Libro digital
Estructuras de Datos

José Sánchez Juárez
Cesar Román Martinez García

5 de diciembre de 2013

Índice general

Capítulo 1

Introducción

Todos los sucesos de la vida real se tienen que medir y por lo tanto se hacen cálculos para saber de su transformación en otros sucesos, por lo que estas transformaciones requieren de operar sobre entidades que tengan como característica una medida. Una de estas entidades son los tipos de datos. En especial los tipos de datos básicos. Otros tipos de datos que también tienen esta característica son la combinación de los tipos de datos básicos, llamadas estructuras de datos. En este libro se tratarán con algún detalle las estructuras de datos. Para comenzar el estudio de las estructuras de datos se abordarán los siguientes temas:

1. Presentar una semblanza de lo que se abordará en el curso, así como la forma de calificar.

2. Concepto de tipos de datos.

3. Abstracción.

4. Concepto de abastracción de datos.

1.1. Objetivo

Desarrollar programas en pseudocódigo para resolver problemas de la vida real con tendencias a la Ingeniería, aplicando las estructuras de datos.

1.2. Semblanza del curso

El curso es del tipo teórico y práctico, por lo que se evaluará con tres exámenes teóricos y con la entrega de 12 prácticas. El extraordinario se evaluará con un proyecto. Las prácticas se harán en pseudocódigo, la programación en lenguaje como C, C++ y Java se encargará a un grupo de alumnos los que presentarán en clase los programas realizados que valdrá 2 puntos sobre la calificación parcial, por lo que se hará tres veces durante el semestre, este tercer punto valdrá 20 %.

Se presentará la información adicional al curso en el grupo de Yahoo "jsjuarez", tal como prácticas, temario del curso y diapositivas. También se dará una semblanza de los datos básicos usados en el mundo real mediante ejemplos.

1.3. Concepto de tipos de datos

Un programa de computadora es un conjunto auto-contenido de instrucciones para operar una computadora que produce un resultado específico. Para realizar las instrucciones se utilizan los datos, los cuales son de diferentes tipos que para poder aplicarlos en la computadora se requiere que estos se representen de forma binaria.

Para evaluar la información por medio de un programa se usan variables las cuales se colocan al principio de un bloque de un programa, estos son los tipos de datos donde se declara una lista de variables que se marcan como tipos de datos, al marcar un tipo de dato esto hace que cada una de las variables tengan propiedades. Ya que si se conocen las propiedades de los datos se pueden manipular para ingresarlos y obtenerlos a la salida de los programas.

Los tipos de datos que se permiten van de acuerdo a ANSI (American National Standards Institute) que se encarga de supervisar el desarrollo de estándares para productos, servicios, procesos y sistemas en los Estados Unidos y que pertenece a la Organización Internacional para la Estandarización (ISO). ASCII(American Standard Code for Information Interchange) que se traduce al Español como Código Estándar Estadounidense para el Intercambio de Información. Es un código de caracteres basado en el alfabeto latino, tal como se usa en inglés moderno y en otras lenguas occidentales. Fue creado en 1963 por el Comité Estadounidense de Estándares (ASA, conocido desde

1969 como el Instituto Estadounidense de Estándares Nacionales, o ANSI) como una refundición o evolución de los conjuntos de códigos utilizados entonces en telegrafía. Más tarde, en 1967, se incluyeron las minúsculas, y se redefinieron algunos códigos de control para formar el código conocido como US-ASCII.

El código ASCII utiliza 7 bits para representar los caracteres, aunque inicialmente empleaba un bit adicional (bit de paridad) que se usaba para detectar errores en la transmisión de los datos. A menudo se llama incorrectamente ASCII a otros códigos con caracteres de 8 bits, como el estándar ISO-8859-1 que es una extensión que utiliza 8 bits para proporcionar caracteres adicionales usados en idiomas distintos al inglés, como el español.

ASCII fue publicado como estándar por primera vez en 1967 y fue actualizado por última vez en 1986. En la actualidad define códigos para 32 caracteres no imprimibles, de los cuales la mayoría son caracteres de control obsoletos que tienen efecto sobre cómo se procesa el texto, más otros 95 caracteres imprimibles que les siguen en la numeración (empezando por el carácter espacio).

Casi todos los sistemas informáticos actuales utilizan el código ASCII o una extensión compatible para representar textos y para el control de dispositivos que manejan texto como el teclado. No deben confundirse los códigos ALT+número de teclado con los códigos ASCII.

1.3.1. Representación binaria

La representación binaria de los datos se usa en el lenguaje de máquina, todos los tipos de datos tienen su representación binaria por lo que cada tipo de dato tiene un tamaño de espacio en la memoria de la computadora, así que : Los tipos de datos determinan cuanto espacio de almacenamiento se debe permitir junto con los tipos de operaciones permitidas [2].

Un ejemplo de como se almacenan los enteros en forma binaria es el Número = 13:

La representación decimal = 1*101 + 3*100.
La representación binaria = 1101 = 1*23 + 1*22 + 0*21 + 1*1.

Cada 1 or 0 es llamando un bit, por lo que el número 13 requiere de 4 bits. Ahora para representar -127 en forma binaria se procede de la siguiente manera:

1. Se convierte el 127 a forma binaria.
2. El complemeto de cada bit se hace como sigue: se pone un 0 si hay un 1 y si hay un 0 se pone un 1. De modo que se tiene 1000 0000.
3. Se suma 1 al número del punto anterior: 1000 0000 + 1 = 1000 0001 (-127).

Así que la representación binaria del número -127 es 1000 0001.

1.3.2. Los tipos de datos

Los tipos de datos se dividen en familias [10]. Las cuales son: la familia de enteros y la familia de flotantes. La familia de enteros sirve para representar a los números enteros $\mathbb{Z}$ que son los números naturales $\mathbb{N}$ y los números negativos. La familia de flotantes sirve para representar a los números reales $\mathbb{R}$ que abarcan a los números irracionales y los racionales $\mathbb{Q}$.

Para declarar los datos en pseudocódigo se hace de la siguiente manera:

Algorithm 1.1: Datos básicos

```
1 TIPE i, j, big:ENTERO;
2 TIPE car:CARACTER;
3 TIPE n:REAL;
4 TIPE ap:APUNTADOR;
5 TIPE ap:↑ ap;
  Data: ↑ a_p , ↑ a_p1 .
  Result: ↑ p_1
6 i, j ← {0, 1, 2, 3, ...}
7 car ← {a, ..., z, A..., Z}
8 n ← {..., −3, −2, −1, 0..., 1, 2, 3, ...}
```

Que expresados en algún leguaje de programación se hace de la siguiente manera:

```
Integer family
   char data type
   int data type
   short int data type
   long int data type
```

Y de la siguiente manera:

```
Float family (número real con punto decimal)
   Float data type
   Double data type
```

En los caracteres con signo se tiene el siguiente intervalo:

-128 a $+127$, i.e. $(-2^8\ a\ 2^8 - 1)$. Para los otros tipos de datos se presentan a continuación.

Carácter con signo	1 byte	$-2^7\ a\ 2^7 - 1$ (-128 a 127)
Carácter sin signo	1 byte	$0\ a\ 28 - 1$ (0 a 255)
short	2 byte	$-2^{15}\ a\ 2^{15} - 1$ (-32768 a 32767)
short sin signo	2 byte	$0\ a\ 2^{16} - 1$ (0 a 65535)
long entero	4 byte	$2^{31}\ a\ 2^{31} - 1$ (2,147,483,648 a 2,147,483,647)
entero	2 o 4 byte	

Precisamente los tipos de datos se usan para conocer el sobre-flujo de ellos y no cometer errores como se muestra en el siguiente programa:

```
 // el programa da valores maximos y minimos de el tipo de dato
#include <stdio.h>
main()
{
char i,j ;
i = 1;
while (i > 0) // A
{
j = i; // B
i++; // C
}
printf ("el valor maximo del char es %d\n",j);
printf ("el valor de char despues del sobreflujo es %d\n",i);
}
```

El tipo char:

Si se declara a c como un carácter de la siguiente forma char c; luego se puede asignar el caracter 'A' como sigue:

```
char c;
c = 'A';
```

```
c = 65;
c = '\x41';   // Representanción Hexadecimal
c = '\101';   // Representación Octal
```

Se puede escribir c = 'A' porque 'A' representa una cadena. La siguiente lista de comandos son para poder dar a la escritura de los programas el formato.

```
\a alert (bell) character \\backslash
\b backspace              \? question mark
\f form feed              \' single quote
\n new line               \" double quote
\r carriage return        \ooo octal number
\t horizontal tab         \xhh hexadecimal number
\v vertical tab
```

Para hacer la conversión de tipos, se hace el siguiente programa:

```
 #include <stdio.h>
  main()
  {
  double d1 = 123.56;
\\ A
  int i1 = 456;
\\ B

  printf("el valor de d1 como int sin el
  operador cast %d\n",d1);
\\ C
  printf("el valor de d1 como int con el
  operador cast %d\n",(int)d1);
\\ D
  printf("el valor de i1 como double sin el
  operador cast %f\n",i1);
\\ E
  printf("el valor de i1 como double con el
  operador cast %f\n",(double)i1);
\\ F
```

```
   i1 = 10;
   printf("efecto de operador unario
   múltiple %f\n",(double)++i1);
\\ G
   i1 = 10;
\\ H
   //printf("efecto de operador unario
   múltiple %f\n",(double) ++ -i1);
   error
\\ I
   i1 = 10;
   printf("efecto de operador unario
   múltiple %f\n",(double)- ++i1);
\\ J
   i1 = 10;
\\ K
   printf("efecto de operador unario
   múltiple %f\n",(double)- -i1);
\\ L
   i1 = 10;
\\ M
   printf("efecto de operador unario
   múltiple %f\n",(double)-i1++);
\\ N

   }
```

1.4. Abstracción

El universo esta lleno de sistemas complejos. Se aprende de estos sistemas por medio de modelos. Los modelos pueden ser matemáticos, tales como las ecuaciones que describen el movimiento de los satélites al rededor de la tierra. Un objeto físico como un tipo de aeroplano que se usa en un túnel de viento, es otro tipo de modelo. Lo que se busca del modelo es recopilar las características más significativas. Dejando de lado las características irrelevantes. La abstracción es el modelo del sistema que incluye solamente las características esenciales.

1.5. Concepto de abstracción de datos

Se define como la separación de las propiedades lógicas de los tipos de datos de su implementación. Los datos son los sustantivos de un programa. En el mundo real los objetos son manipulados, en un programa de computadora la información es procesada. Es más confortable ver el mundo real que la abstracción del mismo. Sin embargo cuando se habla de un entero es fácil usarlo pues lo hemos convertido en una generalidad ya que todo el tiempo lo aplicamos para realizar programas. Y un entero es una abstracción de un tipo de dato. Como ejemplo la siguiente operación es una abstracción de datos:

```
resultado := a + b;
```

Puesto que usa un modelo de datos, donde solamente se utilizan las características más relevantes. Este modelo es ideal para realizar operaciones utilizando un programa de computadora donde se aplica un lenguaje.

1.6. Conceptos fundamentales

Los datos del mundo real son muchos y la mayoría son inecesarios, la limitante de la computadora es que no puede considerar todos los datos que representan a un objeto del mundo real, sin embargo se hacen consideraciones de tal manera que con el menor número de datos se pueda representar en la computadora el objeto del mundo real, estas consideraciones tienen por nombre abstracción de los datos. La abstracción de los datos es la consideración de las características más importantes llamadas características primarias, estas características primarias se pueden representar por medio de conjuntos de datos, los datos adecuados se usan de acuerdo al problema a resolver. La forma de seleccionar los tipos de datos se hace siguiendo dos pasos. El primero es seleccionar el tipo de dato que representa la abstracción de la realidad adecuadamente, el segundo es la representación de la información. Los tipos de datos representan características como en matemáticas los números reales, complejos, o variables lógicas. O variables que representan valores individuales o conjuntos de valores o conjuntos de conjuntos de valores. O funciones, funcionales o conjuntos de funciones.

1.6.1. Concepto de función

Una función es una herramienta de abstracción para separar su comportamiento físico de su implementación. La separación es importante por dos razones: Es fácil de diseñar programas usando descomposición funcional (descomponer un gran problema en pequeños subproblemas, cada uno expresado como una llamada a función).

La separación entre el comportamiento físico y la implementación alienta la creación del software reusable (piezas de software que pueden usar en diferentes aplicaciones). Ejemplos familiares son librerias de funciones *sqrt* y *abs* para calcular la raíz cuadrada y el valor absoluto.

Las funciones caen en dos categorías: aquellas que regresan un sólo valor de función y aquellas que no (procedimientos) [6]. En esta cita se habla de efectos colaterales, como la modificación de variables en la lista de parámetros o la modificación de variables globales. Al diseñar e implementar las funciones se requiere poner atención cuidadosa al flujo de datos de la lista de parámetros. El flujo de datos es el flujo de la información del llamador hacia la función y de la función hacia el llamador, para cada parámetro de la lista de parámetros de la función, hay tres posibilidades: un camino de datos dentro de la función, un camino fuera de la función, o dos flujos dentro y fuera de la función. La dirección del flujo de datos determina el método de pasar los parámetros.

Las clases son métodos que vinculan los datos con las funciones las cuales operan en ellos, las clases son tipos de datos definidos por el usuario [7] -p28.

Dividir un programa en funciones es uno de los mayores principios del descenso - p77, en programación estructurada. En un programa existen la declaración de la función, la llamada de la función y la definición de la función.

El prototipo de la declaración de la función es:

```
type nombre_de_la_función(lista de argumentos);
```

Ejemplo de la declaración de la función, donde la lista de argumentos se declara con tipo y nombre, y cada argumento se debe declarar independientemente, de la siguiente manera:

```
float volumen(int x, float y, float z);
```

Cuando la función se usa en la modalidad de llamada de una función no se consideran los nombres de las variables, ya que el compilador solamente

verifica el tipo de argumento. Un ejemplo de llamada a función es la siguiente: -p80.

```
float volumen(int, float, float);
```

Si los nombres de las variables se usan no tienen que ser acoplados con la llamada de una función o con la definición de una función.

En la definición de la función es necesario declarar la lista de argumentos con tipo y nombre, de la siguiente manera:

```
float volumen(int a, float b, float c)
{
float v = a*b*c;
}
```

La función *volumen*() puede ser invocada de la siguiente forma:

```
float cubo1 = volumen(b1, w1, h1);
```

Con la condición de que las variablas $b1$, $w1$ y $h1$ se declaren con anterioridad. Pero si la función se declara con la lista de argumentos vacía esta función no pasa los parametros y es igual a:

```
void desplegar(void);
```

Sin embargo en C un parentesis vacio implica cualquier número de argumentos. Una función en C++ también es una lista de parametros abierta:

```
void hacer_algo(...);
```

El flujo de datos dentro de la función, el llamador pasa una copia del parámetro en uso a la función y no quiere que la función modifique el parámetro en uso. La función inspeccionará pero no modificará el parámetro en uso del llamador. Este es llamado el paso de los argumentos por valores.

En forma tradicional las funciones pasan los argumentos por valor. La llamada a función crea un nuevo conjunto de variables y copia los valores de los argumentos dentro de ellos. La función no tiene acceso a las variables actuales en el programa que llama y solamente trabaja en las copias de los valores. Un ejemplo es el siguiente:

```
int Trans(int someInt)
{
return 2 * someInt + 5;
}
```

Otra forma de transferir los datos dentro de la función, aunque en esta si se modifique al parámetro formal pero no al parámetro en uso:

```
int Trans(int someInt)
{
someInt = 2 * someInt + 5;
return someInt;
}
```

La forma de transferir los datos fuera de la función, el llamador pasa la localidad o dirección del parámetro en uso a la función y quiere que la función modifique el parámetro en uso. La función no inspeccionará el valor en uso del parámetro de entrada pero lo modificará. Esto es llamado el paso de argumentos por referencia. Como en el siguiente ejemplo:

```
void inicializar(float& delta, float& epsilon)
{
delta = 0.1;
epsilon = 0.002;
}
```

Si se requiere cambiar los valores en la llamada al programa. Esto se muestra en el ordenamiento de la burbuja donde se necesita comparar dos elementos adiacentes en la lista e intercambiar sus valores si el primer elemento es más grande que el segundo. Si una función es usada para ordenar por burbuja, esta debe alterar los valores de las variables de la llamada de la función, lo cual no es posible si se usa el método "llamada por valor".

Para esto se usan las variables referencias que se declaran de la manera siguiente:

```
tipo_dato & nombre_referencia = nombre_variable;
```

Una variable referencia da un nombre alias a una variable definida previamente, como por ejemplo:

```
float total = 1000;
float & sum = total;
```

La variable definida previamente es *total* y el alias de esta variable ahora es *sum*.

Este mismo método se puede aplicar a una llamada de función por referencia como en el siguiente ejemplo:

```
void intercambiar(int &a, int &b)
{
int t = a;
a = b;
b = t;
}
```

Aquí otro ejemplo:

```
void f(int & x)
{
x = x + 10;
}

int main()
{
int m = 10;
f(m);
....
....
}
```

Cuando se llama a la función f(m), la siguiente inicialización ocurre:

```
int & x = m;
```

Dos formas entrada y salida de la función, el llamador pasa la localidad o dirección del parámetro en uso a la función y quiere que la función use el parámetro en uso y luego posiblemente lo modifique. La función inspeccionará el valor en uso del parámetro de entrada y luego posiblemente lo modifique. Ejemplo:

```
void Update(int& alpha, int& beta)
{
alpha = 3 * alpha;
beta++;
}
```

Para acceder a las variables existe el operador de resolución de ámbito que se representa por ::. Con este operador se tiene acceso a las variables declaradas por bloque. Y se usa de la siguiente forma:

```
:: nombre_variable
```

Se tiene el siguiente ejemplo:

```
#include <iostream>

using namespace std;
int m = 10;

int main()
{
    int m = 20;
    {
         int k = m;
         int m = 30;

         cout << "Estamos en el bloque interno \n";
         cout << "k =" << k << "\n";
         cout << "m = " << m << "\n";
         cout << "::m =" << ::m << "\n";
    }

    cout << "Estamos en el bloque externo \n";
    cout << "m =" << m << "\n";
    cout << "::m =" << ::m << "\n";

    return 0;
}
```

Correr el siguiente programa:

```
#include<iostream>
typedef int Boolean;    // Definir identificador
                        // Booleano como sinonimo para int
const int TRUE = 1 :
const int FALSE = 0 :

void Printvalue( int , char , int ) :     // Funcion prototipo
void Convert( const char[ ] , int& , Boolean& > :

int main()
{
char strl[31];   // Primer cadena de numero Romano
char str2[31];   // Segunda cadena de numeroRomano
char op;   // Operador con expresion
int decl;  // Valor decimal del primer operando
int dec2;    //  "      " "  segundo  "
Boolean strlvalid;     //primera cadena valida?
Boolean str2Valid;     //Segunda cadena valida?
Boolean opValid;     //Operador valido?


cout << "Expresar numeral Romano:\n";
while ( cin >> strl ) {
    cin >> op >> str2;
    Convert(str1, decl, strlVa i d );
    Convert( str2 , dec2, str2Va i d );
    opValid = (op=='+' || op== '-' || op=='*' || op=='/' );
    if( strlValid && str2Valid && opValid)
    PrintValue(dec1, op, dec2);
    else
       cout << " MAL DATO\n\n";
    cout << "Expresar numeral Romano: \n " ;
    }
return 0;
}
```

1.6.2. Algoritmos

La sintaxis para declarar un algoritmo es la siguiente:

```
Algoritmo Nombre(< Lista de parámetros >)
```

Así uno de los algoritmos es encontrar y regresar el número máximo de una serie n de números dados. El pseudocódigo es el siguiente:

```
Algoritmo Maximo(A, n)
 Begin
   max := A[1];
   FOR i := 2 TO n DO
     IF A[i] > max THEN
        max := A[i];
     END IF
   END FOR
   RETURN max;
 END ALGORITMO
```

Ordenar de manera ascendente una colección de elementos de $n \geq 1$ de tipo arbitrario, una manera de presentar el algoritmo, es:

1. Examinar los elementos de a[1] hasta a[n].
2. Suponer al elemento más pequeño como a[j].
3. Intercambiar a[i] por a[j].

La otra forma de presentar al algoritmo es por medio del pseudocódigo:

```
Algoritmo Ordenamiento(a, n)

 Begin
   FOR i := 1 TO n DO
      j := i;
      FOR k := i + 1 TO n DO
         IF ( a[k] < a[j] ) THEN
              j := k;
         END IF
```

```
      t := a[i];
      a[i] := a[j];
      a[j] := t;
      END FOR
   END FOR
END ALGORITMO
```

Este algoritmo tiene sus limitaciones, debido a que hay que aplicarlo varias veces para hacer el ordenamiento ascendente.

1.6.3. Arreglos

La representación de los números decimales se hace por medio de la base diez, de la siguiente manera:

$$d_0 + d_1 * 10^{-1} + \cdots + d_n * 10^{-n}$$

Lo que se puede sintetizar como:

$$\sum_{i=0}^{n} d_i * 10^{-i}$$

Si se requiere representar una serie de potencias negativas del número dos, expresado de la siguiente manera:

$$2^{-i}$$

Si se quiere expresar las potencias negativas de dos para $i = 1, 2, \cdots, n$, se escribe para 1 como $2^{-1} = \frac{1}{2} = 0{,}5$, sin embargo esto es una división decimal, para convertirla en una división entera se multiplica el numerador por diez de la siguiente forma: $\frac{1*10}{2} = 5$. Ahora para 2 se escribe como $2^{-2} = \frac{1}{2*2} = \frac{0{,}5}{2} = 0{,}25$ para convertirla en una división entera se escribe como: $\frac{\frac{1*10}{2}*10}{2} = 25$. Si se quiere calcular la potencia de 3 se escribe como $2^{-3} = \frac{1}{2*2*2} = \frac{0{,}25}{2} = 0{,}125$ para convertirla en una división entera $\frac{\frac{\frac{1*10}{2}*10}{2}*10}{2} = 125$. Así si inducimos el enesimo término 2^{-n}, quiere decir que lo tenemos que multiplicar por 10^n y lo debemos dividir por 2 n veces.

