Resumen Listas

Este capitulo y resumen analiza tres de las estructuras de datos mas básica y simples, que al final todos los programas "Importantes" Usaran al menos una de estas. entre los temas del capitulo son los siguientes:

- Introducir el concepto de tipos de datos abstractos (ADT).
- Muestre cómo realizar operaciones eficientemente en listas.
- Introduzca la pila ADT y su uso en la implementación de la recursividad.
- Introducir la cola ADT y su uso en sistemas operativos y diseño de algoritmos.

3.1 tipo de datos abstractos o (ADT).

Un ADT es un conjunto de objetos junto con un conjunto de operaciones. objetos como listas. Objetos como listas, conjuntos y gráficos, junto con sus operaciones se pueden considerar como ADT, del mismo modo como los números enteros, reales y booleanos , tienen operaciones asociadas, al igual que los ADT. Para el conjunto ADT, podríamos tener operaciones como agregar, eliminar, dimensionar y contener. es posible que solo se agreguen dos operaciones, unión y búsqueda.

Las clases en c++ permite la implementación de las ADT, agregando adecuadamente los detalles de la implementación, Por lo tanto cualquier otra parte del programa que necesite hacer una operación con el ADT podrá hacerlo llamando el apartado indicado.

No hay una regla que indique que operaciones admiten los ADT, es mas bien una cuestión de diseño y del programador. se verán tres estructuras de datos y como se implementan correctamente.

3.2 The list ADT.

Usando como base una lista de forma A0, A1, A2, ... An-1. decimos que el tamaño de la lista es de N. por la tanto la llamaremos lista vacía.

Para cualquier lista excepto la antes mencionada lista vacía, decimos que el primer elemento es A0 y el ultimo es An-1. por ende no se define el ni el antecesor de A0 ni el sucesor de AN-1.

Aociado a estas definiciones hay un conjunto de operaciones que podemos realizar con las lista talez como. printList y makeEmpty, que devuelve la posición de la primera aparición de un elemento; insertar y eliminar, que generalmente insertan y eliminan algún elemento de alguna posición en la lista; y findKth, que devuelve el elemento en alguna posición, ejemplificando si la

lista contiene los siguientes datos (34, 12, 52, 16, 12) entonces find(52) te devolvería el valor 2 que es la posición de 52; insert(x,2) haría que la lista se viera ahora así, (34 ,12 ,x ,52 ,16 ,12), insertando una x en la posición del 52.
también se podrían agregar operaciones que el programador necesite y decida como anterior y siguiente que devuelvan los valores anteriores y siguientes del dato solicitado.

3.2.1 simple array implementación of list.

Todas las instrucciones se pueden implementar simplemente usando una matriz. aunque los arreglos se crean con la capacidad fija, la clase vector permite crear un arreglo dinámico que crece como se vaya necesitando. esto resuelve el problema mas grave del uso de una matriz.

Una implementación de matriz permite que printList se lleve a cabo en tiempo real y la operación de findKth requiere un tiempo constante que se puede esperar. sin embargo la inserción y eliminación son potencialmente costosas dependiendo de donde se produzca la inserciones o eliminaciones. en el peor de los casos , el insertar en la posición 0 o en el inicio de la lista requiere empujar toda la matriz un espacio hacia arriba y el eliminar es recorrer toda la matriz un espacio hacia abajo. entonces el peor de los casos para estas operaciones es empezar en el A0. en promedio es necesario mover la mitad de la lista para cualquiera de las operaciones por lo que aun se requiere tiempo lineal. por otro lado si eliminamos o agregamos al final de la lista esta accion requiere un tiempo de 0(1).

Hay Muchas situaciones en la que la lista se construye mediante inserciones en el extremo superior y luego solo se producen operaciones de acceso a la matriz, es decir operaciones de findKth. en tal caso la matriz es una implementación adecuada. en lo contrario la opción mas optima seria una lista enlazada.

