Recordemos

Tipos Abstractos de Datos (TAD)

Idea clave: Abstracción

Abstracción. Consiste en ignorar los detalles de la manera particular en que esta hecha una cosa, quedándonos solamente con su visión general.

-Un tipo de dato abstracto (TDA) o Tipo abstracto de datos (TAD) es un modelo matemático compuesto por una colección de operaciones definidas sobre un conjunto de datos para el modelo y un nombre que lo identifica.

Se define como un conjunto de valores que pueden tomar los datos de ese tipo, junto a las operaciones que los manipulan.

TAD = Nombre del TAD + valores (tipo de dato, dominio, etc.…) + operaciones.

Más TADs

El TAD Set

En el diseño de algoritmos, la noción de conjunto es usada como base para la formulación de tipos abstractos muy importantes.

Un conjunto es una colección de elementos (o miembros), los que a su vez pueden ellos mismos ser conjuntos o si no elementos primitivos. Todos los elementos de un conjunto son distintos, lo que implica que un conjunto no puede contener dos copias del mismo elemento.

Una notación usual para exhibir un conjunto es listar sus elementos de la siguiente forma: {1, 4}, que denota al conjunto cuyos elementos son los naturales 1 y 4. Es importante destacar que los conjuntos no son listas, el orden en que los elementos de un conjunto son listados no es relevante ({4, 1} , y también {1,4,1} denotan al mismo conjunto).

La relación fundamental en teoría de conjuntos es la de pertenencia, la que usualmente se denota con el símbolo $\in$. Es decir, a $\in$ A significa que a es un elemento del

conjunto A. El elemento a puede ser un elemento atómico u otro conjunto, pero A tiene que ser un conjunto.

Un conjunto particular es el conjunto vacío, usualmente denotado $\varnothing$, que no contiene elementos.

Las operaciones básicas sobre conjuntos son unión ($\cup$), intersección ($\cap$) y diferencia (/).

Módulo de definición para el TAD Set de enteros

```
#ifndef CONJUNTO1_H

#define CONJUNTO1_H


typedef struct Conjunto;

//contructoras

Conjunto* Crear();

Conjunto* agregarElemento( Conjunto* c, int n);

//predicados

bool esVacioConjunto(Conjunto* c);

bool pertenece(Conjunto* c, int n);

//auxiliares

Conjunto* interseccion(Conjunto* c1, Conjunto*c2);

Conjunto* unionC(Conjunto* c1, Conjunto*c2);

Conjunto* diferencia(Conjunto* c1, Conjunto*c2);

void imprimirConjunto(Conjunto* c);

//destructoras

void destruir(Conjunto* &c);
```

```
#endif
```

¿**Operaciones adicionales ?**

Por ejemplo imprimir elementos del conjunto…

Una implementación del TAD Set de enteros

Esta implementación es dinámica (la cardinalidad es potencialmente infinita si la memoria física lo permitiera), pero podríamos haber especificado el TAD de forma estática, dando en la especificación alguna constante que indicara el máximo de elementos que pudiera contener un conjunto (cardinalidad máxima). Así entonces décimos que es una implementación estática cuando el tamaño máximo TAD esta acotado. Por ejemplo en este caso la representación del TAD Set de enteros para una especificación Estática sería

```
struct Set{

   int tam;

   int elementos[MAX_TAM];//donde MAXTAQM se //encuentra declarada en       //Conjunto.h

};
```

Observar que agregamos algunas operaciones auxiliares para implementar las operaciones ofrecidas en la interfaz(especificación), estás operaciones auxiliares no son vistas por ningún modulo que utilice Conjunto ya que no se especificaron en el .h.

Observación en la representación que elegiremos (Lista de Enteros, donde se insertaran los elemento en orden), podemos optar por implementar algunas operaciones recursivas, dependiendo de cómo representemos la lista vacía.

En este caso modelamos la lista vacía con una celda dummy que apunta a Null, pero se podría haber modelado la lista vacía como NULL.

