Estructuras de Datos

José M. Saavedra Rondo

February 28, 2010

1 Introducción

Toda la información que se maneja dentro de un computador se encuentra almacenada en su memoria, que en trminos simples es una secuencia de caracteres (bytes) en donde se encuentran las instrucciones y datos a los que se accede directamente a travs del procesador del computador.

Los sistemas o mtodos de organizacin de datos que permiten un almacenamiento eficiente de la informacin en la memoria del computador son conocidos como **estructuras de datos**. Estos mtodos de organizacin constituyen las piezas bsicas para la construccin de algoritmos complejos, y permiten implementarlos de manera eficiente. Las estructuras de datos permiten luego definir **tipos abstractos de datos**.

- **Tipo Abstracto de Dato (TAD)**: Un modelo computacional definido por una estructura de datos determinada y un conjunto de operaciones sobre dicha estructura: Ej. (Pila, Cola, Cadena son TAD's).

TAD	**operaciones**
Pila	push, pop
Cadena	charAt, append, getLength.
Lista	insert, delete, getValue, sort.

- **Estructuras Estticas y Dinmicas**: Las estructuras de datos gestionan la memoria del computador para poder almacenar los datos. En este sentido existen estructuras estticas y dinmicas segn la gestin de la memoria.
  - **Estructuras Estticas**: Define un tamao fijo (contiguo) de la memoria y solamente en ese espacio reservado se pueden almacenar los datos (Ej. Arreglos)
  - **Estructuras Dinmicas**: El tamao de la memoria no se fija *a priori*, y por lo tanto los datos se encuentran dispersos en la memoria (Ej. Listas enlazadas)
- **Manejo de Memoria**: Al ejecutar un programa la memoria se divide en 3 segmentos. El segmento de Texto, que alamacena el cdigo compilado, el segmento Stack, para almacenar la funciones que se ejecutan junto a sus variables locales y el segmento Heap para varibles globales, estticas y asignacin dinmica de memoria.

2 Arreglos

Un arreglo es una secuencia contigua de un número fijo de elementos homogéneos. Los arreglos se pueden crear según los siguientes ejempos:

```
int A[10]

int *A= (int*)malloc(10*sizeof(int));
```

2.1 Problemas

1. Obtener el mayor de una secuencia.
2. Buscar un elemento en un arreglo.
3. Buscar un elemento en tiempo $O(logn)$.
4. Crear una base de datos de M vectores aleatorios de tamao d cada uno. Ahora, dado un vector p, encontrar el vecino más cercano a p usando la distancia euclideana.

Ejemplo de solución:

```
 //Este programa permite leer datos en un arreglo y obtener el mayor
 #include <stdlib.h>
 #include <stdio.h>

int obtenerMayor(int A[], int N)
{
    int i=0, may=0;
    may=A[0];
    for(i=0;i<N;i++)
    {
       if(A[i]>may)
       {
          may=A[i];
       }
    }
    return may;
}


int* leerDatos(int N)
{
    int  *A=(int*)malloc(N*sizeof(int));
    int i=0;
    printf("Ingreso de datos: \n");
    for(i=0;i<N;i++)
    {
        printf("A[%d]=",i);
        scanf("%d",&A[i]);
```

```
    }
    return A;
}


int main()
{
    int n=0;
    printf("Ingrese n:");
    scanf("%d", &n);
    int *A=leerDatos(n);
    int may=obtenerMayor(A,n);
    printf("El mayor es: %d \n", may);
    return 0;
}
```

2.2 Búsqueda Binaria

Una de las operaciones más frecuentes en un arreglo es la búsqueda de un elemento. En general, la búsqueda requiere revisar (comparar) cada uno de los elementos del arreglo, por lo que el tiempo de búsqueda es proporcional al número de elementos del arreglo. Formalmente, se dice que el algoritmo de búsqueda es de orden n o simplemente se nota $O(n)$. A continuación se presenta un ejemplo de la implementación de una función de búsqueda, que debuelve la posición del elemento buscado si existe. En caso contrario devolverá -1.

```
int buscar(int A[], int N, int valor)
{
    int pos=-1;
    int encontrado=0;
    int i=0;
    while((i<N)&&(!encontrado))
    {
        if(A[i]==valor)
        {
            pos=i;
            encontrado=1;
        }
        i++;
    }
    return pos;
}
```

Sin embargo, si suponemos que los datos almacenados en un arreglo se encuentran ordenados (supondremos una ordenación ascendente) entonces podemos realizar la búsqueda de un elemento usando simplemente log(n) comparaciones, donde n es el tamaño del arreglo. Este algoritmos se denomina **Búsqueda Binaria**.