3.2.2 simple linked list

Para evitar el costoso lineal de inserción y eliminación, debemos asegurarnos que la lista no se almacene de forma contigua, ya que de lo contrario será necesario mover partes enteras de la lista.

La lista enlazada consta de una serie de nodos, que no necesariamente son adyacentes en la memoria. cada nodo contiene el elemento y un enlace al sucesor.

Para ejecutar comandos como printLust() o find(x), simplemente se comienza en el primer nodo de la lista y luego la recorremos siguiendo los siguientes enlaces. Esta operación es en tiempo lineal como en la implementación de la matriz.

El método de eliminación se puede ejecutar simplemente omitiendo el dato seleccionado y haciendo el cambio de puntero.

El método de inserción requiere obtener un nuevo nodo del sistema mediante una nueva llamada luego ejecutar dos maniobras el cual es agregar el nuevo elemento y conectarlo con el nodo anterior y el subsiguiente.

Como podemos ver en un principio sabemos donde se va a realizar un cambio eliminar insertar o mover un elemento de la lista no requiere mover muchos elementos y en cambio implica solo un numero constante y un cambio de nodos.

La idea obvia de mantener un tercer enlace al penúltimo nodo no funciona, porque también sería necesario actualizarlo durante una eliminación. En cambio, hacemos que cada nodo mantenga un enlace a su nodo anterior en la lista.

3.3 vector and list in the stl

Dentro de c++ se incluyen una biblioteca de plantillas dentro de la misma esta la biblioteca estándar de plantillas (STL por sus siglas en ingles). List ADT es una de las estructuras que implementa STL.

Hay dos implementaciones populares de ADT. El vector proporciona una implementación de matriz con capacidad de crecimiento. La ventaja de utilizar el vector es que es indexable (es decir es: Registrar ordenadamente datos e informaciones, para elaborar su índice.)en tiempo constante. La desventaja es que la inserción de nuevos elementos y la eliminación de elementos existentes es costosa, a menos que los cambios se realicen al final del vector. La lista proporciona una implementación de lista doblemente enlazada de list ADT. La ventaja de utilizar la lista es que la inserción de nuevos elementos y la eliminación de elementos existentes es económica, siempre que se conozca la posición de los cambios. La desventaja es que la lista no es fácilmente indexable.

Tanto el vector como la lista son ineficientes para las búsquedas. Tanto el vector como la lista son plantillas de clase de las que se crean instancias con el tipo de elementos que almacenan. Ambos tienen varios métodos en común. Los primeros tres métodos mostrados están disponibles para todos los contenedores STL:

- int size() const: -> devuelve el numero de elementos en el contenedor.
- void clear(): -> elimina todos los elementos del contenedor.
- bool empty: const -> devuelve true si el contenedor tiene datos false si no.

3.3.1 Iterators

Algunas operaciones en listas, sobre todo aquellas para insertar y eliminar del medio de la lista, requieren la noción de posición. En STL, una posición esta representada por un tipo anidado, iterador.Al describir algunos métodos, simplemente usaremos iterador como

abreviatura, pero al escribir código, usaremos el nombre de clase anidada real. Inicialmente, hay tres cuestiones principales que abordar: primero, cómo se obtiene un iterador; segundo, qué operaciones pueden realizar los propios iteradores; tercero, qué métodos List ADT requieren iteradores como parámetros.

Getting an iterator

Para el primer problema, las listas STL definen un par de métodos:

- iterator begin() ->: devuelve un iterador apropiado que representa el primer elemento.
- Iteratir ends ->: devuelve un iterador apropiado que representa el marcador final

iterator methods:.

los iteradores se pueden comparar, probablemente tenga constructores de copia y operador= definidos. Por tanto, los iteradores tienen métodos, y muchos de los métodos utilizan la sobrecarga de operadores. Además de copiar, lo más común
Las operaciones utilizadas en iteradores incluyen las siguientes:

1. itr++ y ++itr
2. xitr
3. itr1 == itr2
4. itr1 != itr2

container Operations That require iterators

Para el último número, los tres métodos más populares que requieren iteradores son aquellos que agregan o eliminar de la lista en una posición especifica.