//Conjunto.c

```
#include "Conjunto.h"

//librerias que utilizo en la implementación del módulo

#include <cstdlib>

#include <iostream>
```

```
using namespace std;


struct Conjunto{
     int elem;
     Conjunto* sig;
};
//contructoras
Conjunto* Crear(){
  Conjunto* c=new Conjunto;//utilizo celda dummy
  c->sig=NULL;
  return c;
}


void agregarElementoAuxiliar( Conjunto*& c, int n){
       if(c==NULL){
          c=new Conjunto1;
          c->sig=NULL;
          c->elem=n;
       }else{
         if(c->elem>n){

             Conjunto1* nuevo=new Conjunto1;
             nuevo->elem=n;
             nuevo->sig=c;
             c=nuevo;
```

```
          }else{
             if(c->elem<n){
                agregarElementoAuxiliar(c->sig,n);
             }
             //si es igual no lo agrego
          }
        }

}


Conjunto* agregarElemento( Conjunto* c, int n){
          Conjunto * nuevo;
          agregarElementoAuxiliar(c->sig,n);
         nuevo=c;
         return nuevo;

}


//predicados
bool esVacioConjunto(Conjunto* c){
   return ( c->sig==NULL);
}
```

```
bool pertenece(Conjunto* c, int n){
    if(!esVacioConjunto(c)){
       if(c->sig->elem==n) return true;
       else if(c->sig->elem>n) return false;//poda
              else {
                c=c->sig;
               while(c!=NULL){
                       if((c->elem)>n) return false;
                       else{
                           if(c->elem==n) return true;
                            else c=c->sig;
                       }
                }//end while
                return false;
              }
    }else return false;
}
//auxiliares
Conjunto* interseccion(Conjunto* c1, Conjunto*c2){
   //Implementado de forma ineficiente
   if(esVacioConjunto(c1) || esVacioConjunto(c2))
      return Crear();
   else{
```

```
        c1=c1->sig;
        Conjunto* inter=Crear();
        while(c1!=NULL){
              if(pertenece(c2,c1->elem)){
                agregarElemento(inter,c1->elem);
              }
              c1=c1->sig;
        }
        return inter;
    }


}
Conjunto* unionC(Conjunto* c1, Conjunto*c2)
 {
    //Implementado de forma ineficiente
    if(esVacioConjunto(c1) && esVacioConjunto(c2))
       return Crear();
    else{
        Conjunto* unionc=Crear();


        if(!esVacioConjunto(c1)){
         c1=c1->sig;


         while(c1!=NULL){
```

```
            agregarElemento(unionc,c1->elem);
            c1=c1->sig;
        }

       }
       if(!esVacioConjunto(c2)){
        c2=c2->sig;

        while(c2!=NULL){
            //¿Porqué no verificamos si pertenece?
            agregarElemento(unionc,c2->elem);
            c2=c2->sig;
        }

       }
       return unionc;
   }
}
```

```
Conjunto* diferencia(Conjunto* c1, Conjunto*c2){

   //Implementado de forma ineficiente
   if(esVacioConjunto(c1))
```

```
      return Crear();
   else{
       Conjunto1* dif=Crear();
       c1=c1->sig;
       while(c1!=NULL){
             if(!pertenece(c2,c1->elem)){
               agregarElemento(dif,c1->elem);
             }
             c1=c1->sig;
       }
        return dif;
    }



}

void imprimirConjuntoAux(Conjunto1* c){
    if(!(c==NULL)){
          cout<< c->elem << endl;
          imprimirConjuntoAux(c->sig);
    }

}

void imprimirConjunto(Conjunto1* c){
```

```
    if(!esVacioConjunto(c)){
      imprimirConjuntoAux(c->sig);
    }
     cout<<"FIN"<<endl;
}



//destructoras
void destruir(Conjunto1* &c){
  if(c!=NULL && c->sig!=NULL) destruir(c->sig);
  if(c!=NULL){
     delete c;
     c=NULL;
  }
}
```

TAD Stack y Queue

El TAD Stack

Un *stack* (o pila) es una clase especial de lista en la que todas las inserciones y borrados de elementos se efectúan sobre uno de sus extremos, llamado el ***tope*** del stack. Otro nombre que se le da a este tipo de estructura es el de lista *LIFO* (last-in-first-out).