La idea de la búsqueda binaria es que aprovechando la ordenación de los datos, rápidamente podamos descartar en cada comparación la mitad de los

datos. En este sentido, revisamos el elemento central que lo representamos con x, si el valor que buscamos v es igual a x entonces la búsqueda se detiene y devolvemos la posición correspondiente a x. Si ese no es el caso, pueden suceder dos cosas, que v sea mayor a x o que sea menor. En el primer caso, todos los elementos a la izquierda de x (la primera mitad del arreglo) deben ser descartados pues resultan también menores que v, así, ahora buscamos solamente en los elementos mayores que x, es decir, en la segunda mitad. La búsqueda se prosigue de igual modo mediante la comparación del elemento central. En el segundo caso (v menor que x), se descarta la segunda mitad, buscando ahora solamente en los elementos menores que x. La búsqueda se sigue iterativamente hasta encontrar el elemento buscado o hasta que todos los elementos se hayan descartado. A continuación se muestra el algoritmo de búsqueda:

Algoritmo de Búsqueda binaria

- **INPUT:** A: arreglo, N: tamaño del arreglo, v: elemento a buscar
- **OUTPUT:** posición encontrada, -1 si no existe v.

```
izq=0
der=N-1
encontrado=false
pos=-1;
Mientras((izq<=der)&&(!encontrado))
    posC=(izq+der)/2
    if(v==A[posC])
       encontrado=true
       pos=posC
    fin_if
    if(v>A[posC])
       izq=posC+1
    fin_if
    if(v<A[posC])
       der=posC-1
    fin_if
fin_mientras
devolver pos
```

3 Arreglos Multidimensionales

Similar al caso unidimensional, uno puede crear arreglos de más de una dimensión. El caso más simple son las matrices formadas por 2 dimensiones. La creación de una matriz se puede realizar de las siguientes maneras:

```
(Ejemplo para una matriz de dos filas y  3 columnas)

int A[2][3]

int **A= (int**)malloc(2*sizeof(int*));
for(i=0;i<3;i++)
```

```
{
   A[0]=(int*)malloc(3*sizeof(int));
}
```

3.1 Problemas

- Dada una matriz M de $n \times m$ determinar si M es triangular superior, triangular inferior o no es triangular.
- Leer dos matrices numéricas y obtener el producto matricial de ambas.
- Dada un matriz de enteros que varían entre 1 y 256, crear un arreglo que almacene la frecuencia de ocurrencia de cada número en la matriz. El arreglo es de tamaño 256. ¿Por qué ?.
- Representar un grafo, el cual representa distancias entre ciudades, mediante una matriz de modo que se pueda buscar la ciudad más cercana y lejana de una ciudad dada. (La ciudad puede ser representada como un índice)
- Hemos visto que buscar el mayor de un arreglo se realizar en n pasos. Sin embargo, si queremos buscar el segundo mayor lo podemos hacer en $log(n)$ pasos. Utilizando arreglos implemente una función que busque el mayor en n pasos, pero el segundo mayor en $log(n)$ pasos. Este es el algoritmo del torneo.