- Iterator insert().
- Iterator erase().

ejemplo

```
template
void removeEveryOtherItem( Container & lst )
{
auto itr = lst.begin( ); // itr is a Container::iterator
while( itr != lst.end( ) )
{
itr = lst.erase( itr );
if( itr != lst.end( ) )
++itr;
```

```
}
}
```

3.3.3 const_Iterators

El resultado del puntero de itr es solo el valor del elemento que esta viendo el itr sino también el elemento en si . esta distinción hace que los iteradores sean muy poderosos pero también introduce algunas complicaciones. Para ver el beneficio, supongamos que queremos cambiar todos los elementos de una colección a un valor específico.

```
template <typename Container, typename Object>
void change( Container & c, const Object & newValue )
{
typename Container::iterator itr = c.begin( );
while( itr != c.end( ) )
*itr++ = newValue;
}
```

Para ver el problema potencial, supongamos que el contenedor c se pasó a una rutina utilizando una llamada por referencia constante. Esto significa que esperaríamos que no se permitieran cambios en c, y el compilador garantizaría esto al no permitir llamadas a ninguno de los mutadores de c.

```
void print( const list<int> & lst, ostream & out = cout )
{
typename Container::iterator itr = lst.begin( );
while( itr != lst.end( ) )
{
out << *itr << endl;
*itr = 0; // This is fishy!!!
++itr;
}
}
```

La solución proporcionada por STL es que cada colección contiene no solo un tipo anidado de iterador sino también un tipo anidado const_iterator. La principal diferencia entre un iterador y const_iterator es que operator* para const_iterator devuelve una referencia constante y, por lo tanto, xitr para const_iterator no puede aparecer en el lado izquierdo de una declaración de asignación. Además, el compilador te obligará a utilizar un const_iterator para recorrer una colección constante. Lo hace proporcionando dos versiones de inicio y dos versiones de fin.

3.4 Implementation of vector

En esta sección, proporcionamos la implementación de una plantilla de clase vectorial utilizable. El vector será un tipo de primera clase, lo que significa que, a diferencia de la matriz primitiva en C++, el vector se puede copiar y la memoria que utiliza se puede recuperar automáticamente

```
#include <algorithm>

template <typename Object>
class Vector
{
public:
explicit Vector( int initSize = 0 ) : theSize{ initSize },
theCapacity{ initSize + SPARE_CAPACITY }
{ objects = new Object[ theCapacity ]; }

 Vector( const Vector & rhs ) : theSize{ rhs.theSize },
theCapacity{ rhs.theCapacity }, objects{ nullptr }
{
objects = new Object[ theCapacity ];
for( int k = 0; k < theSize; ++k )
objects[ k ] = rhs.objects[ k ];
}

Vector & operator= ( const Vector & rhs )
{
Vector copy = rhs;
std::swap( *this, copy );
return *this;
}

~Vector( )
{ delete [ ] objects; }

Vector( Vector && rhs ) : theSize{ rhs.theSize },
theCapacity{ rhs.theCapacity }, objects{ rhs.objects }
{
rhs.objects = nullptr;
rhs.theSize = 0;
rhs.theCapacity = 0;
}

Vector & operator= ( Vector && rhs )
{
std::swap( theSize, rhs.theSize );
std::swap( theCapacity, rhs.theCapacity );
```

```
std::swap( objects, rhs.objects );

return *this;
}

void resize( int newSize )
{
if( newSize > theCapacity )
reserve( newSize * 2 );
theSize = newSize;
}

void reserve( int newCapacity )
{
if( newCapacity < theSize )
return;

Object *newArray = new Object[ newCapacity ];
for( int k = 0; k < theSize; ++k )
newArray[ k ] = std::move( objects[ k ] );

theCapacity = newCapacity;
std::swap( objects, newArray );
delete [ ] newArray;
}
```

3.5 Implementacion of list

En esta sección, proporcionamos la implementación de una plantilla de clase de lista utilizable. Como en el caso de la clase vector, nuestra clase lista se llamará Lista para evitar ambigüedades con la clase biblioteca.

