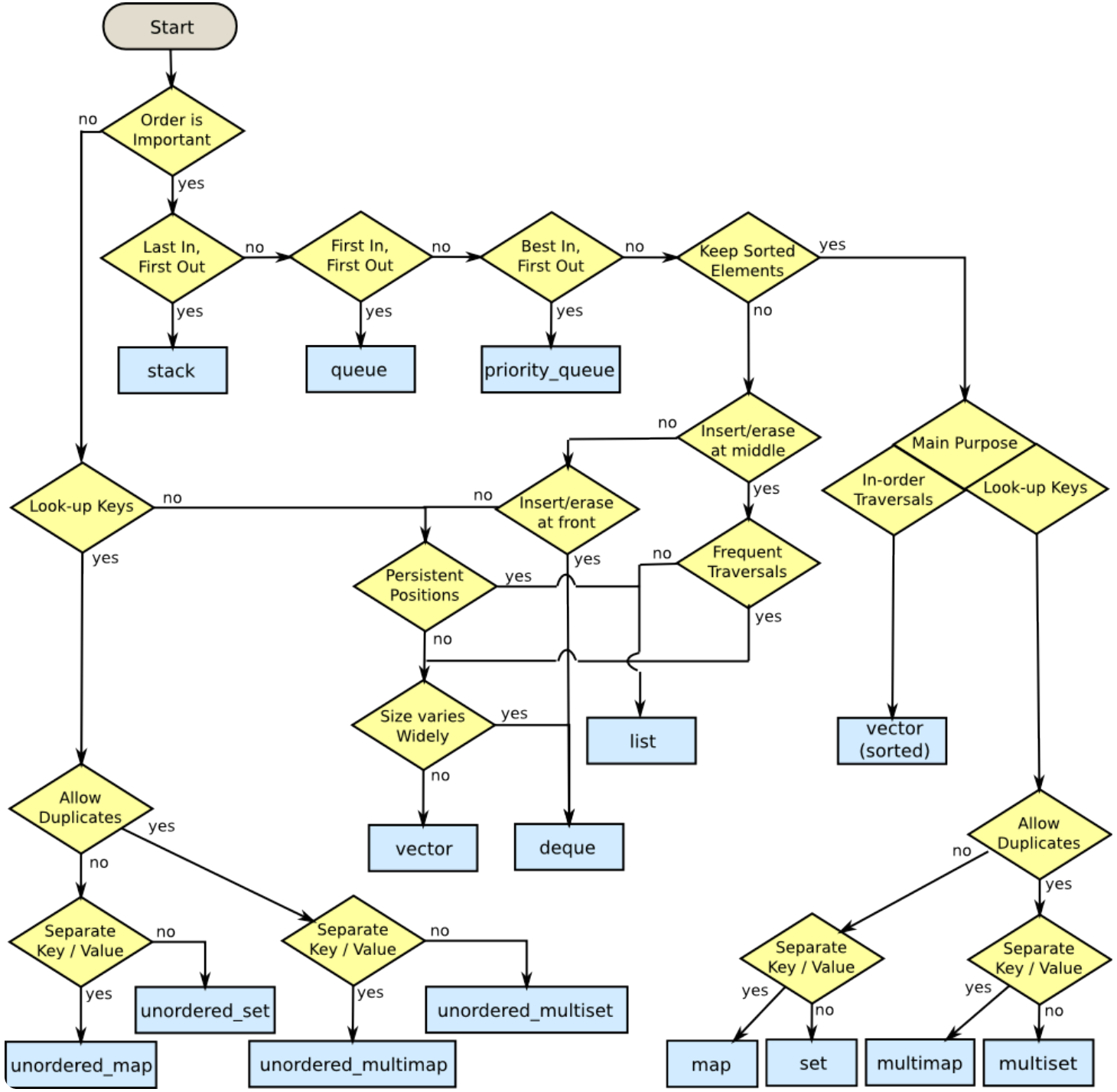
```
multimap<int, int> gquiz1;
gquiz1.insert(pair<int, int>(1, 40));
gquiz1.insert(pair<int, int>(2, 30));
gquiz1.insert(pair<int, int>(3, 60));
gquiz1.insert(pair<int, int>(6, 50));
gquiz1.insert(pair<int, int>(6, 10));
multimap<int, int>::iterator itr;
gquiz1.insert(pair<int, int>(4, 50));
gquiz1.insert(pair<int, int>(5, 10));
multimap<int, int> gquiz2(gquiz1.begin(), gquiz1.end());
cout << "\nThe multimap gquiz2 after assign from gquiz1 is : \n";
cout << "\tKEY\tELEMENT\n";
for (itr = gquiz2.begin(); itr != gquiz2.end(); ++itr)
    cout << '\t' << itr->first << '\t' << itr->second << '\n';
int num;
num = gquiz2.erase(4); // remove all elements with key = 4
```

Imprimir elementos de un mapa

```
for (const auto& pair : mapVar) // con pairs
    cout << pair.first << ": " << pair.second << endl;

auto it = mapVar.begin(); // con iteradores
while (it != mapVar.end()) {
    cout << it->first << ": " << it->second << endl;
    ++it;
}
```

¿Qué Estructura Usar para que caso?


Start
Order is Important
no
yes
Last In, First Out
First In, First Out
Best In, First Out
Keep Sorted Elements
stack
queue
priority_queue
Look-up Keys
Insert/erase at middle
Insert/erase at front
Persistent Positions
Frequent Traversals
Size varies Widely
list
vector
deque
Main Purpose
In-order Traversals
Look-up Keys
vector (sorted)
Allow Duplicates
Separate Key / Value
unordered_map
unordered_set
unordered_multimap
unordered_multiset
map
set
multimap
multiset

Problemas

- **469A** I Wanna Be the Guy
- **1730A** Planets

Referencias

- GeeksforGeeks. (n.d.). *Set in C++ STL*. Recuperado de https://www.geeksforgeeks.org/set-in-cpp-stl/
- GeeksforGeeks. (n.d.). *Unordered set in C++ STL*. Recuperado de https://www.geeksforgeeks.org/unordered_set-in-cpp-stl/
- GeeksforGeeks. (n.d.). *Multiset in C++ STL*. Recuperado de https://www.geeksforgeeks.org/multiset-in-cpp-stl/
- GeeksforGeeks. (n.d.). *Map – Associative containers | The C++ Standard Template Library (STL)*. Recuperado de https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/

  = Olimpiada Informática Mexicana. (n.d.). *Bitmask*. Recuperado de https://aprende.olimpiada-informatica.org/node/374
- Olimpiada Informática Mexicana. (n.d.). *Greedy*. Recuperado de https://aprende.olimpiada-informatica.org/node/375
- GeeksforGeeks. (n.d.). *Multimap – Associative containers | The C++ Standard Template Library (STL)*. Recuperado de https://www.geeksforgeeks.org/multimap-associative-containers-the-c-standard-template-library-stl/