Ahora si usamos arreglos, lo debemos hacer de la siguiente manera:

$$r := A[i] + r * 10$$

y

$$A[i] := r \ DIV \ 2$$

y

$$r := r - 2 * A[i]$$

Si el exponente es 1 o 2^{-1}, lo que quiere decir que $i := 1$ en el arreglo se almacena de la siguiente manera:

$$r := r * 10 + A[1] = 1 * 10 + 0 = 10$$

Así que el valor del arreglo en la posición 1 es:

$$A[1] := 10 \ DIV \ 2 = 5$$

El residuo para este caso es:

$$r := r - 2 * A[1] = 10 - 2 * 5 = 10 - 10 = 0$$

El valor almacenado en el arreglo es:

$$A[1] := 5$$

Si el exponente es 2 o 2^{-2}, lo que quiere decir que $i := 2$ en el arreglo se almacena de la siguiente manera, considerando el cálculo anterior de $A[1] := 5$:

$$r := r * 10 + A[1] = 0 * 10 + 5 = 5$$

Así que el valor del arreglo en la posición 1 es:

$$A[1] := 5 \ DIV \ 2 = 2$$

El residuo para este caso es:

$$r := r - 2 * A[1] = 5 - 2 * 2 = 5 - 4 = 1$$

El valor nuevo almacenado ahora en la posición 1 del arreglo es:

$$A[1] := 2$$

Ya que el exponente llega a 2 debe haber dos posiciones en el arreglo, así que considerando los valores calculados en el paso anterior se hace el almacenamiento en el arreglo como sigue:

$$r := r * 10 + A[2] = 1 * 10 + 2 = 10$$

Así que el valor del arreglo en la posición 2 es:

$$A[2] := 10 \ DIV \ 2 = 5$$

El residuo para este caso es:

$$r := r - 2 * A[2] = 10 - 2 * 5 = 10 - 10 = 0$$

Los valores nuevos almacenados ahora en las posiciones 1 y 2 del arreglo son:

$$A[1] := 2$$

y

$$A[2] := 5$$

Si el exponente es 3 o 2^{-3}, lo que quiere decir que $i := 3$ en el arreglo se almacena de la siguiente manera, considerando el cálculo anterior de $A[1] := 2$ y $A[2] := 5$:

$$r := r * 10 + A[1] = 0 * 10 + 2 = 2$$

Así que el valor del arreglo en la posición 1 es:

$$A[1] := 2 \ DIV \ 2 = 1$$

El residuo para este caso es:

$$r := 2 - 2 * A[1] = 2 - 2 * 1 = 2 - 2 = 0$$

El valor nuevo almacenado ahora en la posición 1 del arreglo es:

$$A[1] := 1$$

Ya que el exponente llega a 3 debe haber tres posiciones en el arreglo, así que considerando los valores calculados en el paso anterior se hace el almacenamiento en el arreglo como sigue:

$$r := r * 10 + A[2] = 0 * 10 + 5 = 5$$

Así que el valor del arreglo en la posición 2 es:

$$A[2] := 5 \ DIV \ 2 = 2$$

El residuo para este caso es:

$$r := 5 - 2 * A[2] = 5 - 2 * 2 = 5 - 4 = 1$$

Así que para la posición 3 del arreglo los valores que se almacenan se calculan de la siguiente manera:

$$r := r * 10 + A[3] = 1 * 10 + 0 = 10$$

Así que el valor del arreglo en la posición 3 es:

$$A[3] := 10 \ DIV \ 2 = 5$$

El residuo para este caso es:

$$r := 10 - 2 * A[3] = 10 - 2 * 5 = 10 - 10 = 0$$

Los valores nuevos almacenados ahora en las posiciones 1, 2 y 3 del arreglo son:

$$A[1] := 1$$

$$A[2] := 2$$

y

$$A[3] := 5$$

Si el exponente es 4 o 2^{-4}, lo que quiere decir que $i := 4$ en el arreglo se almacena de la siguiente manera, considerando el cálculo anterior de $A[1] := 1$, $A[2] := 2$ y $A[3] := 5$:

$$r := r * 10 + A[1] = 0 * 10 + 1 = 1$$

Así que el valor del arreglo en la posición 1 es:

$$A[1] := 1 \ DIV \ 2 = 0$$

El residuo para este caso es:

$$r := 1 - 2 * A[1] = 1 - 2 * 0 = 1 - 0 = 1$$

El valor nuevo almacenado ahora en la posición 1 del arreglo es:

$$A[1] := 0$$

Ya que el exponente llega a 4 debe haber cuatro posiciones en el arreglo, así que considerando los valores calculados en el paso anterior se hace el almacenamiento en el arreglo como sigue:

$$r := r * 10 + A[2] = 1 * 10 + 2 = 12$$

Así que el valor del arreglo en la posición 2 es:

$$A[2] := 12 \ DIV \ 2 = 6$$

El residuo para este caso es:

$$r := 12 - 2 * A[2] = 12 - 2 * 6 = 12 - 12 = 0$$

Así que para la posición 3 del arreglo los valores que se almacenan se calculan de la siguiente manera:

$$r := r * 10 + A[3] = 0 * 10 + 5 = 5$$

Así que el valor del arreglo en la posición 3 es:

$$A[3] := 5 \ DIV \ 2 = 2$$

El residuo para este caso es:

$$r := 5 - 2 * A[3] = 5 - 2 * 2 = 5 - 4 = 1$$

Así que para la posición 4 del arreglo los valores que se almacenan se calculan de la siguiente manera:

$$r := r * 10 + A[4] = 1 * 10 + 0 = 10$$

Así que el valor del arreglo en la posición 4 es:

$$A[4] := 10 \ DIV \ 2 = 5$$

El residuo para este caso es:

$$r := 10 - 2 * A[4] = 10 - 2 * 5 = 10 - 10 = 0$$

Los valores nuevos almacenados ahora en las posiciones 1, 2, 3 y 4 del arreglo son:

$$A[1] := 0$$

$$A[2] := 6$$

$$A[3] := 2$$

y

$$A[4] := 5$$

Se podría calcular hasta la posición n con un nḿero k de dígitos. Así que el pseudocódigo queda de la siguiente manera:

```
r := 1;
A[i] := 0;
FOR k := 1 TO n DO
  FOR i := 1 TO k - 1 DO
    r := r * 10 + A[i];
    A[i] := r DIV 2;
    r := r - 2 * A[i];
```

Los enteros no negativos n y k, con $n > 1$. Imprimir k digitos en representación decimal de los numeros $\frac{1}{n}$. Si existen dos representaciones decimales, como 0.499 y 0.500, se imprimirá la última. El programa deberá usar variables enteras solamente.

Mover el punto decimal del número $\frac{1}{n}$, k posiciones a la derecha para dar el número $\frac{10^k}{n}$. Para imprimir la parte entera se debe calcular $10^k \ DIV \ n$. No se debe calcular 10^k para evitar un sobreflujo, mejor se hace una división ordinaria.

```
m := 0;
r := 1;
{m digitos de 1/n se imprimen; el resto de digitos k - m a
 imprimir de la expansión decimal de r/n}

WHILE m <> k DO BEGIN
 ESCRIBIR((10 * r) DIV n);
```

```
 r := (10 * r) MOD n;
 m := m + 1;
END
```

Un número natural $n > 1$. Encontrar la longitud del punto decimal del número $\frac{1}{n}$.

El punto de la fracción decimal es igual al punto de la secuencia del resto r. Considerar el hecho de que el punto de la fracción no puede ser menor que el punto de la secuencia de restos. En la secuencia de restos todos los términos que forman el punto son distintos y la longitud del segmento inicial no periódico no excede n. Por ello, es suficiente para encontrar el $n+1$ término de la secuencia, y entonces para encontrar el mínimo k tal que $n+1+k$ término es igual al $n+1$ término.

```
m := 0;
r := 1;
{r/n = what remains from 1/n after the decimal point
 is moved m positions to the right and the integral
 part is discarded}
WHILE m <> n + 1 do begin
r := (10 * r) mod n;
m := m + 1;
end;
c := r;
{c = (n + 1)-th termino de la secuencia del resto}
r := (10 * r) mod n;
k := 1;
{r = (n + k + 1)-th termino de la misma secuencia}
WHILE r <> c do begin
 r := (10 * r) mod n;
 k := k + 1;
END
```

1.6.4. Apuntadores

Las estructuras fundamentales de datos como array, registro y conjunto, se usan para formar estructuras de datos más complejas. El hecho de definir los datos es para dejarlos fijos con la finalidad de asignarle espacio en la memoria, a estos tipos de datos se les llama datos estáticos. Pero hay otros problemas que requieren que las estructuras de datos cambien durante el cálculo de la solución, a estos tipos de datos se les llama datos dinámicos. Se puede hacer una analogía entre la definición de las estructuras de datos y la definición de la estructura de los algoritmos.

La instrucción elemental no estructurada es la asignación. La estructura de dato elemental o no estructurada es la de tipo escalar. Usando estos dos elementos se construyen instrucciones y tipos de datos respectivamente. Las estructuras más simples obtenidas por medio de la enumeración o sucesión (No a la creación de datos por datos enumerados), son la instrucción compuesta y la estructura de dato registro. Así como la estructura más simples como la estructura formada por la enumeración de datos simples, se componen de un número finito de elementos los cuales pueden ser todos diferentes. La declaración de los elementos involucrados se hace individualmente, esta declaración funciona para todos los datos del mismo tipo.

La instrucción **for** y la estructura **array** se usa para indicar la multiplicación por un factor finito conocido.

Una elección entre dos o más variables se hace por medio de la instrucción **condicional** o la instrucción **case** y con la estructura registro con variante.

La repetición por un factor inicialmente desconocido y potencialmente infinito, se expresa por medio de las instrucciones **while** o **repeat**. La estructura de datos de datos correspondiente es la secuencia (archivo), que es la estructura más simple que permite la construcción de tipos de cardinalidad infinita.

La instrucción procedimiento tiene la propiedad de **recursión**. Como los valores de un tipo de datos recursivo contendrían uno o más componentes pertenecientes a su mismo tipo, de una forma similar a como un procedimiento contiene una o más llamadas a si mismo. El ejemplo es la expresión

aritmética de los lenguajes de programación. La recursión se utiliza para jerarquizar expresiones, donde se usen subexpresiones entre paréntisis como operandos de expresiones. Una expresión se define informamlmente como:

Una expresión está formada por un término seguido de un operador seguido de un término. Los dos términos constituyen los operandos del operador. Un término es, bien una variable (representado por un identificador) o una expresión que está entre paréntisis.

Se definen de la siguiente manera:

```
TYPE expresion = REGISTRO op:operador;
                    op1, op2:termino
                END

TYPE termino = REGISTRO
                   IF t THEN (id:alfa)
                   ELSE (subex:expresion)
                END
```

Lo cual se aplica a las siguientes expresiones:

1. $x + y$
2. $x - (y * z)$
3. $(x + y) * (z - w)$
4. $(x/(y + z)) * w$

Donde $+$ es el operador identificado por *operador*, x y y son terminos identificados por T.

Una forma de empezar a declarar las estructuras de datos enlazadas es con el uso de apuntadores, así que para declararlos se hace de la siguiente manera:

```
Item *item_ptr;
Elem *elem_ap;
```

Esta declaración quiere decir que un objeto elem_ap esta apuntando a un objeto del tipo Elem. Las variantes de uso del apuntador es si se usa de la siguiente manera:

```
star *
```

Lo que señala es que es un apuntador o que se accesa a un objeto referenciado por un apuntador. El asterisco puede aparecer a la derecha o a la izquierda de un objeto.

Por lo que en el siguiente ejemplo:

```
*p
```

donde *p denota el objeto al cual apunta p.

Algorithm 1.2: Declaraciones

```
1 TIPE i, j, k:ENTERO;
2 i := 6;
3 j := 8;
4 k := i + j;
5 imprimir k;
```

En el lenguaje C, es como sigue:

```
#include <stdio.h>
main()
{
int i,j,k; // Definir las variables Sentencia A
i = 6;    // Sentencia B
j = 8;
k = i + j;
printf("suma de dos numeros es %d \n",k); // Impresión de resultados
}
salida : suma de dos numeros es 14
```

El hecho de dar propiedades a las variables permite que se pueda manejar

sus valores de la siguiente manera:

Algorithm 1.3: Manejo de datos

```
1  TIPE i, j, big:ENTERO ;
   Data: ↑ p_a , ↑ p_1 .
   Result: ↑ p_1
2  i, j ← {0, 1, ... n.}
3  i := 6;
4  j := 8;
5  k := i + j;
6  if i < j then
7  |  big := j;
8  else
9  |  return k;
10 return k;
11 if i < j then
12 |  big := j;
13 else
14 big := i;
15 return k;
```

Que traducido al lenguaje C, es como sigue:

```
#include <stdio.h>
main()
{
int i,j,big;  //declaracion de variables
scanf("%d%d",&i,&j); big = i;
if(big < j)  // sentencia A
{       // C
big = j;         // Parte Z, luego parte
}       // D
printf("el más grande de los dos números es %d \n",big);
if(i < j)  // sentencia B
    {
    big = j;  // Parte X
    }
  else
    {
```

```
    big = i;  // Parte Y
    }
printf("el más grande de los dos números (usando else) es %d \n",big);
}
```

Los tipos de datos se pueden construir de acuerdo a las necesidades y van a depender del tipo de lenguaje.

Algorithm 1.4: Creación de nuevos datos

```
1 TIPE i, j, big:ENTERO ;
  Data: ↑ p_a , ↑ p_1 .
  Result: ↑ p_1
2 i, j ← {0, 1, ... n.}
3 i := 1;
4 while i > 0 do
5 |   j := i;
6 |   i := i + 1;
7 return k;
8 return k;
```

```
 // el programa da el valor máximo y mínimo de el tipo de dato
#include <stdio.h>
main()
{
int i,j ;// A
i = 1;
while (i > 0)
{
j = i;
i++;
}
printf ("el valor máximo del entero es %d\n",j);
printf ("el valor del entero después del sobreflujo es %d\n",i);
}
```

Para representar a los apuntadores en pseudocódigo, se aplica lo siguiente:

```
 TYPE Tp = apT;
```

Si tenemos una variable p tipo apuntador, declarada de la siguiente forma:

```
TYPE Tp = apT;
p:Tp;
```

↑

Donde ap es la forma de expresar el apunatador, en C y C++ es el asterisco (*), en pseudocódigo es la ↑.

Así que la instrucción:

```
new(p);
```

Asigna memoria a una variable del tipo T. O se puede asignar de la siguiente forma:

```
TYPE p = apT;
```

Esta representa la dirección del dato T y la siguiente asignación:

```
p ap:T;
```

Es necesario que exista un componente variante en todo tipo recursivo, para asegurar la cardinalidad finita:

```
TYPE T = REGISTRO IF p THEN S(T) END;
```

Un ejemplo del uso de apuntadores:

```
TYPE expresion = REGISTRO op:operador;
                    opd1, opd2: ap termino
                 END;
TYPE termino = REGISTRO
                  IF t THEN (id:alfa)
                  ELSE (sub: ap expresion)
               END
TYPE persona = REGISTRO nombre:alfa;
                        padre, madre: ap persona
               END
```

Una consecuencia adicional del uso explicito de apuntadores es la posibilidad de definir y manipular estructuras cíclicas de datos, la manipulación de estructuras cíclicas de datos puede fácilmente conducir a procesos que no acaben nunca. La estructura recursiva pura de datos podria al nivel del procedimiento, y los apuntadores son comparables con la instrucción **goto**. El siguiente pseudocódigo es un caso especial de recursión y una llamada a una llamada a un procedimiento recursivo P:

```
PROCEDURE P;
BEGIN
   IF B THEN BEGIN P0; P END
END
```

Donde P:

```
WHILE B DO P0
```

Existen una relación similar entre tipos recursivos de datos y la secuencia:

```
TYPE T = REGISTRO
           IF B THEN (t0:T0; t:T)
         END
```

Se puede reemplazar por el tipo secuencial de datos:

```
FILE OF T0;
```

La recursión se puede reemplazar por la iteración.
Los datos que se necesitan en la vida real, como los siguientes ejemplos:

1. Los datos que se necesitan para una fiesta.
2. Los datos de los colores de la ropa para hacer un viaje.
3. Los datos ocultos de un proceso físico.

Se pueden crear por medio de los datos que se verán en el curso, como los siguientes:

1. Listas enlazadas

2. Colas
3. Pilas
4. Árboles
5. Grafos

Capítulo 2

Tipos abstractos de datos

Los siguientes temas son los que se desarrollarán en este capítulo:

1. Abstracción en lenguajes de programación.
2. Tipos Abstractos de Datos.
3. Especificación de los TAD.
   a) Especificación Informal.
   b) Especificación Formal.

Si consideramos que los datos son cualquier información con las que la computadora debe operar o manipular y que ya hemos visto que la abstracción, cuando se aplica a la solución de problemas, le permite concentrarse en la solución del problema, sin preocuparse de los detalles de su implementación. Lo mismo es cierto con los datos. La abstracción de los datos le permite trabajar con los datos sin importar cómo son representados y tratados en la memoria de la computadora. Tomemos la operación de suma como ejemplo. Para la suma de enteros, se deben proporcionar dos argumentos de tipo entero que serán sumados y regresará la suma entera de los dos. ¿Deberá cuidar los detalles de cómo la computadora implementa la suma o representa los enteros en memoria? ¡De seguro que no! Lo que tiene que hacer es proporcionar la información para que la operación de suma realice su trabajo y revisar la información que regresa. En otras palabras, todo lo que tiene que saber es lo que debe suministrar a la operación y cómo le responderá. A la manera en que los tipos abstractos de datos (TAD) actuarán y reaccionarán para una operación determinada se conoce con el nombre de desempeño.

2.1. Abstracción en lenguajes de programación

Para poder realizar un programa que va a resolver un problema de la vida real no es necesario usar todas las variantes del problema y su contexto, sino que es necesario abstraer lo que se usa en el programa, lo que se declara y se opera son los datos, por lo que es importante hacer uso de la abstracción de datos. Para hacer la abstracción de datos se hace de manera lógica con el uso del razonamiento o aplicando herramientas.

Las herramientas ideaware se usan para abstraer, aplicar el encapsulamiento que es la información oculta, refinar los pasos, y aplicar las herramientas visuales.

La forma en el que cada uno de los componentes están relacionados unos con otros y la declaración de las operaciones que pueden ser realizadas sobre los elementos del tipo de dato abstracto.

1. Tipos básicos de datos

2. El código Ascii

2.2. Tipo abstracto de datos

Abstracción de datos es la separación de las propiedades lógicas de los tipos de datos de su implementación. El punto de vista abstracto indica como se construyen las variables de los tipos y como se accesan a componentes individuales de los programas [1].

Concepto de un TDA. El tipo de dato abstracto es un tipo de dato definido por el usuario y se compone de dos partes una parte pública que especifica como manipular los objetos del tipo y una parte privada que es usada internamente por el tipo para implementar el comportamiento especificado por la interfase pública. El TDA es un tipo de dato cuyas propiedades son especificadas independientemente de cualquier implementación particular.

La clase es similar a un TAD. La clase es la base de C++. Una clase se nombra usando la palabra clave class. Las clases no son parte del lenguaje C. Una clase es una colección de variables y funciones que manipulan a las variables. Las variables y funciones que forman a la clase son llamadas miembros. La forma general de la clase es:

```
 class class-name : inheritance-list {
 // miembro privado por default
protected:
 // miembro privado que puede ser inherited
public:
 // miembro público
} object-list;
```

La class-name es el nombre del tipo de clase. Una vez que la clase ha sido compilada, la class-name es un nuevo tipo de nombre que puede ser usada para declarar objetos de la clase. El object-list es una lista separadas por comas de objetos del tipo class-name. Esta lista es opcional. Los objetos clase pueden ser declarados después en el programa solamente usando la class-name. La inheritance-lst es también opcional. Cuando esta presente este especifica la base clase o clases de la nueva clase inherente.

La clase puede incluir la función constructor y la función destructor. Ambos son opcionales. Un constructor es llamado cuando un objeto es primero creado. El destructor es llamado cuando un objeto es destruido. Un constructor tiene el mismo nombre que la clase. Un destructor tiene el mismo nombre que la clase pero lo precede un ~ Ni el constructor ni el destructor regresan tipos. En una clase jerárquica, el constructor es ejecutado en orden de derivación y el destructor en orden inverso.

Por default todos los elementos de la clase son privados y puedan ser accesados por otros elementos de la clase. Para permitir que un elemento de la clase sea accesado por un función que no sea miembro de la clase, se le debe anteponer la palabra clave public. Por ejemplo:

```
 class myclass {
  int a, b; // privado de myclass
public:
  // class members accessible by nonmembers
  void setab(int i, int j) { a = i; b = j; }
  void showab() { cout << a << ' ' << b << endl; }
} ;

myclass ob1, ob2;
```

Esta declaración crea un nuevo tipo de clase myclass que contiene dos variable privadas a y b, también contiene dos funciones públicas setab( ) y

showab( ). El fragmento también declara dos objetos del tipo myclass ob1 y ob2. Para permitir que dos miembros de la clase sean inherentes pero que de otra forma sean privados, especificara a estos como protected. Un miembro protegido esta disponible a ser una clase derivada, pero no esta disponible si su salida es una clase jerárquica.

Cuando operamos un objeto de una clase se usa el punto, operador para referenciar miembros individuales. El operador flecha se usa cuando accesamos a un objeto a través de un apuntador. Por ejemplo el siguiente acceso a la función putinfo( ) de ob que usa el operador punto y la función show( ) usa el operador flecha.

```
 struct cl_type {
  int x;
  float f;
public:
  void putinfo(int a, float t) { x = a; f = t; }
  void show() { cout << a << ' ' << f << endl; }
} ;

cl_type ob, *p;

// ...

ob.putinfo(10, 0.23);

p = &ob; // poner dirección de ob en p

p->show(); // desplegar los datos de ob
```

es posible crear una clase genérica usando la palabra clave template.

En C++ es posible heredar las características de otro. La clase que hereda es llamada la clase base. La clase heredada es la clase derivada. Cuando una clase hereda a otra se forma una clase jerárquica. La forma general para heredar a una clase es:

```
 class class-name : access base-class-name {
  // . . .
} ;
```

Aquí el acceso lo determina como la clase es heredada. Para heredar a mas de una clase usar una lista separadas por comas.

La siguiente clase jerárquica la herencia derivada es la base como privada. Esto significa que i llega a ser un miembro privado de derivado.

```
 class base {
public:
  int i;
};

class derived : private base {
  int j;
public:
  derived(int a) { j = i = a; }
  int getj() { return j; }
  int geti() { return i; } // OK, derivado tiene acceso a i
};

derived ob(9); // crea un objeto derivado

cout << ob.geti() << " " << ob.getj(); // OK

// ob.i = 10; // ERROR, i es privado y derived!
```

2.3. Especificación del TAD

En muchos casos existen porciones de código similares que no calculan un valor si no que por ejemplo, presentan información al usuario, leen una colección de datos o calculan más de un valor. A esto se llama procedimiento. Los procedimientos son herramientas esenciales de programación que generalizan el concepto de operador. Por medio de los procedimientos se pueden definir nuestros propios operadores que se aplican a operandos que no tienen que ser de tipo básico.

El TAD es un modelo matemático con una serie de operaciones definidas en ese modelo. Éste se especifica de dos formas: de manera formal y de manera informal.

2.3.1. Especificación formal del TAD

La especificación formal del TAD se hace como si fuera un modelo matemático. Por medio de proposiciones de tal manera que se definen los tipos de datos y operaciones que se realizan a los operandos.

Un enfoque riguroso de la especificación de la última parte supone suministrar un conjunto de axiomas que describan completamente el comportamiento de las operaciones del TAD. Estos axiomas pueden entonces ser usados para verificar formalmente la corrección del desarrollo de un TAD. Desgraciadamente, determinar un conjunto completo de axiomas es extremadamente difícil para los TAD no triviales. Por esta razón nos preocuparemos nada más de obtener descripciones informales para las operaciones del TAD.

Las dos partes que componen al TAD donde se declaran los tipos de datos que se hace por medio de aserciones para declarar una precondición y una postcondición, que es la parte pública del TAD y la declaración de las operaciones que es la parte privada donde se usan los procedimientos.

Una aserción es una proposición lógica que puede ser falsa o verdadera. Precondición es una aserción que debe ser verdadera a la entrada dentro de una operación o una función para que la postcondición sea garantizada. Postcondición es una aserción que establece los resultados esperados a la salida de una operación o función, asumiendo que las precondiciones son verdaderas.