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

- la clase contiene enlaces en ambos extremos, el tamaño de la lista y una gran cantidad de métodos.
- la clase nodo, que probablemente sea una clase anidada privada. Un nodo contiene los datos.
- La clase const_iterator, que abstrae la noción de posición y es una clase anidada pública. El const_iterator almacena un puntero al nodo "actual" y proporciona implementación de

las operaciones básicas del iterador, todo en forma de operadores sobrecargados

- La clase iteradora, que abstrae la noción de posición y es una clase pública anidada. 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 al elemento. Una cuestión técnica importante es que un iterador se puede utilizar en cualquier rutina que requiera un const_iterator, pero no al revés.

```
template <typename Object>
class List
{
private:
struct Node
{ /* See Figure 3.13 */ };

public:
class const_iterator
{ /* See Figure 3.14 */ };

class iterator : public const_iterator
{ /* See Figure 3.15 */ };

public:
List( )
{ /* See Figure 3.16 */ }
List( const List & rhs )
{ /* See Figure 3.16 */ }
~List( )
{ /* See Figure 3.16 */ }
List & operator= ( const List & rhs )
{ /* See Figure 3.16 */ }
List( List && rhs )
{ /* See Figure 3.16 */ }
List & operator= ( List && rhs )
{ /* See Figure 3.16 */ }

iterator begin( )
{ return { head->next }; }
const_iterator begin( ) const
{ return { head->next }; }
iterator end( )
{ return { tail }; }
const_iterator end( ) const
{ return { tail }; }

int size( ) const
```

```
{ return theSize; }
bool empty( ) const
{ return size( ) == 0; }

void clear( )
{
while( !empty( ) )
pop_front( );
}
Object & front( )
{ return *begin( ); }
const Object & front( ) const
{ return *begin( ); }
Object & back( )
{ return *--end( ); }
const Object & back( ) const
{ return *--end( ); }
void push_front( const Object & x )
{ insert( begin( ), x ); }
void push_front( Object && x )
{ insert( begin( ), std::move( x ) ); }
void push_back( const Object & x )
{ insert( end( ), x ); }
void push_back( Object && x )
{ insert( end( ), std::move( x ) ); }
void pop_front( )
{ erase( begin( ) ); }
void pop_back( )
{ erase( --end( ) ); }

iterator insert( iterator itr, const Object & x )
{ /* See Figure 3.18 */ }
iterator insert( iterator itr, Object && x )
{ /* See Figure 3.18 */ }

iterator erase( iterator itr )
{ /* See Figure 3.20 */ }
iterator erase( iterator from, iterator to )
{ /* See Figure 3.20 */ }

private:
int theSize;
Node *head;
Node *tail;

void init( )
```

```
{ /* See Figure 3.16 */ }
};


struct Node
{
Object data;
Node *prev;
Node *next;

Node( const Object & d = Object{ }, Node * p = nullptr,
Node * n = nullptr )
: data{ d }, prev{ p }, next{ n}{}

Node( Object && d, Node * p = nullptr, Node * n = nullptr )
: data{ std::move( d ) }, prev{ p }, next{ n}{}
};
```

3.6 the stack ADT

Una pila es una lista con la restricción de que las inserciones y eliminaciones se pueden realizar solo en una posición, es decir el final de la lista llamada parte superior.

3.6.1 stack model

Las operaciones en una pila son push y pop que equivalen a insertar y eliminar el elemento insertado mas reciente. El elemento insertado más recientemente se puede examinar antes de realizar un pop mediante el uso de la rutina superior. Un pop o top en una pila vacía generalmente se considera un error. Por otro lado, quedarse sin espacio al realizar una inserción es un límite de implementación, pero no un error. Las operaciones habituales para crear pilas vacías y probar si están vacías son parte del repertorio, pero esencialmente todo lo que puedes hacer con una pila es push o pop.