Ejemplos de stacks son:

- pila de fichas de poker en una mesa
- pila de platos para lavar
- pila de libros

donde es claramente conveniente quitar el elemento que está en el tope de la pila o agregar un elemento nuevo sobre el tope de la misma.

El TAD Stack incluye las siguientes operaciones:

`Null`
: Construye un stack vacío.

`Push`
: inserta un elemento en el tope del stack.

`Top`
: Retorna el elemento que se encuentra en el tope del stack.

`Pop`
: Retorna el stack al que se le ha quitado el tope.

`Empty`
: Verifica si el stack es vacío o no.

Módulo de definición para el TAD Stack

```
//Especificacion Minima del TAD pILA DE ENTEROS


/***** Constructoras *****/


Pila Null ();

/* retorna el stack vacío */


Pila Push (int i,Pila S );

/* retorna un stack con tope i y el resto del stack es S */


/***** Predicado *****/


bool Empty (Pila S);

/* retorna TRUE si S es vacío */


/***** Selectoras *****/
```

```
int Top (Pila S);

/* pre : S no es vacío retorna el tope del stack S */




Pila Pop (Pila S);

/* pre : S no es vacío retorna el stack resultado de borrar el tope

 de S */
```

Ejemplo. Verificación de Paréntesis Balanceados.

Dada una lista de caracteres que sólo puede contener los elementos (,), [,], { y }, deseamos construir un procedimiento que verifique que la expresión es balanceada.

Por ejemplo:

[()(())] es correcta
[(] es incorrecta

Especificación:

- Inicializar la pila vacía.
- Por cada símbolo de la entrada hacer lo siguiente:
  - Si el símbolo es de apertura => push simbolo en el stack
  - Si el símbolo es de cierre =>
    - Si el stack está vacío => expresión incorrecta; terminar
    - Si no está vacío:
      - Si el tope del stack es del mismo tipo que el símbolo de la entrada => pop en el stack, continuar inspección.
      - en otro caso => expresión incorrecta; terminar
- Si el stack es vacío => expresión correcta.
- Sino => expresión incorrecta.

Implementación en Modula2

```
PROCEDURE BalanExp (l : BS) : BOOLEAN
VAR
        Res : BOOLEAN;
        prim : CHAR;
        resto : BS;
         s     : STS;
PROCEDURE match(a,b: CHAR) : BOOLEAN;
(* retorna TRUE si a y b son apertura y cierre
   del mismo tipo *)
BEGIN
   RETURN (a="(" AND b = ")") OR
          (a="[" AND b = "]") OR
```

```
          (a="{" AND b = "}")
END match;

BEGIN
   (* inicializacion de variables *)
   s:= Null();
   ERROR:= FALSE;

  (* recorrida de la lista *)
   WHILE (NOT EsVacia(l)) AND NOT ERROR DO

      (* inspeccion del siguiente simbolo *)
      prim:= primero(l);
      CASE prim OF

       (* Apertura *)
           '(', '{' ,'[' :  s:= Push(prim,s);
      |
       (* cierre *)
           ')',']','}' :
                     IF Empty(s) THEN
                        ERROR:= TRUE;
                     ELSIF  match(prim,Top(s)) THEN
                        s:= Pop(s);
                     ELSE
                        ERROR:= TRUE
                     END; (* IF *)
      END; (* CASE *)

      (* continuar inspeccionando la lista *)
      l:= resto(l);
   END; (* WHILE *)

   (* no hubo error y no quedan simbolos en el stack *)
   RETURN (NOT ERROR AND Empty(s));
END BalanExp;
```

Implementación estática del TAD Stack

Cualquier implementación del TAD Lista de las que hemos visto es también una implementación válida del tipo Stack.
La implementación usando listas encadenadas es realmente simple, ya que Push y Pop operan solamente sobre el puntero al primer elemento de la lista.
La implementación estática más adecuada para este tipo abstracto es usar un arreglo con tope.
Sin embargo, el usuario de este módulo tiene que conocer el largo máximo del arreglo.