Ejemplo 1: Una función que remueve el último elemento de la lista y que regresa su valor en UltimoValor. GetLast(ListType list, ValueType lastValue). Así que la precondición es que la lista no esta vacía. La post-condición UltimoValor es el valor del último elemento en la lista, el último elemento ha sido removido, y la longitud de la lista ha sido decrementada.

Ejemplo 2: Una función divide un número entre otro y prueba un divisor con valor de cero. Divide(int dividend, int divisor, bool error, float result)

Precondición: Ninguna. Post-condición: el error es verdadero si el divisor es cero. El resultado es el dividendo / divisor , si el error es falso. El resultado es indefinido, si el error es verdadero.

2.3.2. Especificación informal del TAD

Debido a que es muy complicado especificar formalmente un TAD por eso nos preocuparemos nada más de obtener descripciones informales para las operaciones del TAD.

La especificación informal del TAD se hace por medio de expresiones del lenguaje natural, como una especie de pseudocódigo.

2.3.3. Estructuras de datos

Estructura de datos es una colección de elementos de datos cuya organización es caracterizada por accesar a operaciones que son usadas para almacenar y recuperar los elementos dato individuales; la implementación de los miembros datos compuestos en un tipo de datos abstracto.

Clase es un tipo no estructurado que encapsula un número fijo de componentes de datos con la función que los manipula: las operaciones predefinidas sobre una instancia de una clase son cuyos asignamientos y componentes acceso.

Clase objeto es la descripción de un grupo de objetos con propiedades y comportamiento físico similares; un patrón para crear objetos individuales.

Encapsulamiento de datos es la separación de la representación de datos de las aplicaciones que usan a estos datos en un nivel lógico; una característica del lenguaje de programación que ejecuta información oculta.

Presentar ejemplos de:

1. tipos de datos
2. de abstracción de datos
3. de TDA
4. de estructuras de datos.

TIPE ↑ sig:EstListDoble

Capítulo 3

Estructuras de Datos Lineales, Estáticas y Dinámicas.

Las estructuras de datos son una colección de datos que pueden ser caracterizados por su organización y las operaciones que se definen de ellas. Lo que se pretende con las estructuras de datos es facilitar un esquema lógico para manipular los datos en función del problema que haya que tratar y el algoritmo para resolverlo. En algunos casos la dificultad para resolver un problema radica en escoger la estructura de datos adecuada. En general, la elección del algoritmo y de las estructuras de datos que manipulará estará muy relacionada.

Según su comportamiento durante la ejecución del programa las estructuras de datos se pueden clasificar: De acuerdo al orden de almacenamiento de las localidades de memoria se clasifican en lineales y no lineales. Las lineales son aquellas estructuras donde los datos se almacenan en zonas contiguas (sucesivas o adyacentes), una detrás de otras. Ejemplo: Listas enlazadas, Pilas, Colas. Y por su gestión de memoria se consideran en estáticas y dinámicas. Estáticas: su tamaño en memoria es fijo. Ejemplo: arrays, conjuntos, cadenas. Dinámicas: su tamaño en memoria es variable. Ejemplo: listas enlazadas con apuntadores, Pilas, colas, árboles, grafos, etc.

3.1. Pilas

Una pila es un tipo especial de lista abierta en la que sólo se pueden insertar y eliminar nodos en uno de los extremos de la lista. Teniendo en cuenta

que las inserciones y borrados en una pila se hacen siempre en un extremo, lo que consideramos como el primer elemento de la lista es en realidad el último elemento de la pila. Las operaciones básicas de una pila son:

1. Meter o Push: Añade un elemento al final de la pila.

2. Sacar o Pop: Lee y elimina un elemento del final de la pila.

La construcción de la pila se hace de dos formas una estática con arreglos y otra dinámica con listas enlazadas y con apuntadores.

3.1.1. Especificación del TAD Pila

La pila es una estructura lineal, donde el primero que entra es el último que sale. Así que para definir los datos y las operaciones que forman al TAD se hace por medio de las cinco operciones de la pila, las cuales son:

1. ANULA(p) convierte la pila p en una pila vacía. Esta operación es exactamente la misma que para las listas generales.

2. TOPE(p) devuelve el valor del elemento de la parte superior de la pila p. Si se identifica la parte superior de una pila con la posición 1, como suele hacerse, entonces TOPE(p) puede escribirse en función de operaciones con listas como RECUPERA(PRIMERO(p), p).

3. SACA(p), en inglés POP, suprime el elemento superior de la pila, es decir, equivale a SUPRIME(PRIMERO(p), p). Algunas veces resulta conveniente implantar SACA como una función que devuelve el elemento que acaba de suprimir, aunque aquí no se hará eso.

4. METE(x, p) en inglés PUSH, inserta el elemento x en la parte superior de la pila p. El anterior tope se convierte en el siguiente elemento, y así sucesivamente. En función de operaciones primitivas con listas, esta operación es INSERTA(x, PRIMERO(p), p).

5. VACIA(p) devuelve verdadero si la pila p esta vacía, y falso en caso contrario.

Ahora sólo existe un caso posible, ya que sólo podemos leer desde un extremo de la pila. Partiremos de una pila con uno o más nodos, y usaremos un apuntador auxiliar llamado, ↑ nodo:

3.1.2. Representación estática

La representación estática generalmente se hace por medio de arreglos. Donde el primer elemento del arreglo es el fondo de la pila y el último elemento del arreglo es la cima. El número de elementos del arreglo es el número de elementos de la pila. Así que la construcción de la pila por medio de arreglos, se hace de la siguiente manera:

Algorithm 3.1: Construcción de una pila con arreglos

```
1  TIPE Pila = REGISTRO
2  {
3  TIPE cima:TIPO_DATO;
4  TIPE elemento:ARREGLO[1 ... n] of ENTERO;
5  }
6  TIPE Pila p;
7  TIPE n:ENTERO;
8  n := Escribir_Por_Teclado(ENTERO);
9  cima := 1;
10 while n <> NULL do
11 |  p.elemento[cima] := Escribir_Por_Teclado(ENTERO);
12 |  cima := cima + 1;
13 |  n := n - 1;
```

3.1.3. Representación dinámica

Esta representación se hace por medio de listas enlazadas. Para construir una pila se comienza utilizando push aplicandolo a una lista vacía. Partiremos de que ya tenemos el nodo a insertar y por supuesto un apuntador que apunte a él, además el apuntador a la pila valdrá NULL. Podemos considerar el caso anterior como un caso particular de éste, la única diferencia es que podemos y debemos trabajar con una pila vacía como con una pila normal. La siguiente opción se aplicará el Push en una pila no vacía: Por lo que partiremos de un nodo a insertar, con un puntero que apunte a él, y de una pila, en este caso no vacía.

Si la pila cuenta con elementos se puede aplicar la función Pop, leer elementos y eliminar un elemento.

La construcción de la pila de manera dinámica se hace de la siguiente

manera:

Algorithm 3.2: Construcción de una pila de manera dinámica

```
1  TIPE EstPila = REGISTRO
2  {
3  TIPE dato:TIPO_DATO;
4  TIPE ↑sig:EstPila;
5  }
6  TIPE ↑cima, ↑NuevoElemento, ↑m:EstPila;
7  TIPE n:ENTERO;
8  n := Escribir_Por_Teclado(ENTERO);
9  ↑cima := NULL;
10 while n <> 0 do
11     NuevoElemento := NEW EstPila;
12     NuevoElemento.dato := Escribir_Por_Teclado(TIPO_DATO);
13     if ↑cima == NULL then
14         ↑cima := NuevoElemento;
15         NuevoElemento.sig := NULL;
16         ↑m := ↑cima;
17     else
18         ↑cima := NuevoElemento;
19         NuevoElemento.sig := ↑m;
20         ↑m := ↑cima;
21     n := n - 1;
```

3.1.4. Ejercicio de pilas

Se debe aplicar la construcción de listas para tener la sucesión de números en una lista enlazada. Se debe meter el número más grande en una pila, así que 10 va en la pila después se mete 7 a la pila, después 2 y como 3 es mayor que la cima se crea una nueva pila y se mete 3, como 4 es mayor se crea una nueva pila y se mete 4, como 11 es mayor se crea una nueva pila y se mete 11 se mete 1, como 8 es mayor se crea una nueva pila y se mete 8 se mete 6, como 9 es mayor se crea una nueva pila y se mete 9 también se mete 5 y como se llega al NULL se termina el proceso. Así que las pilas que quedan son:

$$< 10, 7, 2 >$$
$$< 3 >$$
$$< 4 >$$
$$< 11, 1 >$$
$$< 8, 6 >$$
$$< 9, 5 >$$

Después se procede con las 6 pilas que tenemos y que estan numeradas como $p_1, p_2, p_3, p_4, p_5, p_6$, así que tenemos que comparar las 6 cimas, se puede meter las 6 cimas a una lista enlazada y hacer la comparación. Lo que quedaría de la siguiente forma:

$$< 2, 3, 4, 1, 6, 5 >$$

A esta lista se aplica el ordenamiento por dos apuntadores. Los números que faltan se meten a otra lista que queda de la siguiente forma:

$$< 7, 11, 8, 9 >$$

Y en la pila queda el número 10.

O se puede hacer la comparación de la pila p_1 con la pila p_2:

$$< 10, 7, 3 >$$
$$< 3 >$$

Se crea una pila auxiliar p_a deonde se mete el número 2:

$$< 10, 7 >$$
$$< 3 >$$
$$< 2 >$$

Después se mete el 3 en la pila p_1 y posteriormente el 2, así que la pila p_1 queda como sigue:

$$< 10, 7, 3, 2 >$$

Esta sigue siendo la pila p_1. Ahora el proceso se repite la pila p_1 con la pila p_3, se crea una pila auxiliar p_a y el conjunto de pilas queda de la siguiente forma:

$$< 10, 7, 3 >$$

$$< 4 >$$

$$< 2 >$$

Ahora se mete la cima mas grande en la pila p_1 y después la siguiente cima más pequeña, así que la pila p_1 queda como sigue:

$$< 10, 7, 4, 3, 2 >$$

Se compara la pila p_1 con la pila p_4 o sea:

$$< 10, 7, 4, 3, 2 >$$

$$< 11, 1 >$$

Así que se agrega 1 a la pila p_1 y queda:

$$< 10, 7, 4, 3, 2, 1 >$$

$$< 11 >$$

Se procede a crear la pila auxiliar lo que quedaría de la siguiente forma:

$$<>$$

$$< 11 >$$

$$< 1, 2, 3, 4, 7, 10 >$$

Se comienza a vaciar la pila auxiliar p_a en la pila p_1 y después la pila p_4 en la pila p_1, lo que queda de la siguiente manera:

$$< 11, 10, 7, 4, 3, 2, 1 >$$

Después se hace la comparación de la pila p_1 con la pila p_5, de la siguiente manera:

$$< 11, 10, 7, 4, 3, 2, 1 >$$

$$< 8, 6 >$$

Se crea una pila auxiliar p_a y se mete los siguiente números:

$$< 11, 10, 7 >$$
$$< 8, 6 >$$
$$< 1, 2, 3, 4 >$$

Al comparar las cimas en este estado propicia la siguiente acción:

$$< 11, 10, 7, 6 >$$
$$< 8 >$$
$$< 1, 2, 3, 4 >$$

Las siguientes comparaciones de las cimas de las pila p_1 y de p_5, propicia lo siguiente:

$$< 11, 10, 8 >$$
$$<>$$
$$< 1, 2, 3, 4, 6, 7 >$$

Después se vacia la pila auxiliar p_a en la pila p_1, por lo que queda de la siguiente manera:

$$< 11, 10, 8, 7, 6, 4, 3, 2, 1 >$$

Por último se compara la pila p_1 con la pila p_6 de la siguiente forma:

$$< 11, 10, 8, 7, 6, 4, 3, 2, 1 >$$
$$< 9, 5 >$$

Como la cima de p_1 es menor se crea una pila auxiliar p_a y se llena de la siguiente forma:

$$< 11, 10, 8, 7, 6 >$$
$$< 9, 5 >$$
$$< 1, 2, 3, 4 >$$

Así que el algoritmo en pseudocódigo de la comparación de las dos pilas es el siguiente 3.3:

Algorithm 3.3: Dos pilas

Data: $\uparrow p_a$, $\uparrow p_b$.
Result: $\uparrow p_1$

```
1  ↑p1 := NULL , ↑L1 := < 1,2,3,4 > ;
2  TIPE Dr:↑ Dr ;
3  TIPE Iz:↑ Iz ;
4  TIPE estructuraArbol = REGISTRO
5  {
6  TIPE dato:TIPO_DATO ;
7  TIPE Dr:estructuraArbol ;
8  TIPE Iz:estructuraArbol ;
9  }
10 pa ← {2...n.};
11 repeat
12 |  while ↑pa <> NULL do
13 |  |  if cima(↑p1) > cima(↑p2) then   /* Mientras se vacia */
14 |  |  |  PUSH(cima(↑p2), ↑p1) ;
15 |  |  else
16 |  |  end
17 |  |  if Pila ↑pa No existe then      /* Si la pila no existe */
18 |  |  |  CrearPila(↑pa) ;
19 |  |  |  PUSH(cima(↑p1), ↑pa) ;
20 |  |  else
21 |  |  end
22 |  |  PUSH(cima(↑p1), ↑pa) ;
23 |  end
24 |  while ↑pa <> NULL do                    /* Comentario */
25 |  |  PUSH(Cima(↑pa), ↑p1) ;
26 |  end
27 until i := 0 to Numero de pilas creadas - 1;
28 return pila ;
```

3.2. Colas

Es una estructura que tiene la filosofía de que el primero que entra es el primero que sale. Se representa por una lista que tiene como identificador el frente y la parte posterior. Estos elementos se identifican por medio de dos apuntadores.

3.2.1. Especificación del TAD Cola

La cola es un tipo especial de lista en el cual los elementos se insertan en un extremo llamado posterior y se suprime en el otro extremo llamado anterior o frente. Las listas también se llaman FIFO.

Esta tiene las siguientes operaciones:

1. ANULA() convierte la cola C en una lista vacía.

2. FRENTE() es una función que devuelve el valor del primer elemento de la cola C.

3. PON_EN_COLA(x,C) inserta el elemento x al final de la cola C.

4. QUITA_DE_COLA(C) suprime el primer elemento de la cola C.

5. VACIA(C) devuelve verdadero si, y sólo si, C es una cola vaía.

3.2.2. Representación estática

La cola se puede construir por medio de arreglos, donde el frente es el último elemento del arreglo y el primer elemento de la cola o frente es el primer elemento del arreglo. Por lo que se construye de la siguiente manera,

donde se muestra el pseudocódigo.

Algorithm 3.4: Construcción de una estructura cola por medio de arreglos

```
1  TIPE Cola = REGISTRO
2  {
3  TIPE Frente:ENTERO;
4  TIPE elemento:ARREGLO[1 ... n] of ENTERO;
5  }
6  TIPE Cola c;
7  TIPE n:ENTERO;
8  n := Escribir_Por_Teclado(ENTERO);
9  Cola := 1;
10 while n <> NULL do
11 |  c.elemento[Cola] := Escribir_Por_Teclado(ENTERO);
12 |  Cola := Cola + 1;
13 |  n := n - 1;
```

3.2.3. Representación dinámica

Para la creación de la estructura cola se hace la inserción de los elementos después del último elemento creado, de tal manera que se siga la filosofía de

la estructura cola, de la siguiente manera:

Algorithm 3.5: Construcción de una estructura cola de manera dinámica

```
1  TIPE EstCola = REGISTRO
2  {
3  TIPE dato:TIPO_DATO;
4  TIPE ↑sig:EstCola;
5  }
6  TIPE ↑c, ↑NuevoElemento, ↑Frente:EstCola;
7  TIPE n:ENTERO;
8  n := Escribir_Por_Teclado(ENTERO);
9  while n <> 0 do
10 |  NuevoElemento := NEW EstCola;
11 |  NuevoElemento.dato := Escribir_Por_Teclado(TIPO_DATO);
12 |  if ↑Frente == NULL then
13 |  |  ↑Frente := NuevoElemento;
14 |  |  NuevoElemento.sig := NULL;
15 |  |  ↑c := ↑Frente;
16 |  else
17 |  |  ↑cima := NuevoElemento;
18 |  |  NuevoElemento.sig := ↑m;
19 |  |  ↑c := ↑Frente;
20 |  n := n - 1;
```

3.2.4. Colas circulares

Las colas lineales tienen un grave problema como las extracciones, que sólo pueden realizarse por un extremo, puede llegar un momento en que el apuntador ↑A sea igual al máximo número de elementos en la cola, siendo que al frente de la misma existan lugares vacíos, y al insertar un nuevo elemento nos mandará un error de overflow (cola llena). Para solucionar el problema de desperdicio de memoria se implementaron las colas circulares, en las cuales existe un apuntador desde el último elemento al primero de la cola.

La representación gráfica de esta estructura es la siguiente:

3.3. Colas de prioridades

Cualquier estructura de datos que soporta las operaciones de búsqueda de un mínimo o un máximo, insertar, y borrar un mínimo o un máximo, es llamada una cola de prioridad. La forma más simple de representar una cola de prioridad es por medio de una lista lineal no ordenada. Se tiene una cola de n elementos y la operación max que es soportada se borra. La cola de prioridad se puede implementar por medio de heap max, que se refiere a un árbol binario completo donde cada nodo es más grande que el valor de sus hijos. Esto quiere decir que el nodo raíz es el más grande.

Una cola de prioridad es una estructura de datos que permite al menos las siguientes dos operaciones: insertar, que añade elementos a la cola, y eliminar mínimo, que busca, devuelve y elimina el elemento mínimo de la cola.

La implantación de la cola de prioridad en la biblioteca de clases de DSTool se realiza mediante un montículo binario.

Para que una estructura sea un montículo binario debe cumplir dos propiedades:

1. Un montículo es un árbol binario completamente lleno, con la posible excepción del nivel más bajo, el cual se rellena de izquierda a derecha. Estos árboles se denominan árboles binarios completos.
2. Todo nodo debe ser menor que todos sus descendientes. Por lo tanto, el mínimo estará en la raíz y su búsqueda y eliminación se podrá realizar rápidamente.

Los árboles binarios completos son muy regulares, lo que permite que los montículos puedan ser representados mediante simples arrays sin necesidad de punteros. En una representación con arrays los hijos de un elemento colocado en la posición i estarían en las posiciones 2i y 2i+1 (hijo izquierdo e hijo derecho respectivamente), y el padre en la posición E(i/2) (parte entera de i/2).

En el siguiente ejemplo:

El árbol se compone claramente de los hijos de B (D y E) y su padre (A). Los hijos de C (F) y su padre es (A). A través de la representación con un array se podría obtener la misma información:

Nodo B en la posición 2.

Hijo izquierdo: posición 2*2 = 4. Luego el hijo izquierdo es D.

Hijo derecho: posición 2*2+1 = 5. Luego el hijo derecho es E.

Padre: posición E(2/2) = 1. Luego el padre es A.

3.4. Listas enlazadas

Presentar el TAD lista:

Como se representa un apuntador:

Se presentan figuras de localidades de memoria y una lista simplemente enlazada presentando las localidades de memoria.

Se presenta la representación gráfica de una lista enlazada.

Se presenta la operación INSERTA(x, p, L) por medio de la representación gráfica y se resumen los pasos para implementar esta operación.

Se debe ver las partes que forman una lista enlazada:

Los elementos se forman de la siguiente manera: el nombre de la lista es un apuntador, los elementos que componen a la lista, el final de la lista.

3.4.1. Creación de la lista enlazada

Para crear la lista, primero se declaran los tipos de datos:

Algorithm 3.6: Declaración de datos de una lista

```
1 TIPE ListaEstructura = REGISTRO
2 {
3 TIPE dato:TipoDato ;
4 TIPE ↑ sig:ListaEstructura;
5 }
6 TIPE NL:↑ NL ;
7 TIPE ↑ NL: ListaEstructura ;
```

El nombre de la lista se crea al construir la lista de la siguiente manera:

Algorithm 3.7: Crear la lista

```
1 funcion CrearLista:ListaEstructura ;
2 var ↑ NL:ListaEstructura ;
3 begin
4 ↑ NL := NULL ;
5 return ↑ NL ;
6 end
```

El siguiente paso es crear los elementos de la siguiente forma:

Algorithm 3.8: Crear los elementos

```
1 funcion CrearElemento:ListaEstructura ;
2 var ↑ elem:ListaEstructura ;
3 begin
4 elem := NEW ListaEstructura ;
5 return elem ;
6 end
```

3.4.2. Otra forma de construir una lista enlazada

La lista simplemente enlazada se construye de la siguiente manera:

```
1  TIPE sig:↑ sig ;
2  TIPE estructuraLista = REGISTRO
3  {
4  TIPE dato:ENTERO ;
5  TIPE sig:estructuraLista ;
6  }
7  TIPE Lista:↑ Lista ;
8  ↑ Lista := NULL ;
9  TIPE nuevoElemento:↑ nuevoElemento ;
10 nuevoElemento := NEW estructuraLista ;
11 nuevoElemento.sig := ↑ Lista ;
12 ↑ Lista := NuevoElemento ;
13 nuevoElemento.dato := dato1 ;
14 for i = 2, 3, ...,10 do
15 |  nuevoElemento := NEW estructuraLista ;
16 |  nuevoElemento.sig := NULL ;
17 |  ↑ p := nuevoElemento ;
```

Para hacer la búsqueda en una lista ya construida:

```
1 while p ↑.sig <> NULL do
2 |  if p ↑.dato == x then
3 |  |  Encontrado() ;
4 |  else
5 |  |  ↑ p := p ↑.sig ;
```

Se implementa la función LOCALIZA(x,L), la cual devuelve la posición

de x:

```
1 while p↑.sig <> NULL do
2     if p↑.dato == x then
3         Encontrado() ;
4         Return(↑ p) ;
5     else
6         ↑ p := p ↑.sig ;
```

Se implementa la función RECUPERA(p,L), la cual devuelve el elemento que esta en la posición p de la lista:

```
1 while p↑.sig <> NULL do
2     if p↑.sig == p then
3         Encontrado() ;
4         Return(p ↑.dato) ;
5     else
6         ↑ p := p ↑.sig ;
```

Para copiar la lista en otra lista, se procede con el siguiente pseudocódigo:

```
1  TIPE a:↑ a ;
2  TIPE ↑ a:estructuraLista ;
3  TIPE Lista1:↑ Lista1 ;
4  ↑ Lista1 := NULL ;
5  ↑ a := ↑ Lista1 ;
6  ↑ p := ↑ Lista ;
7  while p ↑.sig <> NULL do
8      if ↑ a == NULL then
9          nuevoElemento := NEW estructuraLista ;
10         nuevoElemento.sig := NULL ;
11         ↑ a := nuevoElemento ;
12         nuevoElemento.dato := RECUPERA(p,Lista) ;
13     else
14         nuevoElemento := NEW estructuraLista ;
15         nuevoElemento.sig := NULL ;
16         a ↑.sig := nuevoElemento ;
17         nuevoElemento.dato := RECUPERA(p,Lista) ;
18     ↑ p := p ↑.sig ;
```

3.4.3. Algoritmo de construcción de una lista enlazada

Para la construcción de la lista enlazada se aplica el siguiente algoritmo:

```
1  TIPE sig:↑ sig ;
2  TIPE listaEstructura = REGISTRO { TIPE dato:tipoDato ;
3  TIPE sig:listaEstructura;
4  }
5  TIPE NL:↑ NL ;
6  TIPE ↑ NL:listaEstructura ;
7  TIPE guardarValor():tipoDato ;
8  TIPE nuevoElemento:↑ nuevoElemento ;
9  nuevoElemento := NEW listaEstructura ;
10 ↑ NL := nuevoElemento ;
11 nuevoElemento.dato := guardarValor() ;
12 if nuevoElemento == ultimo then
13 |  nuevoElemento.sig := NULL;
14 else
15 |  nuevoElemento.sig := nuevoElemento;
16 |  nuevoElemento := NEW listaEstructura;
17 |  nuevoElemento.dato := guardarValor();
```

Creación de elementos de la lista enlazada

Para crear los elementos de la lista se usa la función crear elementos de la siguiente manera: CrearElemento(),

Resolver el siguiente ejemplo:
Buscar datos por comparación.

```
1 TIPE Ap:↑ Ap ;
2 ↑ Ap := ↑ NL ;
3 Ap ↑.dato == x ;
```

3.4.4. Operaciones en listas enlazadas

Ejemplo de lista.

