Ejemplos de uso de contenedores STL en la simulación de redes de enrutadores

|  |  |
| --- | --- |
| **Contenedor STL** | **Uso sugerido en el proyecto** |
| vector | Lista de vecinos de un enrutador o lista de enrutadores. |
| list | Listas de adyacencia (especialmente si vas a insertar/eliminar con frecuencia). |
| map o unordered\_map | Asociar identificadores de enrutadores con objetos. |
| priority\_queue | Para implementar Dijkstra y encontrar el camino más corto. |

VECTOR se comporta como un arreglo (array), pero con la ventaja de que:

* **Cambia de tamaño automáticamente**.
* Puedes **agregar o quitar elementos fácilmente**.
* Tiene muchas funciones útiles, como push\_back, size, at, etc.

Ejemplo 1: Uso de vector para almacenar costos de conexión

El contenedor vector permite almacenar listas dinámicas de elementos. En el contexto de redes, puede usarse para guardar los costos de conexión de un enrutador a sus vecinos directos.

Ejemplo

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> costos = {4, 7, 5}; // costos desde A a B, C y D

for (int c : costos) {

cout << "Costo: " << c << endl;

}

return 0;

}

Ejemplo

#include <iostream>

#include <vector> // <-- aquí se importa la clase vector

using namespace std;

int main() {

vector<int> numeros = {1, 2, 3}; // Vector de enteros

numeros.push\_back(4); // Agregar un nuevo número

for (int n : numeros) {

cout << n << " ";

}

return 0;

}

LIST

* Representa una lista enlazada doblemente, donde cada elemento apunta al anterior y al siguiente.
* Permite inserciones y eliminaciones eficientes en cualquier parte de la lista.
* No tiene acceso aleatorio eficiente (no puedes acceder rápido por índice como en vector).
* Es ideal cuando necesitas modificar la lista frecuentemente (agregar o quitar elementos en medio).

# Ejemplo 2: Uso de list para almacenamiento dinámico con inserciones frecuentes

El contenedor list es útil cuando se necesitan muchas inserciones o eliminaciones en medio de la estructura, por ejemplo, para mantener una lista de vecinos actualizada dinámicamente.

#include <iostream>

#include <list> // <-- incluye la definición de std::list

using namespace std;

int main() {

list<int> numeros = {1, 2, 3};

numeros.push\_back(4); // Agrega un elemento al final

numeros.push\_front(0); // Agrega un elemento al inicio

// Recorremos la lista y mostramos los números

for (int n : numeros) {

cout << n << " ";

}

return 0;

}

MAP

* std::map es un contenedor que asocia claves únicas a valores.
* Funciona como un diccionario o tabla de búsqueda.
* Las claves están ordenadas (por defecto orden lexicográfico).
* Permite buscar, insertar y eliminar elementos con eficiencia logarítmica.

# Ejemplo 3: Uso de map para representar tablas de enrutamiento

El contenedor map asocia claves a valores. Es ideal para implementar tablas de enrutamiento, donde cada clave puede representar un nodo destino y el valor asociado el costo para llegar a él.

#include <iostream>

#include <map>

#include <string>

using namespace std;

int main() {

map<string, int> tablaEnrutamiento;

// Agregar destinos y costos

tablaEnrutamiento["B"] = 4;

tablaEnrutamiento["C"] = 7;

tablaEnrutamiento["D"] = 5;

// Mostrar tabla

for (const auto& [destino, costo] : tablaEnrutamiento) {

cout << "Destino: " << destino << ", Costo: " << costo << endl;

}

// Buscar un destino

string buscar = "C";

if (tablaEnrutamiento.find(buscar) != tablaEnrutamiento.end()) {

cout << "Costo para llegar a " << buscar << ": " << tablaEnrutamiento[buscar] << endl;

} else {

cout << "Destino " << buscar << " no encontrado en la tabla." << endl;

}

return 0;

}

Código:  
list<string> vecinos;  
vecinos.push\_back("B");  
vecinos.push\_back("C");  
vecinos.remove("B");

# SET

* std::set es un contenedor que almacena elementos únicos y ordenados.
* No permite elementos repetidos.
* Se usa para mantener conjuntos donde la unicidad es importante.
* Internamente, generalmente implementado con árboles balanceados (como árboles rojos-negros), lo que permite búsqueda, inserción y eliminación en tiempo logarítmico.
* En una simulación de redes, set es útil para guardar los nodos ya visitados durante algoritmos de búsqueda o recorrido (como Dijkstra).
* Así se evita procesar un nodo más de una vez.

# Ejemplo 4: Uso de set para mantener conjuntos únicos

El contenedor set mantiene elementos únicos y ordenados. En el algoritmo de Dijkstra, puede utilizarse para guardar los nodos ya visitados.

#include <iostream>

#include <set>

#include <string>

using namespace std;

int main() {

set<string> visitados;

// Insertar nodos visitados

visitados.insert("A");

visitados.insert("B");

visitados.insert("C");

// Intentar insertar repetido (no hará nada)

visitados.insert("B");

// Verificar si un nodo fue visitado

string nodo = "B";

if (visitados.count(nodo)) {

cout << nodo << " ya fue visitado." << endl;

} else {

cout << nodo << " no ha sido visitado." << endl;

}

// Mostrar todos los nodos visitados

cout << "Nodos visitados: ";

for (const auto& n : visitados) {

cout << n << " ";

}

cout << endl;

return 0;

*}*

queue

Permite usar contenedores de tipo cola, incluyendo:

* queue
* deque
* priority\_queue
* Es una cola de prioridad: los elementos se extraen en orden de máxima o mínima prioridad.
*  Por defecto, priority\_queue es un max-heap: el elemento con mayor valor sale primero.
*  Puedes convertirlo en min-heap (útil para Dijkstra) usando greater<>.

# Ejemplo 5: Uso de priority\_queue para selección de mínimo costo

priority\_queue es una cola de prioridad. Usada con greater<> se convierte en un min-heap, útil para seleccionar el nodo con menor costo durante el recorrido de Dijkstra.

***#include <iostream>***

***#include <queue>***

***#include <vector>***

***#include <string>***

***using namespace std;***

***int main() {***

***// Cola de prioridad con menor costo primero (min-heap)***

***priority\_queue<pair<int, string>, vector<pair<int, string>>, greater<>> cola;***

***// Insertar nodos con sus costos***

***cola.push({5, "D"});***

***cola.push({4, "B"});***

***cola.push({7, "C"});***

***// Procesar nodos por orden de menor costo***

***while (!cola.empty()) {***

***auto [costo, nodo] = cola.top();***

***cola.pop();***

***cout << "Nodo: " << nodo << ", Costo: " << costo << endl;***

***}***

***return 0;***

***}***