//Pila.c
#include “Pila.h”

//MAX está definido en el .h
//const MAX = ...; *(* alguna valor apropiado *)*

```
        struct Stack{

                T elems[MAX+1];//tope =0 pila vacía
                int tope;
        };

Stack* Null (){
        Stack*  s;
        S=new Stack;
        s->tope = 0;
        return s
};

Stack* Push ( T  i ,  Stack* s){


                if(s->tope<MAX-1){
                 s->tope = s->tope + 1;
                 s->elem[tope] = i;
                }
                return s;
}

bool Empty ( Stack S){
{
        reurn   S->tope== 0;
}

T  Top ( Stack S){
//Pre: la Pila no esta vacía
     S->elem[S->tope];
 }

Stack Pop ( Stack S){

        S->tope = S->tope - 1;
        return S;
}

.
```

Invocación de Procedimientos y Funciones

El algoritmo para chequear balance de símbolos sugiere una forma de implementar invocación de procedimientos y funciones. La diferencia en este caso es que cuando un procedimiento es invocado todas las variables locales al proceso invocador deben ser salvadas por el sistema ya que, entre otras razones, el procedimiento invocado puede sobreescribir las variables del invocador.

Cuando se invoca a un subprograma la siguiente información debe ser guardada:

- valores de los registros correspondientes a variables del subprograma invocador
- dirección de retorno, es decir, el lugar al que se transfiere el control una vez que el subprograma invocado termina de ejecutarse.

Todo este proceso puede claramente implementarse usando un stack. Cuando un subprograma es invocado, la información arriba descripta es salvada en una estructura (frame) y guardada en el tope del stack. Luego, el control se transfiere al subprograma invocado, el que puede libremente hacer uso de los registros para almacenar su información local. Si este subprograma a su vez invoca a otro, se repite este proceso. Cuando el subprograma invocado desea retornar el control se fija en el frame que se encuentra en el tope del stack, recompone los registros y luego transfiere el control a la dirección de retorno que fue almacenada.

El TAD Queue

Una ***queue*** (cola*)* es otra clase especial de lista, donde los elementos son insertados en un extremo (el final de la cola) y borrados en el otro extremo (el frente de la cola). Otro nombre usado para este tipo abstracto es el de lista ***FIFO*** (first-in-first-out).

Las operaciones para una cola son análogas a las de stack, la diferencia sustancial es que las inserciones son efectuadas al final de la lista. La terminología tradicional para stacks y colas es también diferente.

El TAD Queue incluye las siguientes operaciones:

- la operación que construye una cola vacía, `Null`.
- una operación, `Enqueue`, que inserta un elemento al final de la cola.
- la operación `Front`, que retorna el elemento que se encuentra en el comienzo de la cola.
- la operación `Dequeue`, que borra el primer elemento de la cola.
- y finalmente, `Empty`, que testea si la cola es vacía o no.

Módulo de definición para el TAD Queue

```
typedef struct PrimUlt* Queue;
```

typedef Queue Cola;

/*** Constructoras *****/**
Cola ColaVacia ();
/* retorna la cola vacía */

Cola Enqueue (int i,Cola C);

/* inserta el elemento i al final de la cola C */

/*** Predicado ******/**

bool Empty (Cola C);
/* retorna TRUE si C es vacía */

/*** Selectoras *****/**

int Front (Cola C);
/* pre : C no es vacía retorna el elemento que se encuentra al comienzo de la cola C */

Cola Dequeue (Cola C);
/* pre : C no es vacía
Post: retorna la cola resultado de borrar el primer elemento de C */

Observar que esta última operación no dice nada acerca de si el borrado implica la liberación del espacio físico en memoria. ¿Por qué?