Ejemplo de solución:

```
 /*Este programa permite leer matrices e implementa la multiplicacion matricial*/
#include <stdio.h>
#include <stdlib.h>

/*-----------------------------------------------------*/
int **leerMatriz(int m, int n)
{
    int **A=(int**)malloc(m*sizeof(int*));
    int i=0,j=0;
    for(i=0;i<m;i++)
    {
        A[i]=(int*)malloc(n*sizeof(int));
    }

    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("M[%d][%d]=",i,j);
            scanf("%d",&A[i][j]);
        }
    }
    return A;
}
```

```
/*-------------------------------------------------------*/
void reportarMatriz(int **A, int m, int n)
{
    int i=0, j=0;

    for(i=0;i<m;i++)
    {
        for(j=0;j<n;j++)
        {
            printf("%d \t",A[i][j]);
        }
        printf("\n");
    }
    printf("\n");
}
/*-------------------------------------------------------*/
int **multiplicarMatriz(int **A, int **B, int m, int p, int n)
{
    int i=0, j=0, k=0, s=0;
    int **C=(int**)malloc(m*sizeof(int*));
    for (i=0;i<m;i++)
    {
        C[i]=(int*)malloc(n*sizeof(int*));
        for(j=0;j<n;j++)
        {
            s=0;
            for(k=0;k<p;k++)
            {
                s+=A[i][k]*B[k][j];
            }
            C[i][j]=s;
        }
    }
    return C;
}
/*-------------------------------------------------------*/
int main()
{

    int m1=0,n1=0;
    int m2=0, n2=0;
    int **A=NULL;
    int **B=NULL;
    int **C=NULL;

    printf("Ingrese dimensiones de la matriz A \n");
    scanf("%d",&m1);
    scanf("%d",&n1);
    A=leerMatriz(m1,n1);
```

```
    printf("Ingrese dimensiones de la matriz B \n");
    scanf("%d",&m2);
    scanf("%d",&n2);
    B=leerMatriz(m2,n2);

    printf("\n");
    if(n1==m2)
    {
        C=multiplicarMatriz(A,B, m1,n1,n2);
        printf("A: \n");
        reportarMatriz(A,m1,n1);
        printf("B: \n");
        reportarMatriz(B,m2,n2);
        printf("AxB: \n");
        reportarMatriz(C,m1,n2);
    }
    else
    {
        printf("Error: Las matrices no se pueden multiplicar\n");
    }
    return 0;
}
```

4 Cadenas

Una cadena (string) es una secuencia de caracteres representado por un arreglo. El final de una cadena se representa con un caracter especial, el caracter nulo (\0).

4.1 Algunas funciones de C para cadenas

Para leer una secuencia de caracteres (string) desde la entrada estándar es recomendable usar la función *fgets*. Cuyo uso es como sigue:

```
fgets(buffer, MAX, stdin)
```

donde, buffer es el espacio de memoria donde se almacenará la secuencia. En la práctica buffer es un arreglo de *char* de tamaño MAX. A continuación se muestra una ejemplo para contar las ocurrencias de un caracter en una cadena:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int contar(char A[], char c)
{
    int i=0, cont=0;
    for(i=0;i<strlen(A);i++)
    {
```

```
        if (A[i]==c)
        {
            cont++;
        }
    }
    return cont;
}

int main()
{
    char cad[10];
    char c='\0';
    int cont=0;
    printf("Ingrese cadena:");
    fgets(cad,10,stdin);//es mejor usar fgets que simplemente gets
    printf("Ingrese caracter:");
    scanf("%c",&c);
    cont=contar(cad,c);
    printf("Hay %d  ocurrencias del caracter %c \n", cont, c);
    return 0;
}
```

Las siguiente funciones requieren la inclusión de *string.h*

- strlen: Devuelve la cantidad de caracteres (longitud) de una cadena.

  ```
  int strlen(char *cad)
  ```

- strcmp: Compara dos cadenas devolviendo un número entero.

  ```
  int strcmp(char *cad1, char *cad2)
  ```

  Devuelve :

  - 0 si $cad1 == cad2$.
  - < 0 si $cad1 < cad2$.
  - > 0 si $cad1 > cad2$.

- strcpy: Copia una cadena en otra.

  ```
  char *strcpy(char *destino, char *fuente)
  ```

5 Ordenación

La ordenación consiste en reorganizar los datos, respecto de una llave, residentes en memoria (Ej. en un arreglo) de modo que la nueva organización mantenga la relación *menor que* (para el caso ascendente) o *mayor que* (para el caso descendente) entre elementos consecutivos.

5.1 Algoritmos de Ordenación

Probablemente la ordenación es el problema para el cual existen la mayor cantidad de algoritmos. Algunos de estos algoritmos son:

- Ordenación por Selección (Selection Sort)
- Ordenación Burbuja (Bubble Sort)
- Ordenación por Inserción (Insertion Sort)
- Shell Sort
- Ordenación por Mezcla (Merge Sort)
- Ordenación Rápida (Quick Sort).
- Otros para casos particulares: Counting Sort, Bucket Sort, etc.

5.1.1 Ordenación por Selección

La idea es que para cada celda p (de 0 hasta $n-2$) vamos a seleccionar la celda q que contiene al menor valor, suponiendo una ordenación ascendente sin pérdida de generalidad, de los que faltan ordenar e intercambiamos el valor entre las celdas p y q.

Algoritmo: Selection Sort