Se tiene una lista de direcciones y se desean eliminar las entradas dobles. En forma conceptual, este problema es bastante fácil de resolver; para cada elemento de la lista, se debe eliminar todos los elementos sucesores equivalentes. Para presentar este algoritmo, sin embargo, es necesario definir operaciones que permitan encontrar el primer elemento de una lista, recorrer los demás elementos sucesivos, y recuperar y eliminar elementos.

Matemáticamente, una lista es una secuencia de cero o más elementos de un tipo determinado (que por lo general se denominará tipo_elemento). A menudo se representa una lista como una sucesión de elementos separados por una coma

$$a_1, a_2, \ldots, a_n$$

donde $n \geq 0$ y cada a_i es del tipo_elemento. Al número n de elementos se le llama longitud de la lista. Al suponer que $n \geq 1$, se dice que a_1 es el primer elemento y a_n es el último elemento. Si $n = 0$, se tiene una lista vacía, es decir, que no tiene elementos. Una propiedad importante de una lista es que sus elementos pueden estar ordenados en forma lineal de acuerdo con sus posiciones en la misma. Se dice que a_i precede a a_{i+1} para $i = 1, 2, \ldots, n-1$, y que a_i sucede a a_{i-1} para $i = 2, 3, \ldots, n$. Se dice que el elemento a_i esta en la posición i. Es conveniente postular también la existencia de una posición que sucede a la del último elemento de la lista. Se identificará a la lista como L, la función FIN(L) devolverá la posición que sigue a la posición n en una lista L de n elementos. La posición FIN(L), con respecto al principio de la lista, esta a una distancia que varía conforme la lista crece o se reduce, mientras que las demás posiciones guardan una distancia fija con respecto al principio de la lista [**?**].

Para formar un tipo de datos abstracto a partir de la noción matemática de la lista, se debe definir un conjunto de operaciones con objetos de tipo LISTA o L. Donde L es una lista de objetos de tipo tipo_elemento, x es un objeto de ese tipo y p es de tipo tipo_posición. La posición es otro tipo de datos cuya implantación cambiará con aquellas que se haya elegido para las listas. Aunque de manera informal se piensa en las posiciones como enteros, en la práctica pueden tener otra representación. Las operaciones que se requieren para formar la lista L, son las siguientes:

1. INSERTA(x, p, L). Esta función inserta x en la posición p de la lista L,

pasando los elementos de la posición p y siguientes a la posición inmediata posterior. Esto quiere decir que si L es $a_1, a_2, \ldots, a_n$, se convierte en:

$$a_1, a_2, \ldots, a_{p-1}, x, a_p, \ldots, a_n$$

Si p es FIN(L), entonces L se convierte en $a_1, a_2, \ldots, a_n, x$. Si la lista L no tiene posición p, el resultado es indefinido.

2. LOCALIZA(x, L). Esta función devuelve la posición de x en la lista L. Si x figura más de una vez en L, la posición de la primera aparición de x, es la que se devuelve. Si x no figura en la lista, entonces se devuelve FIN(L).

3. RECUPERA(p, L). Esta función devuelve el elemento que esta en la posición p de la lista L. El resultado no esta definido si p = FIN(L) o si L no tiene posición p. Es necesario que los elementos sean de un tipo que puedan ser devueltos por la función, aunque es posible modificar la función RECUPERA para devolver un apuntador o un objeto de tipo tipo_elemento.

4. SUPRIME(p, L). Esta función elimina el elemento en la posición p de la lista L. Si L es $a_1, a_2, \ldots, a_n$, entonces L se convierte en:

   $$a_1, a_2, \ldots, a_{p-1}, a_{p+1}, \ldots, a_n$$

   El resultado no esta definido si L no tiene posición p o si p = FIN(L).

5. SIGUIENTE(p, L) y ANTERIOR(p, L) devuelven las posiciones siguiente y anterior, respectivamente, a p en la lista L. Si p es la última posición de L, SIGUIENTE(p, L) = FIN(L). SIGUIENTE(p, L) no esta definida si p es FIN(L). ANTERIOR no esta definida si p es 1. Ambas funciones están indefinidas cuando L no tiene posición p.

6. ANULA(L). Esta función ocasiona que L se convierta en una lista vacía y devuelve la posición FIN(L).

7. PRIMERO(L). Esta función devuelve la primera posición de la lista L. Si L esta vacía, la posición que devuelve es FIN(L).

8. IMPRIME_LISTA(L). Imprime los elementos de L en su orden de aparición en la lista.

Usando los operadores anteriores escribir el procedimiento PURGA que toma como argumento una lista y elimina sus elementos con doble entrada. Los elementos son del tipo tipo_elemento, y una lista de ellos es del tipo LISTA L. Se consideran dos posiciones en la lista p y q. Por lo que basados en el concepto de procedimiento,

> Los procedimientos, son herramientas esenciales de programación, que generalizan el concepto de operador. Este se puede usar para definir otros operadores para aplicarlos a operandos que no necesariamente deben ser básicos o primarios.

se crea el procedimiento PURGA, que a continuación se representa:

```
Procedimiento PURGA(L)
 \\PURGA elimina los elementos repetidos de la lista L.
  tipo_posición p, q. \\p será la posición actual en L, y q posición que
                      avanza para encontrar elementos iguales.
  comenzar PURGA
(1)    p = PRIMERO(L);
(2)      mientras p ¡= FIN(L) comenzar a hacer
(3)       q = SIGUIENTE(p, L);
(4)         mientras q ¡= FIN(L) hacer
(5)            si MISMO(RECUPERA(p, L), RECUPERA(q, L)) entonces
(6)                 SUPRIME(q,L)
(N)            o hacer
(7)                 q == SIGUIENTE(q, L)
(8)            p = SIGUIENTE(p, L)
            fin
  fin PURGA
```

La función la definen como:

> Desde un punto de vista práctico, podemos decir que una función es una parte de un programa (subrutina) con un nombre, que puede ser invocada (llamada a ejecución) desde otras partes tantas veces como se desee. Un bloque de código que puede ser ejecutado como una unidad funcional. Opcionalmente puede recibir valores; se ejecuta y puede devolver un valor. Desde el punto de vista de la organización, podemos decir que una función es algo que permite

un cierto orden en una maraña de algoritmos. Como resumen de lo anterior podemos concluir que el uso de funciones se justifica en dos palabras: organización y reutilización del código. Desde este último punto de vista (reutilización), puede decirse que son un primer paso de la programación genérica, ya que representan un algoritmo parametrizado [2].

Algunos conceptos de funciones y su semejanza con los procedimientos:

En C++ las funciones desempeñan ambos papeles, aunque en cierto modo, los ficheros C++ desempeñan algunas funcionalidades de lo que, en otros lenguajes como Modula-2, se denominan módulos [3]. Otra diferencia substancial es que C++ no permite el anidamiento de funciones, es decir, definir funciones dentro de otras. En C++ todas las funciones se definen a nivel de fichero, con lo que tienen ámbito global al fichero.

[1] Las funciones miembro de clases (métodos) deben ser declaradas siempre dentro del cuerpo de la clase, aunque la definición puede estar fuera.

[2] Las funciones han sido definidas como los verbos de los lenguajes de programación (indican alguna acción). Algunos autores mantienen que la función es una unidad lógica de programación, y que su extensión debe limitarse al código que pueda incluirse en una pantalla, con objeto de que pueda tenerse una visión completa de la misma de un solo vistazo y sea más fácil de entender el funcionamiento de cada unidad lógica del programa.

[3] Los módulos de otros lenguajes pueden ser fácilmente mimetizados en C++ definiendo clases en las que todos sus miembros sean estáticos.

Funciones dentro de clases

En la jerga de la programación orientada a objetos, las funciones dentro de las clases se denominan funciones-miembro o métodos,

y las variables dentro de clases, variables-miembro o propiedades. El sentido es el mismo que en la programación tradicional (la nomenclatura es más una cuestión de gustos), si bien referirnos a "propiedades" y "métodos" supone estar utilizando la programación orientada a objetos y que nos referimos a miembros de una clase. En C++ esta aclaración puede ser importante, porque es un lenguaje que podríamos llamar "híbrido"; en ciertas partes puede utilizarse con técnicas de programación tradicional, y en otras con técnicas de POO.

Una función de inicio

Cada programa C++ debe tener una sola función externa denominada main(), que desde la óptica del programador define el punto de entrada al programa. Las funciones se declaran en cabeceras (estándar o específicas de usuario) o dentro de los ficheros fuente. Estas declaraciones son denominadas prototipos. En ocasiones la declaración y definición se realiza en el mismo punto (como ocurre con las variables), aunque es normal colocar al principio del fuente los "prototipos" de las funciones que serán utilizadas en su interior, y las definiciones en cualquier otro sitio (generalmente al final). En el caso del ejemplo anterior, la declaración y definición de func1 se ha realizado en el mismo punto, mientras que la declaración de func2 se realiza dentro del cuerpo de la clase y la definición en el exterior de esta [1].

Para realizar este procedimiento se tendrá que construir la función MISMO, que consiste en encontrar elementos que tienen el mismo valor. La función se representará como MISMO(x, y), donde x y y son elementos tipo_elemento de la lista L. Esta función determinará si los elementos x y y tienen el mismo valor. Si son iguales la función entregará un valor de verdadero y si no son iguales entregará un valor de falso.

```
MISMO(x,y)

 {
```

```
    Si x == y;
    return(1);
}
```

Esta función en pseudocódigo se escribe de la siguiente manera:

Algorithm 3.9: Funcion MISMO

```
1 funcion MISMO:BOOLEANA ;
2 var x, y:REALES ;
3 begin
4 if x == y then
5 return 1 ;
6 end
```

La implementación de un TAD es la traducción en proposiciones de un lenguaje de programación de alto nivel, de la declaración que define una variable como perteneciente a ese tipo, además de un procedimiento en ese lenguaje para cada operación del TDA.

Con Arreglos

Así que la implementación de una lista se puede hacer por medio de arreglos, donde los elementos se almacenan en celdas contiguas de un arreglo. Esta representación permite recorrer con facilidad una lista y agregarle elementos nuevos al final. Pero insertar un elemento a la mitad de la lista obliga a desplazarse una posición dentro del arreglo a todos los elementos que siguen al nuevo elemento para concederle espacio. De la misma forma, la eliminación de un elemento, excepto el último, requiere desplazamientos de elementos para llenar de nuevo el vacío formado. Para representar una lista por medio de un arreglo, se define a la lista utilizando arreglos, de la siguiente manera: Ya que el arreglo es una consecución de celdas, el arreglo debe estar formado por una primer celda, segunda celda, ..., última celda. Para identificar al arreglo se debe tener la longitud del arreglo, así que esto se define como sigue:

```
int longitud_maxima, c;
int elemento_ultimo;
int elemento_posición;
```

```
int posición_ultima;
LISTA L;

 longitud_maxima = c \\Es una constante apropiada.

 L = ARREGLO[longitud_maxima + 1];

 función FIN(L)
 {
 return (posición_ultima + 1)
 }
```

Por lo que para poder implementar las operaciones INSERTA(), SUPRIME(), y LOCALIZA(), se hace de la siguiente forma:

```
 Procedimiento INSERTA(x, p, L)
  \\INSERTA coloca a x en la posición p de la lista L.
   LISTA L;
   int x, p, q;

   p = tipo_posición; \\p será la posición actual en L, y q posición que
   q = tipo_posición; \\avanza para encontrar elementos iguales.

   Comenzar INSERTA
 (1)    Si p = > longitud_maxima Entonces
 (2)      error(La lista esta llena);
 (3)    o Si p > longitud_maxima + 1 o Si p < 1 Entonces
 (4)      error(La posición no existe);
 (N)    o Comenzar
 (5)      for q = p; q = longitud_maxima; 1++
 (N)            Hacer
 (6)                 q = SIGUIENTE(p, L);
 (7)                 ARREGLO[p] = x; \\Coloca a x en ARREGLO[p].
        fin
   Fin INSERTA

 Procedimiento SUPRIME(p, L)
  \\SUPRIME elimina el elemento en la posición p de la lista L.
```

```
  tipo_posición p, q. \\p será la posición actual en L, y q posición que
                      avanza para encontrar elementos iguales.
  comenzar SUPRIME
(1)    p = PRIMERO(L);
(2)      mientras p ¡= FIN(L) comenzar a hacer
(3)       q = SIGUIENTE(p, L);
(4)         mientras q ¡= FIN(L) hacer
(5)            si MISMO(RECUPERA(p, L), RECUPERA(q, L)) entonces
(6)                 SUPRIME(q,L)
(N)            o hacer
(7)                 q == SIGUIENTE(q, L)
(8)            p = SIGUIENTE(p, L)
               fin
  fin SUPRIME

Procedimiento LOCALIZA(x, L)
 \\LOCALIZA devuelve la posición de x en la lista L.
  tipo_posición p, q. \\p será la posición actual en L, y q posición que
                      avanza para encontrar elementos iguales.
  comenzar LOCALIZA
(1)    p = PRIMERO(L);
(2)      mientras p ¡= FIN(L) comenzar a hacer
(3)       q = SIGUIENTE(p, L);
(4)         mientras q ¡= FIN(L) hacer
(5)            si MISMO(RECUPERA(p, L), RECUPERA(q, L)) entonces
(6)                 SUPRIME(q,L)
(N)            o hacer
(7)                 q == SIGUIENTE(q, L)
(8)            p = SIGUIENTE(p, L)
               fin
  fin LOCALIZA
```

Con Apuntadores

Un apuntador es una celda cuyo valor indica o señala a otra. Cuando se representa gráficamente estructuras de datos, el hecho de que una celda A sea un apuntador a la celda B se indica con una flecha de A a B. En el lenguaje de programación C se representa que A apunta a B, como:

```
tipo_dato *A, B; \\Declaración de datos.

A = &B; \\A apunta a B. Ya que la dirección de B
          se asigna a la variable apuntador A.
```

Esta implementación permite eludir el empleo de memoria contigua para almacenar una lista y, por lo tanto, también elude los desplazamientos de elementos para hacer inserciones o rellenar vacíos creados por la eliminación de elementos. No obstante, hay que pagar el precio de un espacio adicional para los apuntadores. Para implementar los procedimientos INSERTA(), SUPRIME(), LOCALIZA(), y ANULA(), se hace lo siguiente:

```
Procedimiento INSERTA(x, L)
 \\INSERTA devuelve la posición de x en la lista L.
  tipo_posición p, q. \\p será la posición actual en L, y q posición que
                      avanza para encontrar elementos iguales.
  comenzar INSERTA
(1)    p = PRIMERO(L);
(2)      mientras p ¡= FIN(L) comenzar a hacer
(3)       q = SIGUIENTE(p, L);
(4)         Mientras q ¡= FIN(L) Hacer
(5)            Si MISMO(RECUPERA(p, L), RECUPERA(q, L)) Entonces
(6)                 SUPRIME(q,L)
(N)            Hacer
(7)                 q == SIGUIENTE(q, L)
(8)            p = SIGUIENTE(p, L)
            fin
  fin INSERTA

Procedimiento SUPRIME(x, L)
 \\SUPRIME devuelve la posición de x en la lista L.
  tipo_posición p, q. \\p será la posición actual en L, y q posición que
                      avanza para encontrar elementos iguales.
  comenzar SUPRIME
(1)    p = PRIMERO(L);
(2)      mientras p ¡= FIN(L) comenzar a hacer
```

```
(3)       q = SIGUIENTE(p, L);
(4)         mientras q ¡= FIN(L) hacer
(5)            si MISMO(RECUPERA(p, L), RECUPERA(q, L)) entonces
(6)                 SUPRIME(q,L)
(N)            o hacer
(7)                 q == SIGUIENTE(q, L)
(8)            p = SIGUIENTE(p, L)
               fin
  fin SUPRIME

Función LOCALIZA(x, L)
 \\LOCALIZA devuelve la posición de x en la lista L.
  tipo_posición p, q. \\p será la posición actual en L, y q posición que
                      avanza para encontrar elementos iguales.
  comenzar LOCALIZA
(1)    p = PRIMERO(L);
(2)      mientras p ¡= FIN(L) comenzar a hacer
(3)       q = SIGUIENTE(p, L);
(4)         mientras q ¡= FIN(L) hacer
(5)            si MISMO(RECUPERA(p, L), RECUPERA(q, L)) entonces
(6)                 SUPRIME(q,L)
(N)            o hacer
(7)                 q == SIGUIENTE(q, L)
(8)            p = SIGUIENTE(p, L)
               fin
  fin LOCALIZA

Función ANULA(x, L)
 \\ANULA devuelve la posición de x en la lista L.
  tipo_posición p, q. \\p será la posición actual en L, y q posición que
                      avanza para encontrar elementos iguales.
  comenzar ANULA
(1)    p = PRIMERO(L);
(2)      mientras p ¡= FIN(L) comenzar a hacer
(3)       q = SIGUIENTE(p, L);
(4)         mientras q ¡= FIN(L) hacer
(5)            si MISMO(RECUPERA(p, L), RECUPERA(q, L)) entonces
(6)                 SUPRIME(q,L)
```

```
(N)            o hacer
(7)                 q == SIGUIENTE(q, L)
(8)            p = SIGUIENTE(p, L)
           fin
  fin ANULA
```

Con Cursores

Un cursor es una celda de valor entero que se utiliza como apuntador a un arreglo. Como métodos de conexión, cursores y apuntadores son en esencia lo mismo. Debido a que algunos lenguajes no tienen apuntadores, los apuntadores se pueden simular con cursores, esto es con enteros que indican posiciones en arreglos. Las celdas del arreglo ESPACIO, se define como:

La función MUEVE, se representa como:

```
Función ANULA(x, L)
 \\ANULA devuelve la posición de x en la lista L.
  tipo_posición p, q. \\p será la posición actual en L, y q posición que
                      avanza para encontrar elementos iguales.
  comenzar ANULA
(1)    p = PRIMERO(L);
(2)      mientras p ¡= FIN(L) comenzar a hacer
(3)       q = SIGUIENTE(p, L);
(4)         mientras q ¡= FIN(L) hacer
(5)            si MISMO(RECUPERA(p, L), RECUPERA(q, L)) entonces
(6)                 SUPRIME(q,L)
(N)            o hacer
(7)                 q == SIGUIENTE(q, L)
(8)            p = SIGUIENTE(p, L)
           fin
  fin ANULA
```

El procedimiento INSERTA:

```
Función ANULA(x, L)
 \\ANULA devuelve la posición de x en la lista L.
  tipo_posición p, q. \\p será la posición actual en L, y q posición que
```

```
                    avanza para encontrar elementos iguales.
  comenzar ANULA
(1)    p = PRIMERO(L);
(2)      mientras p ¡= FIN(L) comenzar a hacer
(3)       q = SIGUIENTE(p, L);
(4)         mientras q ¡= FIN(L) hacer
(5)            si MISMO(RECUPERA(p, L), RECUPERA(q, L)) entonces
(6)                 SUPRIME(q,L)
(N)            o hacer
(7)                 q == SIGUIENTE(q, L)
(8)            p = SIGUIENTE(p, L)
               fin
  fin ANULA
```

El procedimiento SUPRIME:

```
Función ANULA(x, L)
 \\ANULA devuelve la posición de x en la lista L.
  tipo_posición p, q. \\p será la posición actual en L, y q posición que
                    avanza para encontrar elementos iguales.
  comenzar ANULA
(1)    p = PRIMERO(L);
(2)      mientras p ¡= FIN(L) comenzar a hacer
(3)       q = SIGUIENTE(p, L);
(4)         mientras q ¡= FIN(L) hacer
(5)            si MISMO(RECUPERA(p, L), RECUPERA(q, L)) entonces
(6)                 SUPRIME(q,L)
(N)            o hacer
(7)                 q == SIGUIENTE(q, L)
(8)            p = SIGUIENTE(p, L)
               fin
  fin ANULA
```

El procedimiento VALOR_INICIAL:

```
Función ANULA(x, L)
 \\ANULA devuelve la posición de x en la lista L.
  tipo_posición p, q. \\p será la posición actual en L, y q posición que
```

```
                    avanza para encontrar elementos iguales.
  comenzar ANULA
(1)    p = PRIMERO(L);
(2)      mientras p ¡= FIN(L) comenzar a hacer
(3)       q = SIGUIENTE(p, L);
(4)         mientras q ¡= FIN(L) hacer
(5)            si MISMO(RECUPERA(p, L), RECUPERA(q, L)) entonces
(6)                 SUPRIME(q,L)
(N)            o hacer
(7)                 q == SIGUIENTE(q, L)
(8)            p = SIGUIENTE(p, L)
             fin
  fin ANULA
```

Cuando se requiere seguir con la búsqueda de elementos en la lista se hace por medio de apuntadores aplicando el siguiente pseudocódigo:

```
1 TIPE am:↑ am ;
2 TIPE ↑ am: listaEstructura; ↑ am := ↑ NL ;
3 while am ↑.sig == NULL do
4 |_ ↑ am := am ↑.sig
```

La búsqueda por valor se hace con el siguiente pseudocódigo:

```
1 TIPE am:↑ am ;
2 TIPE ↑ am: listaEstructura ;
3 ↑ am := ↑ NL ;
4 if am ↑.dato == valor then
5 |  Busqueda completada
6 else
7 |_ ↑ am := am ↑.sig
```

Si se requiere insertar un elemento, se deben crear dos apuntadores el

apuntador móvil (am) y el apuntador móvil posterior (amp):

```
1 TIPE amp:↑ amp;
2 TIPE ↑ amp: listaEstructura;
3 TIPE NuevoElemento:↑ NuevoElemento ;
4 TIPE ↑ NuevoElemento: listaEstructura;
5 ↑ amp := asignar direccon posterior;
6 ↑ NuevoElemento := NEW listaEstructura;
7 am ↑.sig := NuevoElemento;
8 NuevoElemento.sig := ↑ amp;
```

3.4.5. Listas doblemente enlazadas

Una lista doblemente enlazada tiene una estructura de la siguiente forma:

Algorithm 3.10: Declaración de datos de una lista doblemente enlazada

```
1 TIPE EstListDoble = REGISTRO
2 {
3 TIPE dato:TIPO_DATO;
4 TIPE ↑ sig:EstListDoble;
5 TIPE ↑ ant:EstListDoble;
6 }
7 TIPE ↑ LD: EstListDoble;
```

Donde tiene un apuntador para el siguiente elemento y otro para el elemento anterior, esto hace que la lista se pueda recorrer hacia adelante o hacia atras sin ningún problema, lo que no se puede hacer tan fácilmente en una lista simplemente enlazada. La construcción de la lista doblemente enlazada

se hace de la siguiente manera:

Algorithm 3.11: Construcción de una lista doblemente enlazada

```
1  TIPE EstListDoble = REGISTRO
2  {
3  TIPE Info:TIPO_DATO;
4  TIPE ↑ sig:EstListDoble;
5  TIPE ↑ ant:EstListDoble;
6  }
7  TIPE ↑LD, ↑NuevoElemento, ↑m:EstListDoble;
8  TIPE n:ENTERO;
9  n := Leer_De_Teclado(ENTERO);
10 ↑LD := NULL;
11 while n > 0 do
12     NuevoElemento := NEW EstListDoble;
13     NuevoElemento.Info := Leer_De_Teclado(TIPO_DATO);
14     if ↑LD == NULL then
15         ↑LD := NuevoElemento;
16         NuevoElemento.sig := NULL;
17         NuevoElemento.ant := NULL;
18         ↑m := ↑LD;
19     else
20         m↑.sig := NuevoElemento;
21         NuevoElemento.ant := ↑m;
22         NuevoElemento.sig := NULL;
23         ↑m := m↑.sig;
24     n := n - 1;
```

3.5. Tablas Hash

Una tabla Hash es una estructura bidimensional que cumple con una función para poder determinar la posición del elemento. La función hash es de tal forma que la transformación es una clave que da un número entero único. Para construir una tabla Hash

Una tabla hash guarda direcciones, las cuales son calculadas por medio de una función también llamada hash. La tabla se representa como $A[h(x)]$. Para hacer el cálculo se consideran los siguientes valores, x es el valor del elmento a

guardar, m es el valor de la localidad calculada por medio de la función hash $h(x) = m$. Donde n es el tamaño de la tabla y su valor se considerará $m \geq n$. La creación de la tabla hash se hace por medio del siguiente pseudocódigo:

Algorithm 3.12: Creación de una tabla hash por medio de una lista doblemente enlazada

```
1  TIPE TablaHash = REGISTRO
2  {
3  TIPE dato:TIPO_DATO;
4  TIPE col:ENTERO;
5  TIPE reg:ENTERO;
6  TIPE ↑ sig:TablaHash;
7  TIPE ↑ ant:TablaHash;
8  }
9  TIPE ↑TH, ↑NuevoElemento, ↑m: TablaHash;
10 ↑TH := NULL;
11 n := Escribir_Por_Teclado(ENTERO);
12 while n > 0 do
13     NuevoElemento := NEW TablaHash;
14     NuevoElemento.dato := NULL;
15     NuevoElemento.col := Escribir_Por_Teclado(ENTERO);
16     NuevoElemento.reg := Escribir_Por_Teclado(ENTERO);
17     if ↑TH == NULL then
18         ↑TH:= NuevoElemento;
19         NuevoElemento.sig := NULL;
20         NuevoElemento.ant := NULL;
21         ↑m := ↑LD;
22     else
23         m↑.sig := NuevoElemento;
24         NuevoElemento.ant := ↑m;
25         NuevoElemento.sig := NULL;
26         ↑m := m↑.sig;
27     n := n -1;
28 A[h(x)] := ↑TH;
```

Existe una función de ordenamiento, pues en caso de no meter la tabla en orden. Por medio del siguiente pseudocódigo se detecta si la tabla esta en

orden o de cualquier manera se ordena.

