Lists, Stacks, y 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 ningún men- ción 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. 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, aplicacion y multiaplicacion. Contenedores adaptados. Permiten cambiar un contenedor en un nuevo contenedor modificando la interface (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. Implementacion 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 mayornúmero) 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. Implementacion 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:

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

8. 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.

9. 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 ++.

10. 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 operador* 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_iterador. 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();
Segundo Resumen.txt 2023-10-05
4 / 4
 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().

3.5 - Implementación de lista

Como en el caso del vector, el siguiente código muestra una implementación de una plantilla de clase de tipo lista, la cual es implementada como una lista de doble enlace y por lo tanto se necesita apuntar a ambos lados de la lista. Esto ayuda a mantener un costo de tiempo constante por operación, siempre y cuando se conozca la posición. La posición puede ser cualquier fin de la lista (inicio o fin) o una posición especificada por un iterador.

Se necesitan 4 clases:

1. La clase lista, que contiene enlaces en ambos puntos, el tamaño de la lista y un organizador de métodos.

2. La clase nodo, la cual probablemente sea una clase privada anidada. Un nodo contiene los datos y punteros a los nodos previos y siguientes, junto con los constructores apropiados.

3. La clase const_iterator, quien abstrae la noción de una posición y es una clase pública anidada. Este guarda un puntero al nodo actual y proporciona la implementación de las operaciones básicas de un iterador, en operadores sobrecargados como =, ==, != y ++.

4. La clase iterator, que abstrae la noción de una posición y es una clase pública anidada. Tiene una función similar al const_iterator, con excepción de que operator* regresa una referencia al objeto actual, en vez de una referencia constante al objeto. Un problema técnico importante es que el iterator puede ser usado en cualquier rutina que requiera un const_iterator, pero no vice versa.

Se puede crear un nodo extra al final de la lista para marcar el final de esta, al igual que un nodo extra al inicio de la lista para marcar su inicio. A estos nodos se les conoce como nodo sentinela, también conocidos como nodo cabeza (header node) al inicial, y nodo cola (tail node) al último. Usar estos dos nodos ayua a simplificar el código ya que ayuda a remover una parte de casos especiales. Por ejemplo, si no se usa un nodo inicial, entonces la eliminación del primer nodo se convierte en un caso especial, ya que se debe reiniciar el enlace de la lista al primer nodo durante la eliminación y el algoritmo de eliminación necesita acceder al nodo anterior al nodo que se está eliminando. Sin un nodo cabeza, el primer nodo no tiene un nodo anterior.

```
1 template <typename Object>
2 class List
3 {
4 private:
5 struct Node
6 { /* See Figure 3.13 */ };
7
8 public:
9 class const_iterator
10 { /* See Figure 3.14 */ };
11
12 class iterator : public const_iterator
13 { /* See Figure 3.15 */ };
14
15 public:
16 List( )
17 { /* See Figure 3.16 */ }
18 List( const List & rhs )
19 { /* See Figure 3.16 */ }
```

```
20 ~List( )
21 { /* See Figure 3.16 */ }
22 List & operator= ( const List & rhs )
23 { /* See Figure 3.16 */ }
24 List( List && rhs )
25 { /* See Figure 3.16 */ }
26 List & operator= ( List && rhs )
27 { /* See Figure 3.16 */ }
28
29 iterator begin( )
30 { return { head->next }; }
31 const_iterator begin( ) const
32 { return { head->next }; }
33 iterator end( )
34 { return { tail }; }
35 const_iterator end( ) const
36 { return { tail }; }
37
38 int size( ) const
39 { return theSize; }
40 bool empty( ) const
41 { return size( ) == 0; }
42
43 void clear( )
44 {
45 while( !empty( ) )
46 pop_front( );
47 }
48 Object & front( )
49 { return *begin( ); }
50 const Object & front( ) const
51 { return *begin( ); }
52 Object & back( )
53 { return *--end( ); }
54 const Object & back( ) const
55 { return *--end( ); }
56 void push_front( const Object & x )
57 { insert( begin( ), x ); }
Explicación del Código
58 void push_front( Object && x )
59 { insert( begin( ), std::move( x ) ); }
60 void push_back( const Object & x )
61 { insert( end( ), x ); }
62 void push_back( Object && x )
63 { insert( end( ), std::move( x ) ); }
64 void pop_front( )
65 { erase( begin( ) ); }
66 void pop_back( )
67 { erase( --end( ) ); }
68
69 iterator insert( iterator itr, const Object & x )
70 { /* See Figure 3.18 */ }
71 iterator insert( iterator itr, Object && x )
72 { /* See Figure 3.18 */ }
```

```
73
74 iterator erase( iterator itr )
75 { /* See Figure 3.20 */ }
76 iterator erase( iterator from, iterator to )
77 { /* See Figure 3.20 */ }
78
79 private:
80 int theSize;
81 Node *head;
82 Node *tail;
83
84 void init( )
85 { /* See Figure 3.16 */ }
86 };
```

Conclusiones

Es bastante interesante como los vectores y las listas juegan una parte importante de los codigos, como estos ayudan a enlistar una serie de elementos como el agregar, eliminar, modificar, leer y econtrar dichos elementos. Todos juegan una parte importante, obviamente con sus ventajas y desventajas.

Referencias

Iteradores y generadores - JavaScript | MDN. (2023, 24 julio). https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Iterators_and_generators