```
input: A: arreglo, n: cantidad de elementos
output: A ordenado

para i=1 hasta n-2 hacer
    para j=i+1 hasta n-1 hacer
        si (A[i]>A[j])
            intercambiar(A[i], A[j])
       fin_si
   fin_para
fin_para
```

Este algoritmo tiene un costo de procesamiento de $O(n^2)$. Si los datos ya están ordenados, el algoritmo mantiene su costo cuadrático.

5.1.2 Ordenación Burbuja

Este algoritmo es similar al caso anterio, pero en este caso los datos se comparan por parejas haciendo que los mayores pasen al final (como una burbuja).

Algoritmo: Bubble Sort

```
input: A: arreglo, n: cantidad de elementos
output: A ordenado

para i=n-1 hasta 1 hacer
    para j=0 hasta i-1 hacer
        si (A[j]>A[j+1])
            intercambiar(A[j], A[j+1])
       fin_si
   fin_para
fin_para
```

Este algoritmo también tiene un cost cuadrático $O(n^2)$. Sin embargo, existe una mejora sobre este algoritmo que permite detectar si el arreglo ya está ordenado y por lo tanto no hacer más comparaciones. La mejora consiste en mantener una variable booleana que determine en cada iteración del segundo bucle si hubo algún intercambio, si no lo hubo entonces los datos deberían ya estar ordenados y no se requiere seguir evaluando otro valor de i.

Algoritmo: Bubble Sort(2)

```
input: A: arreglo, n: cantidad de elementos
output: A ordenado

intercambio=true
i=n-1
mientras(intercambio)
    intercambio=false
    para j=0 hasta i-1 hacer
        si (A[j]>A[j+1])
            intercambiar(A[j], A[j+1])
            intercambio=true
       fin_si
   fin_para
   i=i-1
fin_mientras
```

Con esta mejora, el algoritmo sigue teniendo un costo cuadrático en el peor caso. Sin embargo, en el caso que el arreglo ya esté ordenado, solamente requiere realizar una pasada por los datos por lo que su tiempo es $O(n)$.

5.1.3 Ordenación por Inserción

Este método se asemeja a la ordenación de una mano de cartas. Suponiendo que los elementos de 0 hasta $i-1$ se encuentran ordenados entonces el elemento de la posición i debe retroceder hasta encontra su posición correcta.

Algoritmo: Insertion Sort

```
Input: A: arreglo, n: cantidad de elementos
Output: A ordenado

para i=1 hasta n-1
    x=A[i]
    j=i-1
    mientras((j>=0)&&(A[j]>x))
        A[j+1]=A[j]
    fin_mientras
    A[j+1]=x
fin_para
```

El algoritmo de inserción también tiene costo $O(n^2)$ en el peor caso. Sin embargo, en el mejor caso, cuando los datos ya están ordenados el algoritmo tiene un costo lineal $O(n)$.

5.1.4 Ordenación por Mezcla

Este algoritmo utiliza la estrategia **Divide y Conquista** para resolver el problema de ordenación. La idea principal es dividir el arreglo en mitades, ordenar cada mitad siguiendo el mismo algoritmo (idea recursiva) y luego mezclar las dos mitades ya ordenadas. Ordenar dos arreglos ya ordenados se puede realizar en tiempo $O(n)$ usando el algoritmo de mezcla o itercalación.

Entonces los pasos generales son:

1. Separar el arreglo en dos mitades.
2. Ordenar cada una de las mitades por separado, siguiendo el mismo algoritmo.
3. Mezclar las dos mitades para obtener el resultado final.

Algoritmo: Merge

```
Input: Arreglo (A),
       inicio(izq), fin(der), corte(p)
Output: Se mezcla A[izq...p] con A[p+1..der]

i=izq
j=p+1
B=crear arreglo auxiliar de der-izq+1 celdas.
k=0
mientras ((i<=p)AND(j<=der))
    si (A[i]<A[j])
        B[k]=A[i]
        i=i+1
    sino
        B[k]=A[j]
        j=j+1
    fin_si
    k=k+1
```

```
fin_mientras
mientras(i<=p)
    B[k]=A[i]
    k=k+1
    i=i+1
fin_mientras
mientras(j<=der)
    B[k]=A[j]
    k=k+1
    j=j+1
fin_mientras
para i=0 hasta der-izq
    A[izq+i]=B[i]
fin_para
```

Algoritmo: Merge_Sort