Algorithm 3.13: Función para ordenar una tabla hash implementada por medio de una lista doblemente enlazada

```
1 TIPE ↑m, ↑p, ↑q:TablaHash;
2 while ↑m <> NULL do
3 |   ↑p := m↑.sig;
4 |   while ↑p <> NULL do
  |   |       if m↑.col < p↑. col then
  |   |          Intercambiar(columna, renglon);
  |   |       else if m↑.col == p↑.col then
  |   |          if m↑.reg > n↑.reg then
  |   |   |  Intercambiar(renglones);
  |   |       else
  |   |          ↑n := n↑.sig;
  |   |       end if
  |   |   ↑p := ↑p.sig;
5 |   ↑m := m↑.sig;
6 |   ↑p := p↑.sig;
```

Este programa entrega una lista doblemnete enlazada ↑TH, en la cual se van a guardar los datos de acuerdo al cálculo de direcciones por medio de una función hash.

3.5.1. Tipos de funciones hash

Las nuevas funciones hash $h(x)$ aún se reportan, pero las más usuales son:

Las funciones hash por división. Estas se calculan como, $h(x) = x \ mod \ m$. Dentro de estas, estan las funciones cuadráticas: $h(x) = x^2 \ mod \ m$.

Las funciones hash por multiplicación. Estas se calculan como, $h(x) = \lfloor mxA \rfloor \ mod \ m$.

3.5.2. Operaciones de una tabla hash

Las opercaiones de las tablas hash son insertar, búscar y eliminar. Las cuales se implementan de la manera siguiente:

Algorithm 3.14: Insertar valores en la tabla al calcular la dirección por medio de la función hash adecuada

```
1 TIPE ↑m:TablaHash;
2 TIPE x:TIPO_DATO;
3 TIPE n:ENTERO;
4 ↑m := ↑TH;
5 n := h(x);
6 while n > 0 do
7 |   ↑m := m↑.sig;
8 |   regresar(↑m);
9 m↑.dato := x;
```

3.5.3. Colisiones

Cuando al aplicar la función hash se obtiene el mismo valor para dos datos diferentes, se dice que ocurrió una colisión. Las colisiones se evitan aplicando las técnicas de direccionamiento abierto que consiste en secuencialmente seguir avanzando hasta encontrar una posición vacia, la técnica de encadenamiento que consiste en poner en una cadena los datos que tienen la misma dirección y la técnica de hash perfecto.

3.5.4. Listas enlazadas

Se construye una lista enlazada, se crean dos apuntadores y se intercambia, se avanzan los apuntadores creados y se hace la comparación y si es necesario se hace el intercambio, hasta llegar al final de la lista. Esto se puede meter a un ciclo For con el indice $i :=$ *numero de elementos* $- 1$;. Para el caso de la lista anterior el *numero de elementos* $:= 11$;, así que el for se repetirá 10 veces.

Las funciones se escriben de la siguiente manera:

```
1 funcion Intercambiar:TIPO_DATO ;
2 var ↑ p_a, ↑ p_1, ↑ p_2:pila ;
3 begin
4 ↑ p_a := ↑ p_1 ;
5 ↑ p_1 := ↑ p_2 ;
6 ↑ p_2 := ↑ p_a ;
7 return ↑ p_1, ↑ p_2 ;
8 end
```

Capítulo 4

Recursividad y Estructuras de Datos No Lineales

4.1. Recursividad

La iteración ocupa menos memoria que la recursión. Una iteración es el cálculo de la función con un cambio de variable local una cantidad repetida de veces. La diferencia se puede apreciar calculando el factorial de las dos formas:

El caso base es el caso para el que no se aplica la recursividad. El caso general es para el cual la solución es expresada en términos de una pequeña versión de si misma.

La recursividad es el llamado de la función por si misma, que se aplica en el cálculo de permutaciones comoa a continuación se muestra:

Para un conjunto de elementos $n \geq 1$, imprimir todas las posible permutaciones. Si tenemos el conjunto $\{a\}$ así las permutaciones son, $\{(a)\}$. Si ahora tenemos el conjunto $\{a, b\}$ las permutaciones son las siguientes, $\{(a, b), (b, a)\}$. Si el conjunto ahora se compone de $\{a, b, c\}$, luego el conjunto de permutaciones contiene $3! = 6$ elementos, las cuales son:

$$\{(a, b, c), (a, c, b), (b, a, c), (b, c, a), (c, a, b), (c, b, a)\}$$

Si ahora las permutaciones resultantes de cada uno de los conjuntos comenzando con el conjunto $\{a, b\}$ las escribimos de la manera siguiente:

1. a seguido de las permutaciones de $\{b\}$.

2. b seguido de las permutaciones de $\{b\}$.

Ahora para el conjunto $\{a, b, c\}$, se escribe de la forma siguiente:

1. a seguido de las permutaciones de $\{b, c\}$.
2. b seguido de las permutaciones de $\{a, c\}$.
3. c seguido de las permutaciones de $\{a, b\}$.

Y para el conjunto a, b, c, d, se escribe de manera de proceso:

1. a seguido de las permutaciones de $\{b, c, d\}$.
2. b seguido de las permutaciones de $\{a, c, d\}$.
3. c seguido de las permutaciones de $\{a, b, d\}$.
4. d seguido de las permutaciones de $\{a, b, c\}$.

De la descripción anterior del algoritmo se puede ver que esta forma de presentar al algoritmo, se hace por recursión que es la llamada de una función a si misma. Si se usa un arreglo para guardar el conjunto de elementos a permutar, o sea en $A[1]$ se guarda a, en $A[2]$ se guarda b, en $A[3]$ se guarda c y en $A[4]$ se guarda d. Así que explicando en el conjunto $\{a, b, c, d\}$, aplcando los arreglos se presenta de la siguiente forma:

1. $A[1]$ que tiene el valor a seguido de las permutaciones de $\{b, c, d\}$.
2. $A[2]$ que tiene el valor b seguido de las permutaciones de $\{a, c, d\}$.
3. $A[3]$ que tiene el valor c seguido de las permutaciones de $\{a, b, d\}$.
4. $A[4]$ que tiene el valor d seguido de las permutaciones de $\{a, b, c\}$.

Así que si esto se expresa por medio de pseudocódigo se hace de la siguiente manera:

```
ALGORITMO permutacion(A, k, n)

  IF k = n THEN
   Escribir(A[k ... n]);
  ELSE
   FOR i := k To n DO
     t := A[k];
     A[k] := A[i];
     A[i] := t;
     permutaciones(A, k + 1, n);
   END FOR
 END IF
END ALGORITMO
```

Donde A es un arreglo, k es el número de elementos y n es el número de elementos.

```
1 funcion() ;
```

Otro ejemplo de estructura recursiva es el árbol genealógico. Esta definición conduce inevitablemente a una estructura infinita.

```
TYPE arb = REGISTRO
              IF conocido THEN
                 (nombre:alfa;
                  padre, madre: arb)
           END
```

El pseudocódigo aplicado a un ejemplo es al siguiente:

```
x = (T, Ted, (T, Fred, (T, Adam, (F),
    (F)), (F)),(T, Mary, (F), (T, Eva,
    (F), (F)))
```

Donde también se ve la recursividad. Todo procedimiento recursivo debe contener necesariamente una instrucción condicional para que su ejecución pueda terminar. La terminación del proceso debe corresponder con la cardinalidad finita.

La facultad de las estructuras recursivas es la facilidad de variar su tamaño a diferencia de las estructuras fundamentales como los arrays, registros, conjuntos. De esto se desprende que no es posible asignarle una cantidad fija de memoria y como consecuencia un compilador no puede asociar direcciones explicitas con los componentes de tales variables. Para resolver este problema es realizar una asignación dinámica de memoria, que quiere decir asignar memoria para los componentes individuales al tiempo al tiempo que estos son creados durante la ejecución del programa en vez del momento de la compilación. El compilador asigna una cantidad fija de memoria para mantener la dirección del componente asignado dinámicamente. Esto se ve en el árbol genealógico.

4.1.1. Recursividad directa e indirecta

Existen dos tipos de recursividad la recursividad directa y la recursividad indirecta. La recursividad directa es cuando la función se llama a si mismo directamente. La recursividad indirecta es cuando una cadena de dos o más funciones llaman a la función que originó a la cadena.

4.1.2. Backtracking

Es un método que intenta completar una búsqueda para solucionar un problema construyendo una solución parcial, siempre asegurando que la solución sea consistente con los requerimientos del problema. El algoritmo intenta extender una solución parcial hacia la completitud, pero cuando una inconsistencia ocurre con los requerimientos del problema, el algoritmo se regresa removiendo la parte más recientemente construida y trata con otra posibilidad.

4.2. Arboles binarios

Un árbol binario puede ser un árbol vacio o un árbol que consiste de dos subárboles binarios que tienen como padre a la raíz y que son un subárbol izquierdo y un subárbol derecho. El árbol binario puede ser de un nodo, donde el nodo será la raíz y el subárbol izquierdo y el subárbol derecho serán nulos. Puede ser de dos nodos donde uno será la raíz y el otro nodo puede ser el

subárbol izquierdo o el subárbol derecho y el otro estará vacio. Puede ser de tres nodos, un nodo colocado en el subárbol izquierdo y el otro nodo colocado en el subárbol derecho por lo que estará completo, pero si los nodos se colocan en el subárbol izquierdo o en el subárbol derecho estará desbalanceado. Y así sucesivamente de cuatro, cinco, o n nodos.

Si se tiene que crear el siguiente árbol, como ejemplo se procede de la siguiente forma:

4.2.1. Construcción de un árbol binario

Un árbol binario es un árbol donde ningún nodo tine más de dos hijos. Los hijos se describen como hijo izquierdo e hijo derecho que dependen de un nodo padre. Un árbol binario T se define como un conjunto de elementos finito, llamados nodos, tales que:

1. T es vacio si tiene nodos llamados árbol null o árbol vacio.

2. T contiene un nodo especial R, llamado nodo raíz de T, y los nodos restantes de T forman un par ordenado de árboles binarios disjuntos T1 y T2, y son llamados subárbol izquierdo y subárbol derecho de R. Si T1 es no vacio luego su raíz es el sucesor izquierdo de R, similarmente T2 es no vacio luego su raíz es el sucesor derecho de R.

Existe el árbol binario y el árbol extricatamente binario o 2-árbol o árbol binario extendido, el cual contien 2(nḿero de hojas) - 1, o de otra manera tine nodos que terminan con dos nodos hijos o con ningún hijo.

La principal diferencia entre un árbol binario y un árbol ordinario es:

1. Un árbol binario puede ser vacio y un árbol ordinario no.

2. Cada elemento de un árbol binario tiene exactamente dos subárboles (uno o ambos de estos subárboles puede ser vacio). Cada elemento en un árbol puede tener cualquier número de subárboles.

3. Los subárboles de cada elemento en un árbol binario son ordenados, subárbol izquierdo y subárbol derecho. Los subárboles en un árbol son desordenados.

La representación del árbol binario se hace de dos formas:

1. Por medio de arreglos.

2. Por medio de listas enlazadas.

La estructura de árbol esta constituida de una raíz y de hojas, así que la estructura básica para construir un árbol por medio de una lista simplemente enlazada:

```
1 TIPE Dr:↑ Dr ;
2 TIPE Iz:↑ Iz ;
3 TIPE estructuraArbol = REGISTRO
4 {
5 TIPE dato:TIPO_DATO ;
6 TIPE Dr:estructuraArbol ;
7 TIPE Iz:estructuraArbol ;
8 }
```

Para construir el árbol A:

```
1  TIPE A:↑ A ;
2  TIPE ↑ A:estructuraArbol ;
3  ↑ Arbol(n:ENTERO):estructuraArbol
4  {
5  leer(n) ;
6  if n == 0 then
7  |  ↑ A := NULL ;
8  else
9  |  ni := n div 2 ;
10 |  nd := n - ni - 1 ;
11 |  leer(x) ;
12 |  nuevoElemento := NEW estructuraArbol ;
13 |  nuevoElemento.dato := x ;
14 |  nuevoElemento.Iz := Arbol(ni) ;
15 |  nuevoElemento.Dr := Arbol(nd) ;
16 |  ↑ A := nuevoElemento ;
17 }
```

Se quiere obtener el factorial de un número, esto se hace aplicando la propiedad de la recursividad de las funciones:

```
1 leer(n) ;
2 Factorial(n:ENTERO):ENTERO
3 {
4 if n > 0 then
5 |  F := n * Factorial(n - 1) ;
6 else
7 |_ F := 1 ;
8 return(F) ;
9 }
```

Se quiere crear un árbol con n elementos o nodos:

```
1  ↑crearArbol(↑ raiz:estructuraArbol):estructuraArbol
2  {
3  leer(n) ;
4  if n == 0 then
5  |  ↑ raiz := NULL ;
6  else
7  |  while n <> NULL do
8  |  |  if n <> NULL then
9  |  |  |  ↑ raiz := inserta(n, raiz) ;
10 return(↑ raiz) ;
11 }
12 ↑inserta(n:ENTERO,↑ raiz:estructuraArbol):estructuraArbol
13 {
14 p := NEW estructuraArbol ;
15 p.dato := leer(valor) ;
16 p.Iz := NULL ;
17 p.Dr := NULL ;
18 if raiz == NULL then
19 |  ↑ raiz := p ;
20 else
21 |  ↑ temp1 := ↑ raiz ;
22 |  while ↑ temp1 <> NULL do
23 |  |  ↑ temp2 := ↑ temp1 ;
24 |  |  if p.dato < temp1 ↑.dato then
25 |  |  |  ↑ temp1 := temp1 ↑.Iz ;
26 |  |  else
27 |  |  |  if p.dato > temp1 ↑.dato then
28 |  |  |  |  ↑ temp1 := temp1 ↑.Dr ;
29 |  |  |  else
30 |  |  |  |  imprimir(Duplicado) ;
31 |  |  |  |  free(p) ;
32 if p.dato > temp2 ↑.dato then
33 |  temp2↑.Iz := p ;
34 else
35 |  temp2↑.Dr := p ;
36 }
```

```
node *createtree(node *root)
{
int n;
do{ printf(\\nEnter number<0 to stop>:");
scanf(\%d",&n);
if(n!=0)
root= insert(n,root);
}while(n!=0);
return(root);
}
node *insert(int n,node *root)
{
node *temp1=NULL;
node *temp2=NULL;
node *p=NULL;
p=(node *)malloc (sizeof(node));//dynamic allocation of memory foe each element
p->data=n; //initialize contents of the structure
p->lptr=NULL;
p->rptr=NULL;
//A new node has been created now our task is to insert this node
//in the appropriate place. If this is the first node to be created
//then this is the root of the tree.
if(root==NULL)
root=p;
else
// We will use temp1 for traversing the tree.
// Temp2 will be traversing parent
// p is the new node we have created.
{ temp1=root;
while(temp1!=NULL)
{ temp2=temp1; // store it as parent
// Traverse through left or right sub tree
if(p->data < temp1->data)
temp1 = temp1->lptr; // left subtree
else
if(p->data > temp1->data)
temp1 = temp1->rptr; // right sub tree
else
```

```
{
printf(\\n\tDUPLICATE VALUE");
free(p);
break;
}//end else
} //end of while
// we have traversed to the end of tree
// node ready for insertion
if(temp1 == NULL)
{ // attach either as left son or right son of parent temp2
if(p->data<temp2->data)
temp2->lptr=p; // attach as left son
else
temp2->rptr=p; // attach as right son
}
printf(\\n successful insertion\n");
}//end of else
return(root);
}//end of create tree
```

4.2.2. Recorridos de un árbol

Existen tres tipos de recorridos en un árbol binario. Preorden, Inorden y Postorden.

El recorrido del árbol es de la siguiente forma:

1. Preorden, se visita la raíz, luego se visita el subárbol izquierdo y al final se visita el subárbol derecho.

2. Inorden, se visita el subárbol izquierdo luego se visita la raíz y al final el subárbol derecho.

3. Postorden, se visita el subárbol izquierdo luego se visita el subárbol derecho y al final la raíz.

Para hacer las funciones de recorrido en un árbol binario se utiliza la recursividad de la siguiente manera. Así que para poder hacer los recorridos

del árbol en preorden, se usa el siguiente pseudocódigo:

Algorithm 4.1: Función preorden

```
1 funcion preorden():estArbol;
2 Imprimir(raiz);
3 preorden(subArbolIzquierdo);
4 preorden(subArbolDerecho);
```

Así que para poder hacer los recorridos del árbol en Inorden, se usa el siguiente pseudocódigo:

Algorithm 4.2: Función Inorden

```
1 funcion Inorden():estArbol;
2 Inorden(subArbolIzquierdo);
3 Imprimir(raiz);
4 Inorden(subArbolDerecho);
```

Así que para poder hacer los recorridos del árbol en Postorden, se usa el siguiente pseudocódigo:

Algorithm 4.3: Función Postorden

```
1 funcion postorden():estArbol;
2 Inorden(subArbolIzquierdo);
3 Inorden(subArbolDerecho);
4 Imprimir(raiz);
```

4.2.3. Arboles de Expresión

Un árbol de expresión se construye de una sentencia que contienen a opernados y operadores, los operandos se colocan en las hojas del árbol y los operadores se colocan en los nodos padres de las hojas del árbol. Si la operación es una idualdad el subárbol izquierdo contendra a la variable, la raíz el signo de igualdad y el subárbol derecho contendra el resto de la expresión.

4.2.4. Árboles de Búsqueda

Un arbol binario de buque da o ABB, es un arbol binario en el cual para todo elemento, los elementos mayores a él, se ubican en su rama derecha, mientras que los elementos menores van en su rama izquierda. Cada elemento se almacena una sola vez por lo que no existen elementos repetidos.

Ya con estas definiciones claras sobre arboles; ahora estos son conceptos generales de lo que es un arbol, para poder implementarlos en lenguaje C++ tenemos que tener conocimientos previos sobre listas enlazadas y su implementación.

Cada elemento(nodo) de un árbol ABB cuenta con tres campos:

1. Dato(numero, letra, palabra, etc), en este caso usaremos un numero(entero).
2. Puntero al nodo derecho.
3. Puntero al nodo izquierdo.

Los punteros tienen que ser del tipo arbol, ya que apuntaran a un nodo del mismo tipo, este seria un ejemplo de como se seria el tipo arbol ABB.

4.3. Árbol Equilibrado de Búsqueda

La creación de un árbol binario balanceado, se puede hacer de dos formas. Una de manera recursiva como a continuación se muestra en el siguiente pseudocódigo:

4.3.1. Rotación simple

Las rotaciones se dividen en dos casos, simples o dobles. Las simples se conocen como LL si son de la rama izquierda, y RR si son de la rama derecha. Las rotaciones dobles son LR o RL, ya que son combinadas y más complejas como sugiere su nombre. Las rotaciones simples.

En el caso simple LL, tomemos un nodo 1 que quedará con FB = 2, insertamos un nodo en el subárbol izquierdo de su hijo izquierdo, con lo que tenemos una inserción en la izquierda-izquierda (por eso Left Left). Para balancear el árbol deberemos tomar el hijo izquierdo de 1 (llamémoslo 2), y ponerlo como padre de 1, el hijo derecho de 1 queda en su lugar, el hijo izquierdo de 2 queda en su lugar y el hijo de derecho de 2 debe ir como hijo izquierdo de 1, ya que en su lugar tendremos a 1.

Lo que hacemos es mover el subárbol que queda con un largo excesivo hacia arriba, sacando al padre para hacer lugar, teniendo cuidado eso si de mantener la relación de mayor menor, lo cual hacemos dejando los subárboles

```
1  TIPE EstArbol = REGISTRO
2  {
3  TIPE dato:TIPO_DATO;
4  TIPE Dr:EstArbol;
5  TIPE Iz:EstArbol;
6  }
7  TIPE arbol(ENTERO):EstArbol;
8  TIPE ↑NuevoElemento, ↑raiz:EstArbol;
9  TIPE n, n_i, n_d:ENTERO;
10 n := Leer_De_Teclado(ENTERO);
11 ↑raiz := NULL;
12 if ↑raiz == NULL then
13 |  arbol := NULL;
14 else
15 |  n_i := n DIV 2;
16 |  n_d := n - n_i - 1;
17 |  NuevoElemento := NEW EstArbol;
18 |  NuevoElemento.dato := Leer_De_Teclado(TIPO_DATO);
19 |  NuevoElemento.Iz := arbol(n_i);
20 |  NuevoElemento.Dr := arbol(n_d);
21 |  arbol := NuevoElemento;
```

izquierdos de 2 y derecho de 1 en sus lugares (ya que son menores que todos los demas, y mayores respectivamente), y moviendo de lugar el subárbol derecho de 2 hacia el izquierdo de 1. Esto lo podemos hacer ya que este subárbol contiene elementos mayores que los de 2, pero menores que 1.

