09 – TAD Lista (1 de x)

DOCENTE – FEDERICO VILENSKY

Definición

- Hasta ahora vimos la definición inductiva de las listas
- Habíamos definido a la lista de cualquier tipo a como
  - $[\,]: Lista\ a$
    $\_:\_: a \to Lista\ a \to Lista\ a$
- Con esta notación, lo que estamos haciendo es anotar los tipos de las operaciones
  - O sea, decimos que tipo recibe y que tipo produce
- De acá podemos sacar que las $Lista\ a$ son todos esos objetos que formamos combinando estas dos operaciones
- Lo que vamos a estar haciendo es ver a las listas como un TAD

Genericidad (Generics)

- La primer cosa interesante que vamos a ver es como la definición inductiva no nos da un tipo, sino que nos da un "generador de tipos"
  - $Lista\ a$, forma un nuevo tipo de listas para cada a
    - $Lista\ int$
    - $Lista\ char$
    - $Lista\ Persona$
- Para lograr esto en C++, vamos a usar templates
- Esto es algo que seguramente han usado sin saber lo que era, por ejemplo en java cuando creaban una lista
  - …
    ```
    List<int> valores = new ArrayList<int>();
    ```
    …

TADs y Clases abstractas

- Como vimos con la calculadora, los TADs los vamos a representar mediante clases abstractas
  - Recordemos:
    - Clase: Es una estructura que además de tener asociados datos, nos permite tener comportamiento asociado (métodos) y puede heredar su forma y comportamiento de otra clase existente
    - Clase abstracta: Es la definición que vamos a estar utilizando, pero no esta implementada. En la practica se ve como una clase que no podemos instanciar, pero podemos utilizarla
- Para hacer esto vamos a introducir un modulo que introduce al tipo abstracto Lista de cualquier tipo

TAD Lista

- ```
  template<class T>
  class Lista
  {
  public:
    <operaciones>
  }
  ```
- Qué va en esas esas <operaciones>?
  - Por lo menos van a ir las dos operaciones que teníamos en la definición inductiva
  ```

Constructores

- Lo primero que vamos a hacer entonces es definir un constructor para la lista vacía
- <operaciones>

```
//pos: la lista esta vacia
virtual void Vaciar( ) = 0;

//pos: inserta elem al principio de la lista
virtual void Insertar(T &elem)=0;
<…>
```

Predicados

- Nos interesa saber cuándo la lista está vacía, para ello vamos a usar un método
- Se va a dar todo el tiempo que debemos tener una función para cada constructor inductivo, y un método para chequear la forma del tipo inductivo
- Vamos entonces a hacer la operación EsVacia

EsVacia

- Esta función EsVacia es un predicado que vamos a usar para saber la forma de la lista
  - O bien es una lista vacía
  - O bien tiene elementos
- <operaciones>

```
// pos: retorna true sii la lista es vacia
virtual bool esVacia( ) = 0;
```

Operaciones Selectoras

- Por último, debemos poder acceder a los datos guardados en la lista
- Solo las listas no vacías van a tener información, y esta información va a ser el dato guardado en la cabeza, y la lista de la cola
- Vamos entonces a definirnos dos funciones para hacer esto
- A este tipo de funciones se les suele llamar operaciones **Selectoras-Destructoras**

Cabeza y Cola

- <operaciones>

```
// pre: la lista no esta vacia
// pos: retorna la cabeza de la lista
virtual T Cabeza( ) = 0; //podría retornar T*

// pre: la lista no esta vacia
// pos: elimina el primer elemento de la lista
virtual void Cola( ) = 0;
```

  <…>

- Con esto ya tenemos la especificación mínima necesaria para la lista

```
#ifndef LISTA_H
#define LISTA_H

template <class T>
class Lista
{
public:
  //pos: la lista esta vacia
  virtual void Vaciar( ) = 0;

  //pos: inserta elem al principio de la lista
  virtual void Insertar(T &elem)=0;

  // pos: retorna true sii la lista es vacia
  virtual bool esVacia( ) = 0;

  // pre: la lista no esta vacia
  // pos: retorna la cabeza de la lista
  virtual T Cabeza( ) = 0; //podría retornar T*

  // pre: la lista no esta vacia
  // pos: elimina el primer elemento de la lista
  virtual void Cola( ) = 0;
}

#endif //LISTA_H
```

Especificación Suficiente

- Este método de aplicar los constructores, predicados y selectores, lo podemos aplicar a todo tipo inductivo
- Si lo hacemos nos da una **especificación suficiente**
  - Cualquier otra operación que se nos ocurra, la podemos definir en base a las primitivas que acabamos de definir
- Ejemplo: largo

```
template <class T>
int largo(Lista<T> *l){
  if(l->esVacia()) return 0;
  else return 1+ largo(l->Cola());
}
```

OJO

- Es importante entender que NO existe un único TAD Lista, podemos tener distintas especificaciones, y todas serian un TAD Lista distinta
- Para empezar, los métodos asociados podrían llamarse distinto, esto sin embargo es un caso trivial y podemos decir que "son lo mismo"
- Pero mas interesante, podemos tener TAD Listas con distintas operaciones
  - Por ejemplo una Lista que:
    - Me permita insertar al final, o insertar en la posición n
    - Me indique si un elemento pertenece a la lista
    - Elimina un elemento dado de la lista
    - Elimina el elemento en la posición n de la lista

Implementación

- Como estuvimos viendo, para usar un TAD, vamos a tener una (o varias) **representaciones** y sus **implementaciones** correspondientes
- Para las listas que definimos antes, lo mas natural es implementarlas basándonos en la representación de **listas encadenadas** (que ya la estuvimos estudiando al comienzo del curso)
- Una implementación de un TAD la vamos a ver como la definición de una subclase concreta (i.e. no abstracta) de la clase abstracta donde definimos las operaciones del TAD
  - OJO, esto de clases abstractas y concretas, solo lo vamos a ver en POO, pero podemos tener TADs en otros paradigmas

Implementación Lista Encadenada

