

Lists, Stacks, and Queues

Datos de tipo abstracto

Un tipo de datos abstracto (ADT) es un conjunto de objetos junto con un conjunto de operaciones. Abstracto los tipos de datos son abstracciones matemáticas; en ninguna parte de la definición de un ADT hay ninguna mención de cómo se implementa el conjunto de operaciones. Objetos como listas, conjuntos y gráficos, junto con sus operaciones, pueden verse como ADTs, tal como lo son los números enteros, reales y booleanos. Los tipos de datos. Los números enteros, reales y booleanos tienen operaciones asociadas con ellos, al igual que ADTs.

Listas

Una lista es una secuencia de nodos o elementos del mismo tipo, tal que, cada nodo señala, apunta, conoce, o sabe cuál es el siguiente nodo, (si existe), si no hay un nodo siguiente, entonces apunta a NULL, (una dirección segura).

Las listas son similares a los arreglos, pero a diferencia de los arreglos, las listas en algunos lenguajes de programación permiten insertar y eliminar elementos en cualquier posición en tiempo de ejecución.

Las listas son una estructura de datos dinámica lo que significa que su tamaño puede cambiar dinámicamente dependiendo de la cantidad de elementos que se agreguen o quiten.

como otras estructuras en programación, también nos permiten almacenar múltiples datos. Es el caso de variables como ENTRADA y SALIDA, que son -y siempre han sido- ¡listas! A partir de ahora las listas pasan a estar disponibles, como una estructura más, dentro del laboratorio.

Standard Template Library

La Standard Template Library (STL) es una colección de estructuras de datos genéricas y algoritmos escritos en C++. STL no es la primera de tales librerías, así la mayor parte de los compiladores de C++ disponen (o disponían) de librerías similares y, también, están disponibles varias librerías comerciales. Proporciona una colección de estructuras de datos contenedoras y algoritmos genéricos que se pueden utilizar con éstas. Una estructura de datos se dice que es contenedora si puede contener instancias de otras estructuras de datos.

Tipos de contenedores

Contenedores lineales. Almacenan los objetos de forma secuencial permitiendo el acceso a los mismos de forma secuencial y/o aleatoria en función de la naturaleza del contenedor. Ejemplos: Vector, lista y doble cola

Contenedores asociativos. En este caso cada objeto almacenado en el contenedor tiene asociada una clave. Mediante la clave los objetos se pueden almacenar en el contenedor, o recuperar del mismo, de forma rápida. Ejemplo: Conjunto, multiconjunto, aplicación y multiaplicación.

Contenedores adaptados. Permiten cambiar un contenedor en un nuevo contenedor modificando la interfaz (métodos públicos y datos miembro) del primero. En la mayor parte de los casos, el nuevo contenedor únicamente requerirá un subconjunto de las capacidades que proporciona el contenedor original. Ejemplo: Fila, cola y cola de prioridad.

Iteradores

Una de las operaciones más comunes sobre una estructura de datos contenedora es la de recorrer todos o algunos de los elementos almacenados en ella según un orden predefinido. Un iterador es un objeto que puede retornar por turno una referencia a cada uno de los elementos de un contenedor. Los iteradores son la base de construcción de los algoritmos genéricos.

Implementación de un Vector

el vector será un tipo de primera clase, lo que significa que a diferencia de la matriz primitiva en C++, el vector puede copiarse y la memoria que utiliza se puede recuperar automáticamente (a través de su destructor).

Para evitar ambigüedades con la clase de biblioteca, llamaremos a nuestra plantilla de clase Vector. Antes de examinar el código Vector, describimos los detalles principales:

1. El Vector mantendrá la matriz primitiva (a través de una variable de puntero al bloque de memoria asignada), la capacidad de la matriz y el número actual de elementos almacenados en el Vector.
2. Vector implementará los Big-Five para proporcionar una semántica de copia profunda para la copia. constructor y operador =, y proporcionará un destructor para recuperar la matriz primitiva. También implementará la semántica de movimiento de C++11.
3. El Vector proporcionará una rutina de cambio de tamaño que cambiará (generalmente a un número mayor) el tamaño del Vector y una rutina de reserva que cambiará (generalmente a una mayor número) la capacidad del Vector. La capacidad se cambia obteniendo un nuevo bloque de memoria para la matriz primitiva, copiando el bloque antiguo en el bloque nuevo, y recuperando el antiguo bloque.
4. El Vector proporcionará una implementación del operador [] (el operador [] generalmente se implementa tanto con un accesor como con un mutador versión).
5. El Vector proporcionará rutinas básicas, como tamaño, vacío, claro (que normalmente son frases ingeniosas), atrás, pop_back y push_back. La rutina push_back llamará a reserve si el El tamaño y la capacidad son los mismos.
6. Vector proporcionará soporte para los tipos anidados iterator y const_iterator, y métodos de inicio y fin asociados.