En el caso de la rotación simple RR sucede lo mismo que acabamos de ver, solo que en el caso de insertar a la derecha del hijo derecho:

Hemos visto los dos casos simples, donde insertamos en la rama menor que todos o mayor que todos. Veremos ahora que sucede cuando insertamos en el hijo derecho del subárbol izquierdo, o en el izquierdo del derecho, es decir cruzado.

En este caso lo que haremos será aplicar dos rotaciones simples, con lo cual haremos que el nodo insertado "suba" acortando la altura del árbol. Noten que el nodo que tiene FB = 2 es el nodo 1, y no el nodo 2 que es donde insertamos (en su subárbol derecho). Por lo cual deberemos modificar la altura de sus hijos para reducir el FB a 1.

El proceso que seguiremos será el siguiente, tomamos el nodo en la rama donde se generó el desbalance, en este caso el 2, como la inserción es en el fijo derecho de 2, tomamos este nodo (el 3), y lo guardamos como la nueva raiz. Guardamos los dos hijos del 3 (ya que en realidad el árbol puede ser más amplio, y haber nodos hacia "abajo") y ponemos como su hijo izquierdo a 2, y su hijo derecho a 1. Así sabemos que el orden se mantiene, 3, estaba a la derecha de 2, por lo que es mayor, y a la izquierda de 1, por lo que es menor. Colocado 3 en el lugar de 1, tomamos los dos posibles hijos de 3 que guardamos y los colocamos respectivamente como hijo derecho de 2 al hijo izquierdo, y como hijo izquierdo de 1 al derecho de 3. Otra vez manteniendo el orden.

En resumen la idea es, cuando detectamos que un nodo tiene FB = 2 (recuerden, en valor absoluto), entonces de acuerdo a sea positivo o negativo tomamos la rama izquierda o la derecha, subimos el nodo correspondiente y reasignamos con cuidado los hijos para no perder el orden. Las operaciones LR y RL, como las LL y RR son simétricas, por lo que el código será idéntico cambiando el hijo seleccionado. Como mencionabamos antes, las operaciones complejas se pueden reducir a dos operaciones simples, por ejemplo la LR podemos reducrila a realziar primero una RR en el hijo izquierdo y luego una LL en la raíz.

4.3.2. Rotación doble

Rotación doble a la derecha (DD)

Esta rotación se usará cuando el subárbol izquierdo de un nodo sea 2 unidades más alto que el derecho, es decir, cuando su FE sea de -2. Y además, la raíz del subárbol izquierdo tenga una FE de 1, es decir, que esté cargado a la derecha. Este es uno de los posibles árboles que pueden presentar esta estructura, pero hay otras dos posibilidades. El nodo R puede tener una FE de -1, 0 ó 1. En cada uno de esos casos los árboles izquierdo y derecho de R (B y C) pueden tener alturas de n y n-1, n y n, o n-1 y n, respectivamente. El modo de realizar la rotación es independiente de la estructura del árbol R, cualquiera de las tres produce resultados equivalentes. Haremos el análisis para el caso en que FE sea -1. En este caso tendremos que realizar dos rotaciones. Llamaremos P al nodo que muestra el desequilibrio, el que tiene una FE de -2. Llamaremos Q al nodo raíz del subárbol izquierdo de P, y R al nodo raíz del subárbol derecho de Q.

1. Haremos una rotación simple de Q a la izquierda.

2. Después, haremos una rotación simple de P a la derecha. Con más detalle, procederemos del siguiente modo:

   a) Pasamos el subárbol izquierdo del nodo R como subárbol derecho de Q. Esto mantiene el árbol como ABB, ya que todos los valores a la izquierda de R siguen estando a la derecha de Q.

   b) Ahora, el nodo R pasa a tomar la posición del nodo Q, es decir, hacemos que la raíz del subárbol izquierdo de P sea el nodo R en lugar de Q.

3. El árbol Q pasa a ser el subárbol izquierdo del nodo R.

4. RDD pasos 1 a 3.

5. Pasamos el subárbol derecho del nodo R como subárbol izquierdo de P. Esto mantiene el árbol como ABB, ya que todos los valores a la derecha de R siguen estando a la izquierda de P.

6. Ahora, el nodo R pasa a tomar la posición del nodo P, es decir, hacemos que la entrada al árbol sea el nodo R, en lugar del nodo P. Como en los

casos anteriores, previamente, P puede que fuese un árbol completo o un subárbol de otro nodo de menor altura.

7. El árbol P pasa a ser el subárbol derecho del nodo R.

Rotación doble a la izquierda (DI)

Esta rotación se usará cuando el subárbol derecho de un nodo sea 2 unidades más alto que el izquierdo, es decir, cuando su FE sea de 2. Y además, la raíz del subárbol derecho tenga una FE de -1, es decir, que esté cargado a la izquierda. Se trata del caso simétrico del anterior. En este caso también tendremos que realizar dos rotaciones. Llamaremos P al nodo que muestra el desequilibrio, el que tiene una FE de 2. Llamaremos Q al nodo raíz del subárbol derecho de P, y R al nodo raíz del subárbol izquierdo de Q.

1. Haremos una rotación simple de Q a la derecha.
2. Después, haremos una rotación simple de P a la izquierda. Con más detalle, procederemos del siguiente modo:
   a) Pasamos el subárbol derecho del nodo R como subárbol izquierdo de Q. Esto mantiene el árbol como ABB, ya que todos los valores a la derecha de R siguen estando a la izquierda de Q.
   b) Ahora, el nodo R pasa a tomar la posición del nodo Q, es decir, hacemos que la raíz del subárbol derecho de P sea el nodo R en lugar de Q.
3. El árbol Q pasa a ser el subárbol derecho del nodo R.
4. RDI pasos 1 a 3
5. Pasamos el subárbol izquierdo del nodo R como subárbol derecho de P. Esto mantiene el árbol como ABB, ya que todos los valores a la izquierda de R siguen estando a la derecha de P.
6. Ahora, el nodo R pasa a tomar la posición del nodo P, es decir, hacemos que la entrada al árbol sea el nodo R, en lugar del nodo P. Como en los casos anteriores, previamente, P puede que fuese un árbol completo o un subárbol de otro nodo de menor altura.
7. El árbol P pasa a ser el subárbol izquierdo del nodo R.

4.4. Árboles B

4.4.1. Tipo Abstracto Árbol

Las operaciones de la estructura árbol para formar un TAD, son:

1. PADRE(n, A), esta función devuelve el padre del nodo n en el árbol A. Si n es la raíz que no tiene padre, se devuelve un árbol vacío. En este contexto un árbol vacío es un nodo nulo, que se usa como señal de que se ha salido del árbol.

2. HIJO_MAS_IZQ(n, A) devuelve el hijo más a la izquierda del nodo n en el árbo A, y devuelve árbol vacío si n es una hoja y por tanto no tiene hijos.

3. HERMANO_DER(n, A), devuelve el hermano a la derecha del nodo n en el árbol A, el cual se define como el nodo m que tiene el mismo padre p que n, de forma que m está inmediatamente a la derecha de n en el ordenamiento de los hijos de p.

4. ETIQUETA(n, A), devuelve la etiqueta del nodo n en el árbol. Sin embargo no se requiere que haya etiquetas definidas para cada árbol.

5. CREAi(v, $A_1, A_2, \cdots, A_i$) es un miembro de una familia infinita de funciones, una para cada valor de $i = 1, 2, \cdots$, CREAi crea un nuevo nodo r con etiqueta v y le asigna i hijos que son las raices de los árboles $A_1, A_2, \cdots, A_i$, en ese orden desde la izquierda. Se devuelve el árbol con raíz r. Obsérvese que si $i = 0$, entonces r es a la vez una hoja y la raíz.

6. RAIZ(A), devuelve el nodo raíz del árbol A, o el árbol vacío si A es el árbol nulo.

7. ANULA(A), convierte A en un árbol nulo.

Las operaciones que contiene son algunas de las que se enumeran a continuación. Las operaciones de un árbol binario son:

1. Crear un árbol binario vacio.

2. Recorrer un árbol binario.

3. Insertar un nodo nuevo.
4. Borrar un nodo.
5. Buscar un nodo.
6. Copiar la imagen espejo de un árbol.
7. Determinar el número total de nodos.
8. Determinar el número total de nodos hoja.
9. Determinar el número total de nodos no hojas.
10. Encontrar el elemento más pequeño en un nodo.
11. Encontrar el mayor elemento.
12. Encontrar la altura del árbol.
13. Encontrar el padre, el hijo izquierdo, el hijo derecho, los hermanos de un nodo cualquiera.

Capítulo 5

Desarrollo de Aplicaciones

5.1. Ejemplos de Aplicaciones con Estructuras de Datos

Para concatenar las listas; Lista y Lista1, se procede con el siguiente pseudocódigo:

```
1 TIPE p:↑ p ;
2 ↑ p := ↑ Lista ;
3 while p ↑.sig <> NULL do
4   ↑ p := p ↑.sig ;
5 p ↑.sig := ↑ Lista1 ;
```

Si se quiere insertar una ↑ Lista entre dos elementos de una lista ya

construida tantas veces como elementos tenga esa lista:

```
1  TIPE a:↑ a ;
2  TIPE b:↑ b ;
3  ↑ a := ↑ Lista ;
4  ↑ b := ↑ Lista ;
5  ↑ b := b ↑.sig ;
6  ↑ p := ↑ Lista ;
7  while p ↑.sig <> NULL do
8      ↑ a := a ↑.sig ;
9      ↑ b := b ↑.sig ;
10     CREAR-LISTA(Lista) ;
11     a ↑.sig := ↑ Lista ;
12     ↑ temp := ↑ Lista ;
13     while temp ↑.sig <> NULL do
14         ↑ temp := temp ↑.sig ;
15     temp ↑.sig := ↑ b ;
16     ↑ p := p ↑.sig ;
```

5.1.1. Algunas operaciones de las listas enlazadas

Crear las operaciones de la lista y usar sus operaciones para construir otras aplicaciones. Como agregarle un elemento a la lista, borrar un elemento a la lista, buscar un elemento en la lista por valor y por posición, encontrar el primer elemento y el último.

```
 TIPE q:APUNTADOR;
 TIPE p:APUNTADOR;
q := NEW estructuraLista;
q.dato := y;
q.sig := p \uparrow .sig;
p \uparrow .sig := q;
```

5.1.2. Suma de polinomios

Ejemplos de listas:

El siguiente ejemplo es la suma de dos polinomios.

```
 programa adicionPolinomio(entrada, salida)

TIPE nodo:APUNTADOR;
TIPE nodo = REGISTRO
{
     c:REAL;
     SIG:nodo;
}
```

5.1.3. Concatenación de listas enlazadas

Concatenar listas enlazadas:

Concatenar dos listas enlazadas. Sean dos listas L1 y L2 que se colocarán

en una lista L3, se procede de la siguiente manera:

Algorithm 5.1: Concatenar dos listas enlazadas

```
1  TIPE EstructuraLista = REGISTRO
2  { TIPE dato:TIPO_DATO;
3  TIPE ↑ sig:EstructuraLista;
4  }
5  TIPE ↑ m, ↑ L1, ↑ L2, ↑ L3: EstructuraLista;
6  TIPE DPL1:BOOLEANA;
7  EscrituraTeclado(DPL1);
8  if DPL1 == 1 then
9  |  ↑ m := ↑ L1;
10 |  while m ↑.sig <> NULL do
11 |  |  ↑ m := m ↑,sig;
12 |  m ↑.sig := ↑ L2;
13 |  ↑ m := ↑ L1;
14 else
15 |  ↑ m := ↑ L2;
16 |  while m ↑.sig <> NULL do
17 |  |  ↑ m := m ↑,sig;
18 |  m ↑.sig := ↑ L1 ;
19 |  ↑ m := ↑ L2;
20 ↑ L3 := ↑ m;
21 return(↑ L3);
```

Concatenar tres listas enlazadas. Sean L1, L2 y L3 que se colocarán con-

catenadas en la lista L4, se procede de la siguiente manera:

Algorithm 5.2: Concatenar tres listas enlazadas

```
1  TIPE EstructuraLista = REGISTRO
2  { TIPE dato:TIPO_DATO;
3  TIPE ↑ sig:EstructuraLista;
4  }
5  TIPE ↑ m, ↑ L1, ↑ L2, ↑ L3, ↑ L4: EstructuraLista;
6  TIPE DPL1, DPL2, DPL3, DSL3:BOOLEANA;
7  EscrituraTeclado(Valor DPL1 o DPL2 o DPL3 o DPL1 y DSL3 o
   DPL2 y DSL3);
       if DPL1 == 1 && DSL3 == 0 then
         Concatenar(↑L1, ↑L2, ↑L3);
       else if DPL1 == 1 && DSL3 == 1  then
         Concatenar(↑L1, ↑L3, ↑L2);
       else if DPL2 == 1 && DSL3 == 0 then
         Concatenar(↑L2, ↑L1, ↑L3);
       else if DPL2 == 1 && DSL3 == 1  then
8:       Concatenar(↑L2, ↑L3, ↑L1);
       else if DPL3 == 1 then
         Concatenar(↑L3, ↑L1, ↑L2);
       else
         Concatenar(↑L3, ↑L2, ↑L1);
       end if
```

Copiar una lista L1 previamente creada en una lista L2 la cual se va

creando, se procede de la siguiente manera:

Algorithm 5.3: Copiar una lista previamente creada en otra no creada

```
1  TIPE EstructuraLista = REGISTRO
2  { TIPE dato:TIPO_DATO;
3  TIPE ↑ sig:EstructuraLista;
4  }
5  TIPE ↑ p, ↑ m, ↑ L1, ↑ L2: EstructuraLista;
6  //La lista L1 se crea previamente
7  ↑ L2 := NULL;
8  ↑ p := ↑ L1;
9  while ↑ p <> NULL do
10     NuevoElemento := NEW EstructuraLista;
11     if ↑ L2 == NULL then
12         CopiarInformacion(NuevoElemento, ↑ L2);
13         ↑ L2 := NuevoElemento;
14         ↑ m := ↑ L2;
15         NuevoElemento.sig := NULL;
16     else
17         CopiarInformacion(NuevoElemento, ↑ L2);
18         m↑.sig := NuevoElemento;
19         NuevoElemento.sig := NULL;
20         ↑ m := m↑.sig;
21     ↑ p := p↑.sig;
```

Invertir una lista L previamente creada:

Algorithm 5.4: Invertir una lista

```
1  TIPE EstructuraLista = REGISTRO
2  { TIPE dato:TIPO_DATO;
3  TIPE ↑ sig:EstructuraLista;
4  }
5  TIPE ↑ p, ↑ q, ↑ r, ↑ L, ↑ LI:EstructuraLista;
6  ↑ p := ↑ L;
7  ↑ q := ↑ L;
8  while r↑.sig <> NULL do
9      ↑ q := p↑.sig;
10     ↑ r := q↑.sig;
11     q↑.sig := ↑ p;
12 r↑.sig := ↑ q;
13 L↑.sig := NULL;
14 ↑ LI := ↑ r;
```

Para insertar una lista creada previamente L2 en otra lista L1, se procede de la siguiente manera:

Algorithm 5.5: Insertar una lista dentro de otra

```
1  TIPE EstructuraLista = REGISTRO
2  {
3  TIPE dato:TIPO_DATO;
4  TIPE ↑ sig:EstructuraLista;
5  }
6  TIPE ↑ m, ↑ q, ↑ p, ↑ L1, ↑ L2:EstructuraLista;
7  ↑ m := ↑ L1;
8  ↑ p := ↑ L2;
9  while m↑.sig <> NULL do
10     ↑ m := m↑.sig;
11     ↑ q := m↑.sig;
12     if m↑.Informacion == x then
13         while q↑.sig <> NULL do
14             ↑ p := p↑.sig;
15         m↑.sig := ↑ L2;
16         p↑.sig := ↑ q;
```

Construir polinomios para aplicarles la suma algebraíca:

Algorithm 5.6: Operación de polinomios

```
1  TIPE EstructuraPolinomio = REGISTRO
2  {
3  TIPE exponente:ENTERO;
4  TIPE coeficiente:ENTERO;
5  TIPE ↑ sig:EstructuraPolinomio;
6  }
7  TIPE ↑ termino, ↑ p1, ↑ p2, ↑ m, ↑ n:EstructuraPolinomio;
8  //construir los polinomios
9  ↑ p1 := NULL;
10 ↑ p2 := NULL;
11 ConstruirPolinomio(p1);
12 ConstruirPolinomio(p2);
13 ↑ m := ↑ p1;
14 ↑ n := ↑ p2;
15 OrdenarPolinomio(p1);
16 ContarTerminos(p1);
17 OrdenarPolinomio(p2);
18 ContarTerminos(p2);
19 SumarTerminos(p1, p2);
20 funcion OrdenarPolinomio():TIPO_DATO ;
21 var ↑ p1, ↑ p2, ↑ p3:EstructuraPolinomio;
22 begin
23 if Terminos(p1) > Terminos(p2) then
24     while terminos <> 0 do
25         BuscarMayor(p2);
26         ContadorExponente(p2);
27         BuscarMayor(p1);
28         ContadorExponente(p1);
29         if m↑.exponente == n↑.exponente then
30             SumarTerminos(p1, p2);
31         else
32             PonerMismo();
33 else
34     Hacerlo(p2);
35 ComparарTermino(p1, p2);
36 end
```

La construcción de las funciones se procede de la siguiente manera:

Algorithm 5.7: Construcción de funciones

```
1 funcion Intercambiar():TIPO_DATO ;
2 var ↑ p_a, ↑ p_1, ↑ p_2:EstructuraLista;
3 begin
4 ↑ p_a := ↑ p_1 ;
5 ↑ p_1 := ↑ p_2 ;
6 ↑ p_2 := ↑ p_a ;
7 return ↑ p_1, ↑ p_2 ;
8 end
```

Si la función no entrega nada se escribe de la siguiente manera:

Algorithm 5.8: Construcción de funciones que no entregan nada

```
1 funcion Intercambiar():TIPO_DATO;
2 var ↑ p_a, ↑ p_1, ↑ p_2:pila;
3 begin
4 ↑ p_a := ↑ p_1;
5 ↑ p_1 := ↑ p_2;
6 ↑ p_2 := ↑ p_a;
7 return void;
8 end
```

La función de contar términos en un polinomio:

Algorithm 5.9: Contar términos

```
1  funcion ContarTerminos():ENTERO;
2  var ↑ a, ↑ p:EstructuraPolinomio;
3  begin
4  0 ← Contador
5  ↑ a := ↑ p;
6  while a↑.sig <> NULL do
7  |   ↑ a := a↑.sig;
8  |   Contador := Contador + 1;
9  return Contador;
10 end
```

Antes de entrar a la construcción de las funciones de un polinomio, se harán otras funciones, como buscar en una lista de números el número de

mayor valor:

Algorithm 5.10: Buscar el número de mayor valor

```
1  funcion BuscarMayorValor():ENTERO;
2  var ↑ L, ↑ p:EstructuraLista;
3  var b:ENTERO;
4  begin
5  ↑ L ←↑ p
6  b := p↑.Informacion;
7  while p↑.sig <> NULL do
8      if b < p↑.Informacion then
9          b := p↑.Informacion;
10     ↑ p := p↑.sig;
11 return b;
12 end
```

Buscar el mayor número y además hacer el intercambio:

Algorithm 5.11: Buscar el número de mayor valor

```
1  funcion OrdenarMayorValor():ENTERO;
2  var ↑ L, ↑ p:EstructuraLista;
3  var b:ENTERO;
4  begin
5  ↑ L ←↑ p
6  b := p↑.Informacion;
7  while p↑.sig <> NULL do
8      if b < p↑.Informacion then
9          b := p↑.Informacion;
10         Intercambiar();
11     ↑ p := p↑.sig;
12 return void(Lista ordenada);
13 end
```

Otra función usada para hacer operaciones con los polinomios:

Algorithm 5.12: Intercambiar términos en un polinomio

```
1  funcion IntercambiarTermino():EstructuraPolinomio;
2  var ↑ a, ↑ p, ↑ temp:EstructuraPolinomio;
3  begin
4  ↑ p1 ←↑ p
5  ↑ a := ↑ p;
6  ↑ p := p ↑.sig;
7  temp↑.Informacion := a ↑.Informacion;
8  a ↑.Informacion := p ↑.Informacion;
9  p ↑.Informacion := temp↑.Informacion;
10 return void;
11 end
```

Para buscar el término en un polinomio:

Algorithm 5.13: Buscar el término de mayor grado

```
1  funcion BuscarMayor():EstructuraPolinomio;
2  var ↑ a, ↑ b, ↑ p:EstructuraPolinomio;
3  begin
4  ↑ p ←↑ a
5  ↑ b := a↑.sig;
6  if a↑.exponente > b↑.exponente then
7  |  temp↑.Exponente := a↑.Exponente;
8  else
9  |  temp↑.Exponente := b↑.Exponente;
10 ↑ b := b↑.sig;
11 while b↑.sig <> NULL do
12 |  if b↑.exponente > temp↑.exponente then
13 |  |  temp↑.exponente := b↑.Exponente;
14 |  ↑ b := b↑.sig;
15 return b↑.exponente;
16 end
```

Si se buscan los términos por grado del término:

Algorithm 5.14: Buscar la existencia de términos

```
1  funcion BuscarTermino():EstructuraPolinomio;
2  var ↑ a, ↑ p:EstructuraPolinomio;
3  begin
4  0 ←↑ a
5  if a↑.exponente == p↑.exponente then
6  |  SumarTerminos();
7  else
8  |  if p↑.exponente == 1 then
9  |  |  poner(p↑.exponente);
10 |  else
11 |  |  poner(a↑.exponente);
12 return ↑ L2;
13 end
```

5.1.4. Concatenar listas enlazadas hetergéneas

Enlazar dos listas hetergéneas.

```
1  programa Nombre de programa ;
2  n ← 8, p_a ← {2 . . . n.}, rango ← 1 . . . n;
3  var ↑ p_a, ↑ p_1:pila ;
4  procedimiento Nombre del procedimiento ;
5  funcion Nombre de la función ;
6  funcion Función principal ;
```

5.2. Problemas de pilas

Agregar elmentos a una pila, para lo que se hace de una pila auxiliar $\uparrow p_a$ a una pila $\uparrow p_1$, por medio del siguiente algoritmo:

Algorithm 5.15: Dos pilas

Data: $\uparrow p_a$, $\uparrow p_1$.
Result: $\uparrow p_1$

```
1 ↑ p1 ← {0, 1, ... n.};
2 while ↑ pa <> NULL do
3 |  PUSH(cima(↑ pa), ↑ p1) ;
4 return pila ;
```

Localizar elementos en un arreglo bidimensional de listas enlazadas. Para lo que existe la forma de que cada elemento del arreglo bidimensional tenga cuatro apuntadores:

Algorithm 5.16: Declaración arreglo bidimensional

```
1 TIPE EstLis = REGISTRO
2 {
3 TIPE dato:TIPO_DATO ;
4 TIPE ↑ der:EstLis ;
5 TIPE ↑ izq:EstLis ;
6 TIPE ↑ arr:EstLis ;
7 TIPE ↑ abj:EstLis ;
8 }
9 ↑ LB:EstLis ;
```

Se crean los elementos del arreglo bidimensional.