```
#ifndef LISTAENCADENADAIMP_H
#define LISTAENCADENADAIMP_H

#include “Lista.h”

template <class T>
class ListaEncadenadaImp: public Lista<T>
{
<valores>
<operaciones (metodos)>
}

#endif //LISTA_H
```

Representación de objetos

- Primero vamos a tener que introducir la estructura mediante la cual vamos a representar al objeto del tipo abstracto
- <valores>

```
private: // o protected:
  struct NodoLista{
    T elem;
    NodoLista *sig;
  };
  typedef NodoLista * PtrNodo;
  PtrNodo ini, posCorr;
  void BorrarLista();
<…>
```

Representación de objetos

- Private: al igual que en Java, private significa que solo podemos ver lo que definimos como private desde adentro de la clase
- Protected: al igual que en Java, protected significa que solo podemos ver lo que definimos como protected desde adentro de la clase o sublcases
- ini: indica el comienzo de la lista encadenada
- posCorr: indica la "cabeza actual" (posición corriente) de la lista, la vamos a usar para ir recorriendo a la lista con Cabeza y Cola

Metodos de ListaEncadenadaImp

- En el ".h"
- <operaciones (metodos)>

```
void Vacia( ) override;
void Insertar(T &elem) override;
bool esVacia( ) override;
T Cabeza( ) override;
void Cola( ) override;

//Constructor y Destructor de C++
ListaEncadenadaImp( ){ Vacia( ); }
~ListaEncadenadaImp( ){ BorrarLista( ); }
```

<…>

Implementación en .cpp

- En ".cpp"
- `#include "ListaEncadenadaImp.h"`

```
<implementación de operaciones (metodos)>
```

Constructores

```
<implementación de operaciones (metodos)>

template<class T>
void ListaEncadenadaImp<T>::Vacia()
{
  ini = nullptr;
  posCorr = nullptr;
}

template<class T>
void ListaEncadenadaImp<T>::Insertar(T &elem)
{
  PtrNodo aux = new NodoLista;
  aux->elem = x;
  aux->sig = ini;
  ini = aux;
  posCorr = aux;
}
<…>
```

Constructores

- La operación Vacia() setea a ini y a posCorr en nullptr;
- La operación Insertar(x) inserta el elemento x al principio de la lista, y nos deja a este nodo como la posición corriente
- Podríamos también agregar, por ejemplo, una función InsertarCorriente(x) que inserte al elemento x en el nodo siguiente a la posición corriente, dejando a la posición corriente como este nuevo nodo

Predicados y Selectoras

```
<implementación de operaciones (metodos)>
template <class T>
bool ListaEncadenadaImp::EsVacia()
{
  return posCorr == NULL;
}

template <class T>
T ListaEncadenadaImp::Cabeza()
{
  return posCorr->elem;
}

template <class T>
void ListaEncadenadaImp::Cola()
{
  posCorr = posCorr->sig;
}
<…>
```

Predicado y Selectoras

- La implementación de EsVacia() es trivial, con la salvedad de que estamos fijándonos en la posición corriente posCorr, y no en el inicio de la lista ini
- Cabeza() nos retorna el dato guardado en la posición corriente posCorr, no del inicio de la lista ini
- El método Cola() que modifica la posición corriente, asignándole el nodo siguiente como posición corriente, nos permite acceder a todos los elementos de la lista, sin tener que modificarla

Manejo de Precondiciones

- Tenemos varias opciones para el manejo de precondiciones
  - Asumir que la precondición se cumple. Pasándole la responsabilidad a quien utiliza el TAD
    - Esto puede hacer que lleguemos a estados inconsistentes o un segfault o errores similares que son difíciles de detectar la causa
  - Considerar casos erróneos
    - Podemos mediante un if chequear que se cumpla la condición. Y no hacer nada (o devolver un valor que indique la existencia de un error) en caso de que no se cumpla con la precondición

Manejo de Precondiciones

- Tirar una excepción/cortar la ejecución
  - En caso de que no se cumpla con la precondición, tiramos una excepción
  - Podemos hacer uso del macro assert, que recibe una expresión booleana
    - Si el valor de la expresión booleana es falso (0 o NULL) entonces imprime la línea y el archivo donde esto sucedió, y aborta la ejecución del programa
      - Para abortar la ejecución usa la función abort()
  - Propagación de error
    - Devolver un error (con un mensaje adecuado de error) sin cortar la ejecución
      - Esto no lo vamos a hacer en C++, pero en otros lenguajes es muy común (JavaScript, Go, Rust, Haskell, etc.)
- Estas dos opciones son las mejores, ya que nos permite identificar al problema

BorrarLista

```
template <class T>
void ListaEncadenadaImp<T>::BorrarLista()
{
  PtrNodo aux = ini;
  PtrNodo temp;
  while(aux != NULL)
  {
    temp = aux;
    aux = aux->sig;
    delete temp;
  }
  ini = NULL;
  posCorr = NULL;
}
```

Ejemplo de uso

```
#include <iostream>
#include “ListaEncadenadaImp.h”

int main()
{
  Lista<int> * l = new ListaEncadenadaImp<int>;
  for(int i = 0; i <= 5; i++)
  {
    l->Insertar(i);
  }
  while(! l->EsVacia())
  {
    std::cout << l->Cabeza() << std::endl;
    l->Cola();
  }
}
```