Implementación de una lista

Recuerde que la clase Lista se implementará como una lista doblemente enlazada y que Es necesario mantener punteros a ambos extremos de la lista. Esto nos permite mantener constante Costo de tiempo por operación, siempre que la operación ocurra en una posición conocida. La posición puede estar en cualquier extremo o en una posición especificada por un iterador.

Al considerar el diseño, necesitaremos proporcionar cuatro clases:

1. La clase Lista en sí, que contiene enlaces a ambos extremos, el tamaño de la lista y un host. de métodos.

2. La clase `Nodo`, que probablemente sea una clase anidada privada. Un nodo contiene los datos. y punteros a los nodos anterior y siguiente, junto con los constructores apropiados.
3. La clase `const_iterator`, que abstrae la noción de posición y es una publicación. clase anidada `lic`. El `const_iterator` almacena un puntero al nodo "actual" y proporciona Implementación de las operaciones básicas del iterador, todo en forma de operadores sobrecargados. como `=`, `==`, `!=` y `++`.
4. La clase `iteradora`, que abstrae la noción de posición y es una clase pública anidada. clase. El iterador tiene la misma funcionalidad que `const_iterator`, excepto que `operator*` devuelve una referencia al elemento que se está viendo, en lugar de una referencia constante a el objeto. Una cuestión técnica importante es que un iterador se puede utilizar en cualquier ruta. diente que requiere un `const_iterator`, pero no al revés. En otras palabras, el iterador `IS-A` `const_iterator`.

Stack en ADT

una pila (stack) es una estructura de datos lineal que se basa en el principio LIFO (Last In, First Out), es decir, el último elemento en entrar es el primero en salir. Una pila se comporta como una caja con varios elementos, en donde solo se pueden realizar dos operaciones: apilar (push) y desapilar (pop). Al apilar un elemento, se agrega al tope de la pila, y al desapilar se elimina el elemento que está en el tope de la pila. utilizan la pila para almacenar datos que son locales a un procedimiento. El espacio para los datos locales se asigna a los temas de la pila cuando el procedimiento se introduce, y son borradas cuando el procedimiento termina.

Las pilas tienen una serie de operaciones que se pueden realizar con ellas:

Insertar elementos: para insertar un elemento en la parte de arriba de la pila, podemos usar la función `push()`.

Eliminar elementos: para quitar el elemento de arriba del todo de la pila, podemos usar la función `pop()`.

Ver cuál es el elemento de arriba del todo de la pila: para ver cuál es este elemento, podemos usar la función `top()`. Esto es muy útil, ya que `pop()` únicamente quitará el primer elemento de la pila, pero no nos dirá cuál es. Esta es una diferencia fundamental entre C++ y otros lenguajes, así que los que vengáis de otros lenguajes tenéis que acordaros de esto, que tiende a ser un bug muy común.

Ver cuántos elementos tiene la pila: para saber esto, podemos utilizar la función `size()`. Además, podremos saber si la pila está vacía con la función `empty()`.

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> pila;
    pila.push(1);
    pila.push(2);
    cout << pila.size() << '\n';
    pila.pop();
    cout << pila.size() << '\n';
    pila.push(1);
    pila.push(3);
    cout << pila.top() << '\n';
    pila.pop();
}
```

```
pila.pop();
if (pila.empty()) cout << "La pila no tiene elementos\n";
else cout << "La pila tiene elementos\n";
return 0;
}
```

The Queue ADT

Las colas se caracterizan por ser estructuras first-in-first-out (FIFO). Esto significa que funcionan como una cola de personas en una panadería: la primera persona en llegar a la panadería será la primera en ser atendida y así sucesivamente hasta vaciar la cola.

Las colas tienen una serie de operaciones que se pueden realizar con ellas:

Insertar elementos: para insertar un elemento al final de la cola, podemos usar la función `push()`.

Eliminar elementos: para quitar el elemento del principio de la cola, podemos usar la función `pop()`.

Ver cuál es el primer elemento de la cola: para ver cuál es este elemento, podemos usar la función `front()`. Esto es muy útil, ya que `pop()` únicamente quitará el primer elemento de la cola, pero no nos dirá cuál es. Esta es una diferencia fundamental entre C++ y otros lenguajes, así que los que vengáis de otros lenguajes tenéis que acordaros de esto, que tiende a ser un bug muy común.

Ver cuál es el último elemento de la cola: hay un equivalente de la función `front()` para el final de la cola, la función `back()`.

Ver cuántos elementos tiene la cola: para saber esto, podemos utilizar la función `size()`. Además, podremos saber si la cola está vacía con la función `empty()`.

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> cola;
    cola.push(1);
    cola.push(2);
    cout << cola.size() << '\n';
    cola.pop();
    cout << cola.size() << '\n';
    cola.push(1);
    cola.push(3);
    cout << cola.front() << '\n';
    cola.pop();
    cola.pop();
    if (cola.empty()) cout << "La cola no tiene elementos\n";
    else cout << "La cola tiene elementos\n";
    return 0;
}
```