Algorithm 5.17: Crear los elementos del arreglo bidimensional

```
1 funcion CrearElemento:EstLis ;
2 var ↑ elem:EstLis ;
3 begin
4 elem := NEW EstLis ;
5 return elem ;
6 end
```

5.3. Ordenamiento con listas enlazadas

Se tiene una secuencia $< 3, 5, 7, 9, 11, 1, 2, 10, 4, 8, 6 >$, meterla en una lista para ordenarla de menor a mayor. Esto se hace con el siguiente pseudocódigo:

```
1  ↑ al := ↑ L ;
2  ↑ ap := ↑ L ;
3  ↑ am := ↑ L ;
4  ↑ ap := ap ↑.sig ;
5  while al ↑.sig <> NULL do
6  |   while ap ↑.sig <> NULL do
7  |   |   if al ↑.dato < ap ↑.dato then
8  |   |   |   ↑ ap := ap ↑.sig ;
9  |   |   else
10 |   |   |   r poner_antes(↑ ap) ;
11 |   ↑ al := al ↑.sig ;
```

La función de poner antes se hace de la siguiente forma:

```
1  poner_antes(↑ ap) ;
2  if ↑ al == ↑ L then
3  |   ↑ temp := ↑ ap ;
4  |   if ap ↑.sig == NULL then
5  |   |   return(↑ ap) ;
6  |   else
7  |   |   ↑ ap := ap ↑.sig ;
8  |   while ↑ am <> ↑ temp do
9  |   |   ↑ am := am ↑.sig ;
10 |   am ↑.sig := temp ↑.sig ;
11 |   temp ↑.sig := ↑ L ;
12 |   ↑ L := ↑ temp ;
13 |   ↑ al := ↑ L ;
14 else
15 |   poner_despues() ;
```

Para crear la función poner_despues(), se hace con el siguiente pseudocódi-

go:

```
poner_despues() ;
```

Si se usan pilas y colas, para ordenar la secuencia $< 3, 5, 7, 9, 11, 1, 2, 10, 4, 8, 6 >$, para meter la secuencia a las pilas se hace por medio del siguiente pseudocódigo:

```
CrearPila(pila) ;
LeerElemento(x_n) ;
MeterElemento(x_n,pila) ;
LeerSiguienteElemento(x_n) ;
while cima < x_n do
    MeterElemento(x_n,pila) ;
    RecorrerCima(pila) ;
    LeerSiguienteElemento(x_n) ;
```

Se construye la función LeerSiguienteElemento() de la manera que se muestra a continuación:

```
LeerSiguienteElemento(x_n) ;
x_n := x_{n+1} ;
return(x_n) ;
```

De la cual se deben construir las funciones CrearPila(), MeterElemento() y hacer la lectura de la secuencia. La lectura de la secuencia se hace llamando a la secuencia elemento por elemento:

```
a := x_n ;
b := x_{n+1} ;
if a < b then
    empilar(a) ;
else
    crearPila(b) ;
```

5.4. Implementación de aplicaciones

5.4.1. Definición y análisis del problema

Si tenemos una sucesión de números y requerimos ordenarla de manera ascendente y descendente, utilizando las pilas que sean necesarias:

$$< 10, 7, 2, 3, 4, 11, 1, 8, 6, 9, 5 >$$

5.4.2. Diseño y Programación

5.4.3. Búsqueda

Existen varios métodos de búsqueda y muchos algoritmos para cada uno de estos métodos, dentro de un conjunto de datos se busca un elemento determinado. El conjunto de datos $\{N\}$ se almacena en un arreglo, lo cual se representa como:

$$VAR\ A : ARRAY[1 \cdots N]\ OF\ ELEMENTOS$$

El elemento se considera como una estructura tipo registro con un campo que actúa como una clave, así que la tarea consiste en encontrar un elemento cuyo campo clave dentro del arreglo sea igual al argumento x. El índice i resultante, satisface la condición $A[i].clave = x$, entonces permite el acceso a otros campos del elemento localizado. Se puede decir que existe una búsqueda pura que consiste en encontrar el elemento dentro del arreglo directamente.

5.4.4. Conversión de decimal a binario

Si queremos representar un número decimal, lo hacemos de manera posicional que se representa de la siguiente manera:

$$b_k b_{k-1} \cdots b_2 b_1 b_0$$

Donde b_i representa un digito y el índice la posición del dígito. Los sistemas numéricos tienen una base que se representa por medio de un conjunto de digitos que se usan para representar los números y la cardinalidad de la base es la cantidad de digitos. Así existen sistemas de numeración binarios que tienen como conjunto base el siguiente:

$$\{0, 1\}$$

Y su cardinalidad es 2.

El siguiente sistema numérico es el de base diez y su conjunto base es el siguiente:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$$

Y su cardinalidad es 10.

Otro sistema numérico es el hexadecimal que tiene como conjunto base el siguiente:

$$\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

Y su cardinalidad es 16.

Así que los elementos de la base son los dígitos que representan los siguientes dígitos:

$$b_k b_{k-1} \cdots b_2 b_1 b_0$$

El valor del dígito es de la siguiente manera:

$$b_i = \{digitos\}_{base}$$

Estos sistemas numéricos son los que se usan mucho, y uno de los requerimientos es que se puede convertir de un sistema a otro, asi que generalmente el sistema numérico en el que se trabaja es el de base diez por lo que se requiere una conversion de decimal a binario.

Capítulo 6

Apéndices

6.1. Programas

6.1.1. Colas de prioridad

```
 #include <iostream>
#include <stdlib.h>
using namespace std;

/*                Estructura de los nodos de la cola
--------------------------------------------------------------------------*/
struct nodo
{
    char dato;
    int priori;        // prioridad del nodo
    struct nodo *sgte;
};

/*                      Estructura de la cola
--------------------------------------------------------------------------*/
struct cola
{
    nodo *delante;
    nodo *atras  ;
};
```

```
/*                          Crear Nodo
-------------------------------------------------------------------------*/
struct nodo *crearNodo( char x, int pr)
{
    struct nodo *nuevoNodo = new(struct nodo);
    nuevoNodo->dato = x;
    nuevoNodo->priori = pr;
    return nuevoNodo;
};

/*                        Encolar cacarter con prioridad
-------------------------------------------------------------------------*/
void encolar( struct cola &q, char valor, int priori )
{

     struct nodo *aux = crearNodo(valor, priori);
     aux->sgte = NULL;

     if( q.delante == NULL)
         q.delante = aux;   // encola el primero elemento
     else
         (q.atras)->sgte = aux;

     q.atras = aux;        // puntero que siempre apunta al ultimo elemento
}

/*                        Mostrar Cola
-------------------------------------------------------------------------*/
void muestraCola( struct cola q )
{
     struct nodo *aux;

     aux = q.delante;

     cout << " Caracter  Prioridad " << endl;
     cout << " ------------------- " << endl;

     while( aux != NULL )
```

```
     {
            cout<<"    "<< aux->dato << "     |   " << aux->priori << endl;
            aux = aux->sgte;
     }
}

/*         Ordenar  por prioridad( criterio de O. por Burbuja)
-------------------------------------------------------------------------*/
void ordenarPrioridad( struct cola &q )
{
     struct nodo *aux1, *aux2;
     int p_aux;
     char c_aux;

     aux1 = q.delante;

     while( aux1->sgte != NULL)
     {
            aux2 = aux1->sgte;

            while( aux2 != NULL)
            {
                   if( aux1->priori > aux2->priori )
                   {
                       p_aux = aux1->priori;
                       c_aux = aux1->dato;

                       aux1->priori = aux2->priori;
                       aux1->dato   = aux2->dato;

                       aux2->priori = p_aux;
                       aux2->dato   = c_aux;
                   }

                   aux2 = aux2->sgte;
            }

            aux1 = aux1->sgte;
```

```
     }
}
/*                Inserta cacacteres en una cola
-------------------------------------------------------------------------*/
void insertar( struct cola &q, char c, int pr )
{
     /* Encolando caracteres */
     encolar( q, c, pr );

     /* Ordenando por prioridad */
     ordenarPrioridad( q );
}

/*                        Menu de opciones
-------------------------------------------------------------------------*/
void menu()
{
    cout<<"\n\t COLAS CON PRIORIDAD EN C++ \n\n";
    cout<<" 1. ENCOLAR                          "<<endl;
    cout<<" 2. MOSTRAR                          "<<endl;
    cout<<" 3. SALIR                            "<<endl;

    cout<<"\n INGRESE OPCION: ";
}

/*                        Funcion Principal
-------------------------------------------------------------------------*/
int main()
{
    struct cola q;

    q.delante = NULL;
    q.atras   = NULL;

    char c ;     // caracter a encolar
    int pr;      // prioridad del caracter
    int op;      // opcion del menu
    int x ;      // numero que devuelve la funcon pop
```

```
do
{
    menu();  cin>> op;

    switch(op)
    {
        case 1:

             cout<< "\n Ingrese caracter: ";
             cin>> c;

             cout<< "\n Ingrese prioridad: ";
             cin>> pr;

             insertar( q, c, pr );

             cout<<"\n\n\t\tCaracter '" << c << "' encolado...\n\n";
        break;

        case 2:

             cout << "\n\n MOSTRANDO COLA DE PRIORIDAD\n\n";

             if(q.delante!=NULL)
                 muestraCola( q );
             else
                cout<<"\n\n\tCola vacia...!"<<endl;
        break;

        default:
                cout<<"\n\tOpcion incorecta..!"<<endl;
                system("pause");
                exit(0);
     }

    cout<<endl<<endl;
    system("pause");  system("cls");
```

```
    }while(op!=3);

    return 0;
}
```

6.1.2. Colas

```
  #include <iostream>
#include <stdlib.h>
using namespace std;


/*                Estructura de los nodos de la cola
--------------------------------------------------------------------------*/
struct nodo
{
    char dato;
    struct nodo *sgte;
};

/*                      Estructura de la cola
--------------------------------------------------------------------------*/
struct cola
{
    nodo *delante;
    nodo *atras  ;
};

/*                          Crear Nodo
--------------------------------------------------------------------------*/
struct nodo *crearNodo( char x)
{
    struct nodo *nuevoNodo = new(struct nodo);
    nuevoNodo->dato = x;
    return nuevoNodo;
};
```

```
/*                        Encolar elemento
-------------------------------------------------------------------------*/
void encolar( struct cola &q, char x, int pos )
{
     struct nodo *aux = crearNodo(x);

     if( pos==1 )
     {
          if( q.delante==NULL)
          {
               aux->sgte = q.delante;
               q.delante = aux;
               q.atras   = aux;
          }
          else
          {
               aux->sgte = q.delante;
               q.delante = aux;
          }
     }
     else
     {
          if( q.atras==NULL )
          {
               aux->sgte = q.atras;
               q.delante = aux;
               q.atras   = aux;
          }
          else
          {
               aux->sgte = (q.atras)->sgte;
               (q.atras)->sgte = aux;
          }
     }
}

/*                        Desencolar elemento
-------------------------------------------------------------------------*/
```

```
char desencolar( struct cola &q, int pos )
{
     char __c ;
     struct nodo *aux = q.delante;

     if( pos==1 )
     {
          __c = (q.delante)->dato;
          q.delante = aux->sgte;
          delete(aux);
     }
     else
     {
          __c = (q.atras)->dato;

          while( aux->sgte!=q.atras )
               aux = aux->sgte;

          aux->sgte = (q.atras)->sgte;
          delete(q.atras);
          q.atras = aux;
     }

     return __c;
}

/*                        Mostrar Cola
------------------------------------------------------------------------*/
void muestraCola( struct cola q )
{
     struct nodo *aux;

     aux = q.delante;

     while( aux != NULL )
     {
            cout<<"   "<< aux->dato ;
            aux = aux->sgte;
```

```
     }
}

/*                          Menu de opciones
------------------------------------------------------------------------*/
void menu()
{
    cout<<"\n\t IMPLEMENTACION DE COLAS DOBLES EN C++\n\n";
    cout<<" 1. INSERTAR                                  "<<endl;
    cout<<" 2. ELIMINAR                                  "<<endl;
    cout<<" 3. MOSTRAR COLA                              "<<endl;
    cout<<" 4. SALIR                                     "<<endl;

    cout<<"\n INGRESE OPCION: ";
}

/*                          Funcion Principal
------------------------------------------------------------------------*/
int main()
{
    struct cola q;

    q.delante = NULL;
    q.atras   = NULL;

    char c;     // caracter a encolar
    char x ;    // caracter que devuelve la funcion pop (desencolar)
    int op;     // opcion del menu
    int pos;    // posicion de isertar o eliminar (inicio o fin)

    do
    {
        menu();  cin>> op;

        switch(op)
        {
            case 1:
                 cout<< "\n Ingrese caracter: ";
```

```
            cin>> c;

            cout<<"\n\t[1] Inserta al inicio " <<endl;
            cout<<"\t[2] Inserta al final  " <<endl;
            cout<<"\n\t    Opcion : ";
            cin>> pos;

            encolar( q, c, pos );

            cout<<"\n\n\t\tNumero '" << c << "' encolado...\n\n";

        break;


        case 2:
            cout<<"\n\t[1] Elimina al inicio " <<endl;
            cout<<"\t[2] Elimina al final  " <<endl;
            cout<<"\n\t    Opcion : ";
            cin>> pos;

            x = desencolar( q, pos );

            cout<<"\n\n\t\tNumero '"<< x <<"' desencolado...\n\n";

        break;


        case 3:
            cout << "\n\n MOSTRANDO COLA\n\n";

            if(q.delante!=NULL)
                muestraCola( q );
            else
                cout<<"\n\n\tCola vacia...!" << endl;

        break;
    }
```

```
        cout<<endl<<endl;
        system("pause");  system("cls");

    }while(op!=4);

    return 0;
}
```

6.1.3. Árboles

```
 #include <iostream>
#include <stdlib.h>

using namespace std;

/*---------  Estructura del arbol  -------*/
typedef struct nodo{
     int nro;
     struct nodo *izq, *der;
}*ABB;

int numNodos = 0; // nummero de nodos del arbol ABB
int numK = 0, k;     //  nodos menores que un numero k ingresado


/* ---------- Estructura de la cola ---------*/
struct nodoCola{
    ABB ptr;
    struct nodoCola *sgte;
};
struct cola{
    struct nodoCola *delante;
    struct nodoCola *atras;
};
```

```
void inicializaCola(struct cola &q)
{
    q.delante = NULL;
    q.atras = NULL;
}

void encola(struct cola &q, ABB n)
{
     struct nodoCola *p;
     p = new(struct nodoCola);
     p->ptr = n;
     p->sgte = NULL;
     if(q.delante==NULL)
         q.delante = p;
     else
         (q.atras)->sgte = p;
     q.atras = p;
}

ABB desencola(struct cola &q)
{
    struct nodoCola *p;
    p = q.delante;
    ABB n = p->ptr;
    q.delante = (q.delante)->sgte;
    delete(p);
    return n;
}

ABB crearNodo(int x)
{
     ABB nuevoNodo = new(struct nodo);
     nuevoNodo->nro = x;
     nuevoNodo->izq = NULL;
     nuevoNodo->der = NULL;

     return nuevoNodo;
}
```

```
void insertar(ABB &arbol, int x)
{
     if(arbol==NULL)
     {
           arbol = crearNodo(x);
           cout<<"\n\t  Insertado..!"<<endl<<endl;
     }
     else if(x < arbol->nro)
          insertar(arbol->izq, x);
     else if(x > arbol->nro)
          insertar(arbol->der, x);
}

void preOrden(ABB arbol)
{
     if(arbol!=NULL)
     {
          cout << arbol->nro <<" ";
          preOrden(arbol->izq);
          preOrden(arbol->der);
     }
}

void enOrden(ABB arbol)
{
     if(arbol!=NULL)
     {
          enOrden(arbol->izq);
          cout << arbol->nro << " ";
          enOrden(arbol->der);
     }
}

void postOrden(ABB arbol)
{
     if(arbol!=NULL)
     {
          enOrden(arbol->izq);
```

```
          enOrden(arbol->der);
          cout << arbol->nro << " ";
     }
}

void verArbol(ABB arbol, int n)
{
     if(arbol==NULL)
          return;
     verArbol(arbol->der, n+1);

     for(int i=0; i<n; i++)
         cout<<"   ";

     numNodos++;
     cout<< arbol->nro <<endl;

     verArbol(arbol->izq, n+1);
}

bool busquedaRec(ABB arbol, int dato)
{
     int r=0;   // 0 indica que lo encontre

     if(arbol==NULL)
        return r;

     if(dato<arbol->nro)
         r = busquedaRec(arbol->izq, dato);

     else if(dato> arbol->nro)
         r = busquedaRec(arbol->der, dato);

     else
        r = 1;   // son iguales, lo encontre

     return r;
}
```

```
ABB unirABB(ABB izq, ABB der)
{
    if(izq==NULL) return der;
    if(der==NULL) return izq;

    ABB centro = unirABB(izq->der, der->izq);
    izq->der = centro;
    der->izq = izq;
    return der;
}

void elimina(ABB &arbol, int x)
{
     if(arbol==NULL) return;

     if(x<arbol->nro)
         elimina(arbol->izq, x);
     else if(x>arbol->nro)
         elimina(arbol->der, x);

     else
     {
         ABB p = arbol;
         arbol = unirABB(arbol->izq, arbol->der);
         delete p;
     }
}

int alturaAB(ABB arbol)
{
    int AltIzq, AltDer;

    if(arbol==NULL)
        return -1;
    else
    {
        AltIzq = alturaAB(arbol->izq);
```

```
        AltDer = alturaAB(arbol->der);

        if(AltIzq>AltDer)
            return AltIzq+1;
        else
            return AltDer+1;
    }
}

void recorrerxNivel(ABB arbol)
{
     struct cola q;
     inicializaCola(q);
     cout << "\t";

     if(arbol!=NULL)
     {
         encola(q, arbol);

         while(q.delante!=NULL)
         {
             arbol = desencola(q);
             cout << arbol->nro << ' ';

             if(arbol->izq!=NULL)
                 encola(q, arbol->izq);
             if(arbol->der!=NULL)
                 encola(q, arbol->der);
         }
     }
}

ABB arbolEspejo(ABB arbol)
{
    ABB temp;

    if(arbol!=NULL)
    {
```

```
          temp = arbol->izq;
          arbol->izq = arbolEspejo(arbol->der);
          arbol->der = arbolEspejo(temp);
    }
    return arbol;
}

void nodosMenoresQueK(ABB arbol, int n)
{
     if(arbol==NULL)
          return;
     nodosMenoresQueK(arbol->der, n+1);
     if(arbol->nro<k)
         numK++;
     nodosMenoresQueK(arbol->izq, n+1);

}

int contarHojas(ABB arbol)
{
        if (arbol==NULL)
        {
                return 0;
        }
        if ((arbol->der ==NULL)&&(arbol->izq ==NULL))
        {
                return 1;
        }
        else
        {
                return contarHojas(arbol->izq) + contarHojas(arbol->der);
        }
}
void menu()
{
     //system("cls");
     cout << "\n\t\t  ..[ ARBOL BINARIO DE BUSQUEDA ]..  \n\n";
     cout << "\t [1]  Insertar elemento                  \n";
```

```
     cout << "\t [2]  Mostrar arbol                       \n";
     cout << "\t [3]  Recorridos de profundiad            \n";
     cout << "\t [4]  Buscar elemento                     \n";
     cout << "\t [5]  Eliminar elemento                   \n";
     cout << "\t [6]  Recorrido por nieveles (Amplitud)   \n";
     cout << "\t [7]  Altura del arbol                    \n";
     cout << "\t [8]  Construir arbol reflejo             \n";
     cout << "\t [9]  Contar nodos                        \n";
     cout << "\t [x]  Contar hojas                        \n";
     cout << "\t [11]  Nodos menores de 'k'                \n";
     cout << "\t [12] SALIR                               \n";

     cout << "\n\t Ingrese opcion : ";
}

void menu2()
{
     //system("cls");   // para limpiar pantalla
     cout << endl;
     cout << "\t [1] En Orden     \n";
     cout << "\t [2] Pre Orden    \n";
     cout << "\t [3] Post Orden   \n";
     cout << "\n\t     Opcion :  ";
}

int main()
{
    ABB arbol = NULL;
    int x;
    int op, op2;


    //system("color f9");   // poner color a la consola
    do
    {
          menu();  cin>> op;
          cout << endl;
```

```
switch(op)
{
     case 1:
          cout << " Ingrese valor : ";  cin>> x;
          insertar( arbol, x);
          break;

     case 2:
          verArbol(arbol, 0);
          break;

     case 3:
          menu2();  cin>> op2;

          if(arbol!=NULL)
          {
               switch(op2)
               {
                   case 1:
                       enOrden(arbol); break;
                   case 2:
                       preOrden(arbol); break;
                   case 3:
                       postOrden(arbol); break;
               }
          }
          else
              cout << "\n\t Arbol vacio..!";
          break;

     case 4:
          bool band;

          cout<<" Valor a buscar: "; cin>> x;

          band = busquedaRec(arbol,x);

          if(band==1)
```

```
            cout << "\n\tEncontrado...";
         else
            cout << "\n\tNo encontrado...";
         break;

      case 5:
         cout<<" Valor a eliminar: "; cin>> x;
         elimina(arbol, x);
         cout << "\n\tEliminado..!";
         break;

      case 6:
         cout<<"\n\n Mostrando recorrido por amplitud\n\n";
         recorrerxNivel(arbol);
         break;

      case 7:
         int h;
         h = alturaAB(arbol);
         cout << " La altura es : "<< h << endl;
         break;

      case 8:
         ABB espejo;
         espejo = NULL;

         cout << "\n\n Arbol incial \n\n";

         verArbol(arbol, 0);

         cout << "\n\n Arbol espejo \n\n";

         espejo = arbolEspejo(arbol);

         verArbol(espejo, 0);
         break;

      case 9:
```

```
                    verArbol(arbol, 0);
                    cout << "\n\n El numero de nodos es : ";
                    cout << numNodos;
                    break;

               case 11:
                    cout << " Ingrese k: "; cin>> k;
                    nodosMenoresQueK(arbol, 0);
                    cout <<" Son  "<< numK << "  numeros";
                    break;


               case 12:
                    exit(0);
          }

          cout<<"\n\n\n";
          //system("pause");  // hacer pausa y presionar una tecla para
                              // continuar
    }while(op!=11);

}
```