```
Input: Arreglo (A)
       inicio(izq), fin(der)
Output: A ordenado ascendentemente

si(izq<der)
    p=(izq+der)/2
    Merge_Sort(A,izq,p)
    Merge_Sort(A,p+1,der)
    Merge(A,izq,der,p)
fin_si
```

El costo del ordenamiento por mezcla está dado por el costo de *Merge* y de los llamados recursivos a *Merge_Sort*. Es fácil darse cuenta que el costo de *Merge* es lineal $O(n)$. Por lo tanto, podemos definir el costo total en forma recursiva como:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{si } n > 1 \end{cases}$$

Resolviendo la recurrencia se tiene que el costo T es de orden $nlog(n)$, es decir el algoritmo tiene un costo $O(nlogn)$. Discutiremos cómo se resuelven recurrencias más adelante.

5.1.5 Ordenación Rápida

Este problema es del mismo tipo del algoritmo de mezcla, es decir ambos utilizan la estrategia **Divide y Conquista**. En este caso, la idea es partir el arreglo respecto de un valor del arreglo llamdo **pivote**. La partición reubica el pivote de modo que los elementos de la izquierda sean menores o iguales que el pivote (suponiendo una ordenación ascendente) y los de la derecha sean mayores o iguales. Así, ahora simplemente habrá que ordenar en forma separada cada una de las partes.

Entonces los pasos generales son:

1. Elegir un pivote del arreglo.
2. Particionar el arreglo respecto del pivote.
3. Ordenar cada una de las dos partes.

Algoritmo: Particionar

```
Input: Arreglo (A),
       inicio(izq), fin(der)
Output: Devuelve el punto de particion

   p=A[izq];//el pivote es el primero
   i=izq;
   j=der-1;
   aux=0
   piv=der;
   intercambiar A[izq] con A[der]

   mientras(i<j)

           mientras((A[i]<p)&&(i<j))
               i++
           fin_mientras
           mientras((A[j]>p)&&(j>i))
               j--
           fin_mientras
           si(i<j)
               intercambiar A[i] con A[j]
               i++;
               j--;
           fin_si
   fin_mientras
   si(A[j]<=p)
       intercambiar A[piv] con A[j+1]
       devolver j+1
   sino
       intercambiar A[piv] con A[j]
       devolver j

   fin_si
```

Algoritmo: Quick_Sort

```
Input: Arreglo (A)
       inicio(izq), fin(der)
Output: A ordenado ascendentemente