Implementación del TAD Queue

Al igual que para stacks, una implementación adecuada para este tipo abstracto es usar listas encadenadas.

Sin embargo, en este caso se puede aprovechar el hecho de que todas las inserciones son efectuadas al final de la cola e implementar la operación `Enqueue` en forma eficiente. En vez de recorrer toda la lista cada vez que queremos insertar un nuevo elemento, lo que se hace es mantener un puntero al último elemento.
Al igual que para listas, también mantendremos un puntero `front` al comienzo de la misma. Para colas, este puntero permitirá implementar `Front` y `Dequeue` en forma eficiente.
Se mantendrá además una celda *"dummy"* como cabeza de la lista, el puntero `front` apuntará a esta celda. Esta convención permitirá hacer un manejo adecuado de la cola vacía.
A continuación definiremos un módulo de implementación para el tipo Queue basado en los conceptos descriptos arriba.

```
//IMPLEMENTACIÓN queues;
/* Implementacion del TAD QEUEE utilizando una estructura enlazada */
#include "Cola.h"
#include <stdlib.h>

//Representación del TAD

        struct NCelda* Celda;

        /* definicion de la celda básica de la lista */

        struct NCelda{
               T elem ;    /* información */
               Celda sig;  /* puntero al siguiente */
```

```
        };


        struct PrimUlt{
                Celda  front; /* puntero a la primera celda */
                Celda  rar;/* puntero a la ultima celda */

        };

//
Queue Null () {
/* retorna la Queue vacia */

        Queue q;
        /* se crea la celda cabezal  */
        q= new  PrimUlt;
        q->front= new NCelda;
        q->front->sig=NULL;
        q->rear = q->front;
        return q;
}

Queue Enqueue(T i,  Queue C){
/* agrega un elemento al final */

        Queue q=Null();
        C->rear->sig = new  NCelda;
        q->rear = C->rear->sig;
        q->rear->elem = i;
        q->rear->sig = NULL;
        q->front = C->front;
        return q;
}

bool Empty (Queue C){
/* retorna True sii la Queue es vacia */
        return (C->front == C->rear)
}



T Front (Queue C){
/* Retorna el primer elemento de la lista
   Se supone el tipo T puede ser devuelto por una función
*/
        return C->front->sig->elem;
}
Queue Dequeue (Queue C){
/* Borra el primer elemento de la lista */
//pre: la cola tien por lo menos un elemento
        Queue q;

        q= Null();
        q->front = C->front->sig;
        q->rear = C->rear;
        return q;
}
```

Implementación estática del TAD Queue

La representación que hemos visto para stack haciendo uso de un arreglo con tope puede ser usada para implementar el tipo Queue. Sin embargo, esta representación no es muy eficiente. La operación de agregar un elemento a la cola se puede ejecutar eficientemente: simplemente se incrementa el tope y en esa posición se da de alta al elemento. Pero dar de baja al primer elemento de la cola requiere desplazar todos los elementos una posición en el arreglo. Para solucionar este problema debemos tomar un enfoque distinto: un arreglo circular. Piense en un arreglo como un círculo, donde la primer posición del mismo "sigue" a la última. La cola se encontrará alrededor del círculo en posiciones consecutivas (en un sentido circular). Para insertar un elemento movemos el puntero al último una posición en el sentido de las agujas del reloj y en esa posición insertamos al elemento. Para dar de baja al primer elemento simplemente movemos al puntero primero una posición en el mismo sentido. De esta forma le ejecución de estas operaciones insume el mismo tiempo independientemente de la cantidad de elementos que componen a la estructura.

Esta representación tiene una desventaja: es imposible distinguir una cola vacía de una que ocupa el arreglo entero. Solución: un indicador booleano que sólo se haga verdadero cuando la cola es vacía. Otra es prevenir que la cola no ocupe todo el arreglo o simplemente, llevar un contador de los elementos de la cola.