3.6.2 Implementacion of stacks

Dado que una pila es una lista, cualquier implementación de lista servirá. Listar claramente y operaciones de pila de soporte vectorial. En ocasiones, puede resultar más rápido diseñar una implementación con un propósito especial. Debido a que las operaciones de pila son operaciones de tiempo constante, es poco probable que esto produzca una mejora perceptible excepto en circunstancias muy singulares. Para estos tiempos especiales, brindaremos dos implementaciones de pila populares. Uno usa una estructura vinculada y el otro usa una matriz, y ambos simplifican la lógica en vector y lista.

3.6.3 applications

Si restringimos las operaciones permitidas en una lista, esas operaciones se puedan realizar muy rápidamente. sin embargo, es que el pequeño número de operaciones que quedan son tan poderosas e importantes. a continuación un listado de las mismas.

- Balancing symbols
- Postfix Expressions
- infix to postfix conversion
- funtion calls

3.7 the queue ADT

al igual que las pilas, las colas son listas, sin embargo, con una cola, la inserción se realiza en un extremo, mientras que la eliminación se realiza en el otro extremo.

3.7.1 queue model

Las operaciones básicas en una cola son poner en cola, que inserta un elemento al final de la lista.

3.7.2 array implementation of queues

Al igual que con las pilas, cualquier implementación de lista es legal para las colas. Al igual que las pilas, tanto las implementaciones de listas vinculadas como de matrices brindan tiempos de ejecución rápidos para cada operación.

Para cada estructura de datos de la cola, mantenemos una matriz, theArray, y las posiciones al frente y al final, que representan los extremos de la cola. También realizamos un seguimiento de la cantidad de elementos que realmente están en la cola, currentSize. La siguiente tabla muestra una cola en algún estado intermedio.

Las operaciones deben ser claras. Para poner en cola un elemento x, incrementamos el tamaño actual y viceversa, luego configuramos theArrayback = x. Para quitar de la cola un elemento, establecemos el valor de retorno en theArrayfront, disminuimos el tamaño actual y luego incrementamos el frente.

Existe un problema potencial con esta implementación. Después de 10 puestas en cola, la cola parece estar llena, ya que atrás ahora se encuentra en el último índice de la matriz y la siguiente cola estaría en una posición inexistente. Sin embargo, es posible que solo haya unos pocos elementos en la cola, porque es posible que varios elementos ya se hayan retirado de la

cola. Las colas, al igual que las pilas, suelen ser pequeñas incluso en presencia de muchas operaciones.

La solución simple es que cada vez que el frente o la parte posterior llegan al final de la matriz, se envuelve hasta el principio. Las siguientes tablas muestran la cola durante algunas operaciones. Esto se conoce como implementación de matriz circular.

Opiniones criticas y conclusiones

#opinion_critica

Después de el capitulo 1 este capitulo es mas fácil de leer pero también es mas complejo, el tema es relativamente sencillo pero la implementación es lo que es lo complicado aun así sueles usar templates para poder crearlas entonces e vuelven mas generales y fáciles.
sigue teniendo el fallo de veras mas en el capitulo tal pero se entiende al ser tan extremadamente complejo deben de ponerlo de distintas partes.

#Conclusiones

el tema es sencillo, es poquito fácil de entender, pero necesitaría al igual que las clases hacer muchas listas para poder decir que lo entiendo de forma perfecta. ocuparia una revisada y un repaso del mismo tema.

#Referencias

Listas en C++ – Programación. (s.f.).
Programación. https://www.programacion.com.py/escritorio/c/listas-en-c

Arrays, arreglos y vectores en c++: USO Y sintaxis en C++ (no date) *ProgramarYa*. Available at: https://www.programarya.com/Cursos/C++/Estructuras-de-Datos/Arreglos-o-Vectores.