6.1.4. Árbol binario de búsqueda

```
 #include <iostream>
#include <stdlib.h>
using namespace std;

struct nodo{
     int nro;
     struct nodo *izq, *der;
};

typedef struct nodo *ABB;
/* es un puntero de tipo nodo que hemos llamado ABB, que ulitizaremos
   para mayor facilidad de creacion de variables */
```

```
ABB crearNodo(int x)
{
     ABB nuevoNodo = new(struct nodo);
     nuevoNodo->nro = x;
     nuevoNodo->izq = NULL;
     nuevoNodo->der = NULL;

     return nuevoNodo;
}
void insertar(ABB &arbol, int x)
{
     if(arbol==NULL)
     {
           arbol = crearNodo(x);
     }
     else if(x < arbol->nro)
          insertar(arbol->izq, x);
     else if(x > arbol->nro)
          insertar(arbol->der, x);
}

void preOrden(ABB arbol)
{
     if(arbol!=NULL)
     {
          cout << arbol->nro <<" ";
          preOrden(arbol->izq);
          preOrden(arbol->der);
     }
}

void enOrden(ABB arbol)
{
     if(arbol!=NULL)
     {
          enOrden(arbol->izq);
          cout << arbol->nro << " ";
```

```
          enOrden(arbol->der);
     }
}

void postOrden(ABB arbol)
{
     if(arbol!=NULL)
     {
          postOrden(arbol->izq);
          postOrden(arbol->der);
          cout << arbol->nro << " ";
     }
}

void verArbol(ABB arbol, int n)
{
     if(arbol==NULL)
          return;
     verArbol(arbol->der, n+1);

     for(int i=0; i<n; i++)
         cout<<"   ";

     cout<< arbol->nro <<endl;

     verArbol(arbol->izq, n+1);
}

int main()
{
    ABB arbol = NULL;   // creado Arbol

    int n;  // numero de nodos del arbol
    int x; // elemento a insertar en cada nodo

    cout << "\n\t\t  ..[ ARBOL BINARIO DE BUSQUEDA ]..  \n\n";

    cout << " Numero de nodos del arbol:  ";
```

```
    cin >> n;
    cout << endl;

    for(int i=0; i<n; i++)
    {
        cout << " Numero del nodo " << i+1 << ": ";
        cin >> x;
        insertar( arbol, x);
    }

    cout << "\n Mostrando ABB \n\n";
    verArbol( arbol, 0);

    cout << "\n Recorridos del ABB";

    cout << "\n\n En orden   :  ";   enOrden(arbol);
    cout << "\n\n Pre Orden  :  ";   preOrden(arbol);
    cout << "\n\n Post Orden :  ";   postOrden(arbol);

    cout << endl << endl;

    system("pause");
    return 0;
}
```

6.1.5. Gráfos

```
 #include<iostream.h>
#include<conio.h>

struct nodo{
            char nombre;//nombre del vertice o nodo
            struct nodo *sgte;
            struct arista *ady;//puntero hacia la primera arista
                               //del nodo
            };
struct arista{
              struct nodo *destino;//puntero al nodo de llegada
```

```
              struct arista *sgte;
              };
typedef struct nodo *Tnodo;//  Tipo Nodo
typedef struct arista *Tarista; //Tipo Arista

Tnodo p;//puntero cabeza

void menu();
void insertar_nodo();
void agrega_arista(Tnodo &, Tnodo &, Tarista &);
void insertar_arista();
void vaciar_aristas(Tnodo &);
void eliminar_nodo();
void eliminar_arista();
void mostrar_grafo();
void mostrar_aristas();


/*                        Funcion Principal
------------------------------------------------------------------------*/
int main(void)
{   p=NULL;
    int op;     // opcion del menu


    system("color 0b");

    do
    {
        menu();
        cin>>op;

        switch(op)
        {
            case 1:
                    insertar_nodo();
                    break;
            case 2: insertar_arista();
```

```
                break;
            case 3: eliminar_nodo();
                    break;
            case 4: eliminar_arista();
                    break;
            case 5: mostrar_grafo();
                    break;
            case 6: mostrar_aristas();
                    break;

            default: cout<<"OPCION NO VALIDA...!!!";
                     break;


        }

        cout<<endl<<endl;
        system("pause");  system("cls");

    }while(op!=7);
    getch();
    return 0;
}

/*                          Menu
------------------------------------------------------------------------*/
void menu()
{
    cout<<"\n\tREPRESENTACION DE GRAFOS DIRIGIDOS\n\n";
    cout<<" 1. INSERTAR UN NODO                "<<endl;
    cout<<" 2. INSERTAR UNA ARISTA             "<<endl;
    cout<<" 3. ELIMINAR UN NODO                "<<endl;
    cout<<" 4. ELIMINAR UNA ARISTA             "<<endl;
    cout<<" 5. MOSTRAR  GRAFO                  "<<endl;
    cout<<" 6. MOSTRAR ARISTAS DE UN NODO      "<<endl;
    cout<<" 7. SALIR                           "<<endl;

    cout<<"\n INGRESE OPCION: ";
```

```
}

/*                      INSERTAR NODO AL GRAFO
-------------------------------------------------------------------------*/
void insertar_nodo()
{
    Tnodo t,nuevo=new struct nodo;
    cout<<"INGRESE VARIABLE:";
    cin>>nuevo->nombre;
    nuevo->sgte = NULL;
    nuevo->ady=NULL;

    if(p==NULL)
     {
        p = nuevo;
        cout<<"PRIMER NODO...!!!";
      }
    else
     {
        t = p;
        while(t->sgte!=NULL)
         {
            t = t->sgte;
          }
        t->sgte = nuevo;
        cout<<"NODO INGRESADO...!!!";
      }

 }

/*                      AGREGAR ARISTA
    funcion que utilizada para agregar la arista a dos nodos
-------------------------------------------------------------------------*/
void agrega_arista(Tnodo &aux, Tnodo &aux2, Tarista &nuevo)
{
    Tarista q;
    if(aux->ady==NULL)
    {   aux->ady=nuevo;
```

```
        nuevo->destino=aux2;
        cout<<"PRIMERA ARISTA....!";
    }
    else
    {   q=aux->ady;
        while(q->sgte!=NULL)
            q=q->sgte;
        nuevo->destino=aux2;
        q->sgte=nuevo;
        cout<<"ARISTA AGREGADA...!!!!";
    }

}
/*                      INSERTAR ARISTA
    funcion que busca las posiciones de memoria de los nodos
    y hace llamado a agregar_arista para insertar la arista
-------------------------------------------------------------------*/
void insertar_arista()
{   char ini, fin;
    Tarista nuevo=new struct arista;
    Tnodo aux, aux2;

    if(p==NULL)
     {
         cout<<"GRAFO VACIO...!!!!";
         return;
     }
    nuevo->sgte=NULL;
    cout<<"INGRESE NODO DE INICIO:";
    cin>>ini;
    cout<<"INGRESE NODO FINAL:";
    cin>>fin;
    aux=p;
    aux2=p;
    while(aux2!=NULL)
    {
        if(aux2->nombre==fin)
        {
```

```
            break;
        }

        aux2=aux2->sgte;
    }
    while(aux!=NULL)
    {
        if(aux->nombre==ini)
        {
            agrega_arista(aux,aux2, nuevo);
            return;
        }

        aux = aux->sgte;
    }
}

/*          FUNCION PARA BORRAR TODAS LAS ARISTAS DE UN NODO
    esta funcion es utilizada al borrar un nodo pues si tiene aristas
    es nesesario borrarlas tambien y dejar libre la memoria
-------------------------------------------------------------------------*/
void vaciar_aristas(Tnodo &aux)
{
    Tarista q,r;
    q=aux->ady;
    while(q->sgte!=NULL)
    {
        r=q;
        q=q->sgte;
        delete(r);
    }
}
/*                      ELIMINAR NODO
    funcion utilizada para eliminar un nodo del grafo
    pero para eso tambien tiene que eliminar sus aristas por lo cual
    llama a la funcion vaciar_aristas para borrarlas
-------------------------------------------------------------------------*/
void eliminar_nodo()
```

```
{   char var;
    Tnodo aux,ant;
    aux=p;
    cout<<"ELIMINAR UN NODO\n";
    if(p==NULL)
     {
         cout<<"GRAFO VACIO...!!!!";
         return;
     }
    cout<<"INGRESE NOMBRE DE VARIABLE:";
    cin>>var;

    while(aux!=NULL)
    {
        if(aux->nombre==var)
        {
            if(aux->ady!=NULL)
                vaciar_aristas(aux);

            if(aux==p)
            {

                    p=p->sgte;
                    delete(aux);
                    cout<<"NODO ELIMINADO...!!!!";
                    return;



            }
            else
            {
                ant->sgte = aux->sgte;
                delete(aux);
                cout<<"NODO ELIMINADO...!!!!";
                return;
            }
        }
```

```
        else
        {
            ant=aux;
            aux=aux->sgte;
         }
    }

}

/*                      ELIMINAR ARISTA
    funcion utilizada para eliminar una arista
-------------------------------------------------------------------------*/
void eliminar_arista()
{
char ini, fin;
    Tnodo aux, aux2;
    Tarista q,r;
    cout<<"\n ELIMINAR ARISTA\n";
    cout<<"INGRESE NODO DE INICIO:";
    cin>>ini;
    cout<<"INGRESE NODO FINAL:";
    cin>>fin;
    aux=p;
    aux2=p;
    while(aux2!=NULL)
    {
        if(aux2->nombre==fin)
        {
            break;
        }
        else
        aux2=aux2->sgte;
    }
     while(aux!=NULL)
    {
        if(aux->nombre==ini)
        {
            q=aux->ady;
```

```
            while(q!=NULL)
            {
                if(q->destino==aux2)
                {
                    if(q==aux->ady)
                        aux->ady=aux->ady->sgte;
                    else
                        r->sgte=q->sgte;
                    delete(q);
                    cout<<"ARISTA  "<<aux->nombre<<"----->"<<aux2->nombre;
                    cout<<" ELIMINADA.....!!!!";
                    return;
                }
            }
            r=q;
            q=q->sgte;
        }
        aux = aux->sgte;
    }
}
/*                      MOSTRAR GRAFO
    funcion que imprime un grafo en su forma enlazada
-------------------------------------------------------------------------*/
void mostrar_grafo()
{   Tnodo ptr;
    Tarista ar;
    ptr=p;
    cout<<"NODO|LISTA DE ADYACENCIA\n";

    while(ptr!=NULL)
    {   cout<<"   "<<ptr->nombre<<"|";
        if(ptr->ady!=NULL)
        {
            ar=ptr->ady;
            while(ar!=NULL)
            {   cout<<" "<<ar->destino->nombre;
                ar=ar->sgte;
             }
```

```
        }
        ptr=ptr->sgte;
        cout<<endl;
    }
}

/*                      MOSTRAR ARISTAS
    funcion que muestra todas las aristas de un nodo seleccionado
-------------------------------------------------------------------------*/
void mostrar_aristas()
{
    Tnodo aux;
    Tarista ar;
    char var;
    cout<<"MOSTRAR ARISTAS DE NODO\n";
    cout<<"INGRESE NODO:";
    cin>>var;
    aux=p;
    while(aux!=NULL)
    {
        if(aux->nombre==var)
        {
            if(aux->ady==NULL)
            {   cout<<"EL NODO NO TIENE ARISTAS...!!!!";
                return;
             }
            else
            {
                cout<<"NODO|LISTA DE ADYACENCIA\n";
                cout<<"   "<<aux->nombre<<"|";
                ar=aux->ady;

                while(ar!=NULL)
                {
                    cout<<ar->destino->nombre<<" ";
                    ar=ar->sgte;
                }
```

```
                cout<<endl;
                return;
            }
        }
        else
        aux=aux->sgte;
    }
}
```

6.2. Ejercicios

6.2.1. Problema de éxitos

Una compañía organiza una encuesta para determinar el éxito de sus productos. Sus productos son discos y casets con éxitos musicales: los éxitos más populares deben estar incluidos en una lista de éxitos. La población encuestada se va a dividir en cuatro categorias según sexo y edad (por ejemplo, menores o iguales a veinte años y mayores de veinte años). A cada persona se le pide que dé el nombre de cinco éxitos. Los éxitos se identifican por los números 1 a N (por ejemplo $N = 30$). Los resultados de la encuesta se presentan por el archivo siguiente:

```
TYPE exito = 1 ... N;
     sexo = (femenino, masculino);
     respuesta =
                  REGISTRO apellido, nombre:alfa;
                           s:sexo;
                           edad:ENTERO;
                           eleccion:ARRAY [1 ... 5] OF exito
                  END
    VAR encuesta:ARCHIVO OF respuesta;
```

Por lo tanto, cada elemento del archivo representa un encuestado e incluye su apellido, nombre, sexo, edad y sus cinco éxitos elegidos en orden de preferencia. Este archivo es la entrada a un programa que calcula los siguientes resultados:

1. Lista de éxitos en orden de popularidad. Para cada uno de la lista el número del éxito y el número de veces que ha sido mencionado en la encuesta. Los éxitos que no han sido mencionados en la encuesta se omiten de la lista.

2. Cuatro listas distintas con nombres y apellidos de todos los encuestados que habian mencionado en primer lugar, uno de los éxitos más populares en su categoría.

Las cinco listas deben ir precedidas por títulos adecuados.

6.2.2. Problema del pingûino

Se tiene un cubo de hielo que se desplaza por el techo de un iglú hasta caer al suelo, describir la trayectoria que realiza el cubo para múltiples velocidades de inicio, considerando la fricción entre el techo del iglú y el bloque de hielo.

6.2.3. Partícula entre campos

Se tienen dos campos uno eléctrico y otro magnético colocados de manera perpendicular entre sí, de tal manera que si se coloca una partícula cargada negativamente entre los campos, esta describirá trayectorias como un circulo, elipse o espiral. Que dependerán del valor de la intensidad de los campos.

6.2.4. Problema de búsqueda

Dar como entrada una palabra que hay que guardar en un arreglo y colocarlo como matriz. Para después buscar en la misma la palabra solicitada.

6.2.5. Problema de suma de matrices

Dar como entrada dos matrices cuadradas de cualquier orden y obtener su suma.

6.2.6. Problema de multiplicación de matrices

Dar como entrada dos matrices cuadradas de cualquier orden y obtener su múltiplicación.

6.2.7. Problema de coordenadas

Hacer la transformación de coordenadas cartesianas en el espacio a cualquier coordenada polar, cilíndrica, esférica o parabólica. Seleccionada de manera aleatoria.

6.2.8. Palindromo con pilas

Hacer un programa con pilas y listas para reconocer si una palabra es un palindromo.

6.2.9. Arreglo bidimensional

Hacer un arreglo bidimensional con pilas. Para construirlo se requiere que cada elemento contenga los siguientes elementos:

```
1 TIPE estructuraBidimensional = REGISTRO
2 {
3 TIPE dato:TipoDato ;
4 TIPE ↑ Der:estructuraBidimensional ;
5 TIPE ↑ Izq:estructuraBidimensional ;
6 TIPE ↑ Abj:estructuraBidimensional ;
7 TIPE ↑ Arr:estructuraBidimensional ;
8 }
```

Algorithm 6.1: Crear los elementos del arreglo bidimensional

```
1 TIPE p:↑ p ;
2 ↑ p := ↑ Lista1 ;
3 if ↑ temp <> NULL then
4 |   e := crearElemento() ;
5 |   temp ↑.Der := e ;
6 |   e.Izq := ↑ temp ;
7 |   e.Arr := NULL ;
8 else
9 e := crearElemento() ;
10 ↑ temp := e ;
11 e.Izq := NULL ;
12 e,Der := NULL ;
13 e.Arr := NULL ;
14 ↑ temp := temp ↑.der ;
15 while p ↑.sig <> NULL do
16 |_ ↑ p := p ↑.sig ;
17 p ↑.sig := ↑ Lista2 ;
```

6.3. Examenes

Resolver los siguientes cuestionarios:

1. Una lista que contenga los cinco primeros números primos.

2. Una lista que contenga los cinco siguientes números primos.

3. Poner la primera lista y después la segunda lista, para obtener una lista que contenga los primeros díez números primos.

4. De la lista de los díez primeros números primos sacar el número cinco y entregar la lista.

5. Dividir la lista del punto 4 en tres listas.

La solución a la primera pregunta del cuestionario es:

```
1  TIPE sig:↑ sig ;
2  TIPE estructuraLista = REGISTRO
3  {
4  TIPE dato:ENTERO ;
5  TIPE sig:estructuraLista ;
6  }
7  TIPE Lista1:↑ Lista1 ;
8  ↑ Lista := NULL ;
9  TIPE nuevoElemento:↑ nuevoElemento ;
10 nuevoElemento := NEW estructuraLista ;
11 nuevoElemento.sig := ↑ Lista ;
12 ↑ Lista := NuevoElemento ;
13 n ;= 2 ;
14 nuevoElemento.dato := n ;
15 TIPE i:ENTERO ;
16 for i = 2, 3, ...,5 do
17 |  nuevoElemento := NEW estructuraLista ;
18 |  nuevoElemento.sig := NULL ;
19 |  ↑ p := nuevoElemento ;
20 |  nuevoElemento.dato := n + 1 ;
```

La solución a la segunda pregunta del cuestionario es:

```
1  TIPE sig:↑ sig ;
2  TIPE estructuraLista = REGISTRO
3  {
4  TIPE dato:ENTERO ;
5  TIPE sig:estructuraLista ;
6  }
7  TIPE Lista2:↑ Lista2 ;
8  ↑ Lista := NULL ;
9  TIPE nuevoElemento:↑El caso base es el caso para el que no se aplica
   la recursividad. El caso general es para el cual la solución es expresada
   en términos de una pequeña versión de si misma. nuevoElemento ;
10 nuevoElemento := NEW estructuraLista ;
11 nuevoElemento.sig := ↑ Lista ;
12 ↑ Lista := NuevoElemento ;
13 n ;= 13 ;
14 nuevoElemento.dato := n ;
15 TIPE i:ENTERO ;
16 for i = 2, 3, ...,5 do
17 |  nuevoElemento := NEW estructuraLista ;
18 |  nuevoElemento.sig := NULL ;
19 |  ↑ p := nuevoElemento ;
20 |  nuevoElemento.dato := n + 2 ;
```

La respuesta a la tercera pregunta del cuestionario es:

```
1  TIPE p:↑ p ;
2  ↑ p := ↑ Lista1 ;
3  while p ↑.sig <> NULL do
4  |  ↑ p := p ↑.sig ;
5  p ↑.sig := ↑ Lista2 ;
```

La respuesta a la pregunta cuatro del cuestionario es:

```
1  TIPE p:↑ p ;
2  ↑ p := ↑ Lista1 ;
3  ↑ a := ↑ Lista1 ;
4  while p ↑.sig <> NULL do
5      if p ↑.dato == 5 then
6          Encontrado() ;
7          a ↑.sig := p ↑.sig ;
8          Return(↑ p) ;
9      else
10         ↑ a := p ↑ ;
11         ↑ p := p ↑.sig ;
12 ↑ ListaRecortada := ↑ p ;
```

La respuesta a la pregunta cinco del cuestionario es:

```
1  TIPE n:ENTERO ;
2  TIPE ListaRecortada1:↑ ListaRecortada1 ;
3  n := 0 ;
4  ↑ p := ↑ ListaRecortada ;
5  while p ↑.sig <> NULL do
6      ↑ p := p ↑.sig ;
7      n := n + 1 ;
8      ↑ ListaRecortada1 := ↑ p ;
9      if n == 3 then
10         ↑ ListaRecortada2 := p ↑.sig ;
11         p ↑.sig := NULL ;
12     if n == 6 then
13         ↑ ListaRecortada3 := p ↑.sig ;
14         p ↑.sig := NULL ;
```

1. Una lista que contenga los cinco primeros números pares.

2. Una lista que contenga los cinco siguientes números pares.

3. Poner la primera lista y después la segunda lista, para obtener una lista que contenga los primeros díez números pares.

4. De la lista de los díez primeros números pares sacar el número seis y entregar la lista.

5. Dividir la lista del punto 4 en tres listas.

La solución a la primera pregunta del cuestionario es:

```
1  TIPE sig:↑ sig ;
2  TIPE estructuraLista = REGISTRO
3  {
4  TIPE dato:ENTERO ;
5  TIPE sig:estructuraLista ;
6  }
7  TIPE Lista1:↑ Lista1 ;
8  ↑ Lista := NULL ;
9  TIPE nuevoElemento:↑ nuevoElemento ;
10 nuevoElemento := NEW estructuraLista ;
11 nuevoElemento.sig := ↑ Lista ;
12 ↑ Lista := NuevoElemento ;
13 n ;= 2 ;
14 nuevoElemento.dato := n ;
15 TIPE i:ENTERO ;
16 for i = 2, 3, ...,5 do
17 |  nuevoElemento := NEW estructuraLista ;
18 |  nuevoElemento.sig := NULL ;
19 |  ↑ p := nuevoElemento ;
20 |  nuevoElemento.dato := n ;
21 |  n := n + 2 ;
```

La respuesta a la segunda pregunta del cuestionario es:

```
1  TIPE sig:↑ sig ;
2  TIPE estructuraLista = REGISTRO
3  {
4  TIPE dato:ENTERO ;
5  TIPE sig:estructuraLista ;
6  }
7  TIPE Lista2:↑ Lista2 ;
8  ↑ Lista := NULL ;
9  TIPE nuevoElemento:↑ nuevoElemento ;
10 nuevoElemento := NEW estructuraLista ;
11 nuevoElemento.sig := ↑ Lista ;
12 ↑ Lista := NuevoElemento ;
13 n ;= 12 ;
14 nuevoElemento.dato := n ;
15 TIPE i:ENTERO ;
16 for i = 2, 3, ...,5 do
17 |  nuevoElemento := NEW estructuraLista ;
18 |  nuevoElemento.sig := NULL ;
19 |  ↑ p := nuevoElemento ;
20 |  nuevoElemento.dato := n ;
21 |  n := n + 2 ;
```

La respuesta a la tercera pregunta del cuestionario es:

```
1  TIPE p:↑ p ;
2  ↑ p := ↑ Lista1 ;
3  while p ↑.sig <> NULL do
4  |  ↑ p := p ↑.sig ;
5  p ↑.sig := ↑ Lista2 ;
```

La respuesta a la pregunta cuatro del cuestionario es:

```
1  TIPE p:↑ p ;
2  ↑ p := ↑ Lista1 ;
3  while p ↑.sig <> NULL do
4  |  ↑ p := p ↑.sig ;
5  p ↑.sig := ↑ Lista2 ;
```

La respuesta a la pregunta cinco del cuestionario es:

```
1 while p ↑.sig <> NULL do
2   ↑ p := p ↑.sig ;
3 p ↑.sig := ↑ Lista2 ;
```

Bibliografía

[1] Dale, Nell B. , C++ plus data structures, 2003 by Jones and Bartlett Publishers, Inc.

[2] Gary J. Bronson, C++ for Engineers and Scientists, Cengage Learning Customer Sales Support, Third Edition.

[3] D. S. Malik, C++ Programming: Program Design Including Data Structures, Fifth edition, CERNING.

[4] Herbert Schildt, C Manual de referencia, McGraw–Hill, cuarta edición, 2001.

[5] William H. Murray III, Chris H. Pappas, Manual de Borland C++, McGraw–Hill, Versión 5.

[6] Mark R. Headington and David D. Rdey, DATA ABSTRACTION AND STRUCTURES USING C++ , JONES AND BARTLETT PUBLISHERS, 1997.

[7] E. Balagurusamy, Object Oriented Programming With C++, MacGraw Hill, fourth edition, 2008.

[8] Robert L. Kruse and Alexander J. Ryba, Data Structures and Program Design in C++, Prentice Hall, 2000.

[9] Mark Allen Weiss, Data Structures and Algorithm Analysis in C, libro en formato chmod.

[10] P. S. Deshpande and O. G. Kakde, C and Data Structures, CHARLES RIVER MEDIA, INC. , 2003.