si(izq<der)
   k=partirQS(A,izq,der);
```

```
    Quick_Sort(A,izq,k-1);
    Quick_Sort(A,k+1,der);
fin_si
```

En este caso, el algoritmo depende de la elección del pivote. Es recomendable determinar el pivote aleatoriamente para minimizar la probabilidad de la ocurrencia de malos casos. Pues *Quick Sort* tiene costo cuadrático en el peor caso. Sin embargo, en promedio, su desempeño práctico es mejor que *Merge Sort*. El costo promedio de *Quick Sort* también es $O(nlogn)$.

6 Inttroducción al Análisis de Algoritmos

Los algoritmos realizan un número de operaciones de acuerdo a la cantidad de datos (n) que procesan. El análisis de algoritmos consiste en determinar teóricamente el orden de la cantidad de estás operaciones. Cuando el número de operaciones es fijo o constante y no depende de n se dice que el algoritmo es de orden 1 y lo representaremos como $O(1)$. En este sentido, un número fijo de asignaciones, multiplicaciones o comparaciones será de orden 1 ($O(1)$).

En general definimos el número de operaciones como $T(n)$, cumpliendo:

- $T(n) \in$ R, $T(n) \geq 0$.
- $n \in$ N, $n \geq 0$.

y $T(n) = O(f(n))$ si:

$$T(n) \leq cf(n) \tag{1}$$

para $c \in$ R y $c > 0$.

Por lo general, $f(n)$ tiene un solo término (Ej, 1, n, n^2, 2^n) y es la menor función que cumple Eq.1. Entonces, si un algoritmo está asociado a un $T(n)$ y $T(n) = O(f(n))$ diremos que el algoritmo es de orden $f(n)$. Así podríamos tener los siguiente tipos de algoritmos:

- Algoritmo de tiempo constante, si $T(n) = O(1)$.
- Algoritmo logarítmico, si $T(n) = O(logn)$.
- Algoritmo lineal, si $T(n) = O(n)$.
- Algoritmo cuadrático, si $T(n) = O(n^2)$.
- Algoritmo exponencial, si $T(n) = O(a^n)$, para una constante a.

Así, mientras menor sea el orden de los algoritmos, más eficientes son estos.

Muchos algoritmos tienen un $T(n)$ representado por un polinomio de la forma:

$$T(n) = a_0 + a_1 n + a_2 n^2 + \cdots + a_p n^p,$$

en este caso el orden del algoritmo está gobernado por el orden del polinomio. Por lo tanto, $T(n) = O(n^p)$. Por ejemplo, si un algoritmo tiene asociado un $T(n) = \frac{1}{2}n^2 + n - 3$, el algoritmo tiene orden cuadrático, es decir $T(n) = O(n^2)$.

Regla del Máximo: La regla del máximo dice que si $T(n) = t_1(n) + t_2(n)$, siendo $t_1 = O(f_1(n))$ y $t_2(n) = O(f_2(n))$, entonces $T(n) = O(max(f_1(n), f_2(n)))$.

6.1 Recurrencias

Los algoritmos recursivos expresan su $T(n)$ también en forma recursiva, como por ejemplo el número de operaciones del algoritmo *Merge_Sort* está dado por:

$$T(n) = \begin{cases} 1 & \text{si } n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{si } n > 1 \end{cases}$$

Calcular el orden de un $T(.)$ recursivo no es fácil, por lo que debemos transformar $T(.)$ en una funcíon equivalente pero no recursiva. Este proceso se denomina *resolución de recurrencia*.

Algunas estrategias para resolver la recurrencia se muestran a continuación.

6.1.1 Recurrencias Teléscopicas

Las recurrencias telescópicas tienen la siguiente forma:

$$X_{n+1} = a_n + X_n$$

resolviendo X_n hasta el caso base X_0 tenemos:

$$\begin{aligned} X_{n+1} &= X_0 + a_0 + a_1 + \cdots + a_n \\ X_{n+1} &= X_0 + \sum_{i=0}^{n} a_i \end{aligned}$$

Ejercicios: Resolver las siguientes recurrencias:

1. $T(n) = 2^n + T(n-1)$, donde $T(0) = 1$.
2. $T(n) = T(n-1) + log_2(n)$, donde $T(0) = 1$.

6.1.2 Cambios de variable

En algunos casos, la forma telescópica no se observa directamente. Utilizando cambios de variable podemos hacer notoria la forma anterior y facilitar la resolución de la recurrencia.

Por ejemplo, sea $T(n) = 2T(\frac{n}{2}) + n$ con el caso base $T(n) = 1$ cuando $n \leq 1$.

Para solucionar este problema supondremos que $n = 2^p$, entonces tenemos:

$$T(2^p) = 2T(2^{p-1}) + 2^p$$

La forma telescópicas no es notoria, pero veamos lo que ocurre si hacemos que $T(2^p) = y(p)$, tedremos:

$$y(p) = 2y(p-1) + 2^p$$

y luego dividimos todo por 2^p, haciendo que $y(p)/(2^p) = r(p)$ tendremos

$$r(p) = r(p-1) + 1$$

ahora sí tenemos la forma telescópica que al resolverla nos queda:

$$\begin{aligned} r(p) &= r(0) + \sum_{i=0}^{p-1} 1 \\ r(p) &= 1 + p \end{aligned}$$

Ahora obtenemos $T(n) = y(p) = r(p)2^p = 2^p + p2^p = O(nlogn)$,

Ejercicios: Resolver las siguientes recurrencias:

1. $T(n) = 2T(n-1) + 1$, donde $T(0) = 1$.

6.1.3 Ecuación característica

Sea la ecuación de la forma:

$$a_k T_{n+(k)} + a_{k-1} T_{n+(k-1)} + \cdots + a_0 T_n = 0$$

reemplazamos $T_n = \lambda^n$, y tenemos:

$$\lambda^n (a_k \lambda^k + a_{k-1}\lambda^{k-1} + \cdots + a_0 \lambda^0) = 0$$

pero como λ^n no puede ser 0, entonces:

$$a_k \lambda^k + a_{k-1}\lambda^{k-1} + \cdots + a_0 \lambda^0 = 0.$$

Ahora, tenemos k raíces: $\lambda_1, \ldots, \lambda_k$.

Solución General: $T_n = c_1\lambda_1^n + c_2\lambda_2^n + \cdots + c_k\lambda_k^n$

Ejercicios: Resolver las siguientes recurrencias:

1. $T(n) = 5T(n-1) - 4T(n-2)$, donde $T(1) = 3, T(2) = 15$.
2. $T(n+1) = \sqrt{T(n)T(n-1)}$, donde $T(0) = 1, T(1) = 2$.

7 Lista Enlazadas

7.1 Preliminares

Antes de entrar al detalle de las listas enlazadas, recordemos algunos conceptos sobre el manejo de memoria.

1. **Dirección de memoria**: La memoria del computador se administra mediante su asignación apropiada. La memoria se divide en celdas de cierto tamaño medido en bytes. Estas celdas quedan identificadas mediante una dirección,**la dirección de memoria**. Por lo general, las direcciones de memoria se especifican con números hexadecimales (Ej. 0000, A0C3, FFFF, etc.).

2. **Puntero**: En programación, un puntero es una variable que almacena una dirección de memoria de un espacio reservado. En lenguaje C, para definir un puntero se utiliza el símbolo $*$. Así, por ejemplo un puntero a un entero se define de la siguiente manera:

   ```
   int *a=NULL;
   ```

   una dirección nula **NULL** se utiliza para indicar que el puntero aún no tiene una dirección establecida.**NULL** solamente pueden ser asignado a variables de tipo puntero.

3. **Referencia de una variable**: Toda variable ocupa un espacio en memoria, las variables que no son punteros representan directamente el dato almacenado, mientras que los punteros representan la dirección del dato. Así, a un puntero de tipo entero no se le puede asignar directamente un valor entero.

```
int a=5;//correcto
int *b=5;//incorrecto
```

para obtener la dirección de memoria de una variable se utiliza el operador **referenciador &**. Por ejemplo, en:

```
int a=5;
int *b=&a; //correcto
```

en caso mostrado, la variable b apunta al contenido de a. Por lo tanto, si a sufre alguna modificación el contenido apuntado por b también se habrá modificado, simplemente por ser el mismo espacio de memoria.

Por otro lado, para extraer el dato de un puntero debemos **derreferenciar**, es decir a partir de la referencia obtener el contenido. Esto lo podemos hacer usando el operador **derreferenciador** $*$. Por ejemplo, en el siguiente código:

```
int a=5;
int *b=&a;
int c=*b;
printf("c=%d\n",c);
```

en este caso, c almacenará el valor 5. Piense ahora qué se imprime con:

```
printf("b=%p\n",&b);
```

Es importante no confundir el uso de $*$ cuando se utiliza para definir un puntero (en la declaración de la variable) con el uso como operador derreferenciador. **Uso de punteros**: Los punteros se pueden usar para diferentes propósitos: para crear estructuras dinámicas (como se verá en esta sección), para modificar parámetros de funciones permitiendo devolver valores en los parámetros. Los arreglos siguen este último caso, pues las variables que representan a los arreglos en realidad son punteros a la primera celda del arreglo. De este modo, cuando una función o procedimiento modifica el contenido de un arreglo definido como parámetro entonces esa modificación se hace realmente sobre el arreglo enviando com argumento.

Para entender con mayor claridad cómo se usan los punteros para modificar los valores de los parámetros, pensemos en el problema de crear un procedimiento que permita intercambiar los valores de dos variables. La idea es tener algo como lo siguiente:

```
int main()
{
   int a=5;
   int b=3;
   intercambiar(a,b);
   printf("a=%d",a);//debe aparecer 3
   printf("b=%d",b);//debe aparecer 5
}
```

¿Cómo debería escribirse el procedimiento *intercambiar*?

Un primer intento sería tener el siguiente procedimiento:

```
int intercambiar(int x, int y)
{
   int aux=x;
   x=y;
   y=aux;
}
```

esta solución está EQUIVOCADA, pues el intercambio solamente se realiza en las variables locales del procedimiento *intercambiar*, pero no en los argumentos, es decir los cambios no se visualizarán en las variables que se envían al procedimiento. La solución correcta debe ser:

```
int intercambiar(int *x, int *y)
{
   int aux=*x;
   *x=*y;
   *y=aux;
}
```

y la forma correcta de usarla es como sigue:

```
int main()
{
   int a=5;
   int b=3;
   intercambiar(&a,&b);//enviamos las referencias para que
                       //las variables se modifiquen
   printf("a=%d",a);//aparece 3
   printf("b=%d",b);//aparece 5
}
```

4. Asignación de memoria: La asignación de memoria se realiza usando la función *malloc* como lo veníamos haciendo con los arreglos. La función **malloc** busca un espacio libre de memoria y devuelve la dirección de inicio del espacio encontrado.

   El uso de malloc requiere un poco de cuidado, pues la memoria se reserva en el *heap*, y en esta zona el espacio reservado no se libera hasta el término

del programa a diferencia del espacio ocupado por variables locales en el *stack*. Por lo tanto, una vez que el espacio no es más ocupado debemos LIBERALO usando la función *free*. El esquema de trabajo es como sigue:

(a) Definición del puntero y asignación de memoria con *malloc*.

(b) Uso del espacio asignado.

(c) Liberación del espacio usando *free*.

Un ejemplo sería:

```
int *a=(int*)malloc(sizeof(int));
//usar a
free a;
```

7.2 Lista Enlazada (LE)

Una lista enlazada es un TAD, que permite almacenar una secuencia de datos. A diferencia de los arreglos, el espacio ocupado crece o decrece dinámicamente a medida que se inserten o se eliminen nuevos datos. El tamaño de la estructura depende exclusivamente de los datos almacenados en un momento dado.

El uso de listas enlazadas es aconsejable en casos en donde no se sabe *a priori* la cantidad de elementos que se tendrá, o en casos en los que la cantidad de datos es altamente variable.

A consecuencia del dinámismo de la estructura, los datos almacenados no se encuentran todos en un espacio consecutivo de memoria, por lo que no es suficiente conocer la primera dirección de memoria de los datos como en el caso de los arreglos. Aquí, debe existir algúna forma de enlazar los datos. Los datos se enlazan mediante punteros al siguiente dato. Así, los datos se encapsulan en una estructura que llamaremos **Nodo** que contiene el dato de interés y un puntero al siguiente nodo.

7.2.1 Nodo

El nodo es el componente clave de una LE, en realidad tendremos una lista enlazada de nodos. El nodo es una estructura de datos compuesta por:

- Dato: Representa al dato de interés.
- Sgt: Puntero al siguiente nodo.

Suponiendo que nuestros datos son enteros, crearemos el tipo de dato TNodo mediante la siguiente estructura:

```
typedef struct Nodo
{
    int dato;
    struct Nodo *sgt;
}TNodo;
```

7.3 Operaciones sobre LE

Podemos implementar diversar operaciones sobre una lista enlazada, en realidad eso dependerá de las aplicaciones que querramos desarrollar. Entre las operaciones más utilizadas tenemos:

- Insertar un elemento en la lista.
- Insertar un elemento en la lista, en un posición i dada.
- Eliminar un elemento de la lista.
- Buscar un elemento de la lista.
- Reportar elementos.
- Ordenar la lista.
- Insertar un elemento en orden.
- Preguntar si la lista está vacía.
- Obtener la cantidad de elementos de la lista, etc.

Para implementar estas operaciones debemos estructurar adecuadamente la lista enlazada. Una lista enlazada solamente necesita conocer la referencia del primer nodo, por lo que su estructura puede definirse simplemente como sigue:

```
typedef struct LE
{
    TNodo *inicio;
}TLE;
```

en donde TLE es nuestro tipo de datos creado que representa un lista enlazada. Para faciliar la implementación de las operaciones creamos dos funciones, una para crear un nodo y la otra para crear una lista enlazada, esta funciones se definen a continuación:

```
 TNodo *crearNodo(int x)
{
    TNodo *nodo=(TNodo*)malloc(sizeof(TNodo));
    nodo->dato=x;
    nodo->sgt=NULL;
    return nodo;
}

TLE *crearLista()
{
    TLE *lista=(TLE*)malloc(sizeof(TLE));
    lista->inicio=NULL;
    return lista;
}
```

7.3.1 Insertar elemento

Por defecto los nuevos elementos se insertarán al final. Así, la operación de inserción simplemente sigue los siguientes pasos:

- Crear nodo nuevo.
- Si la lista está vacía, el nodo nuevo es el primer nodo.
- En otro caso, ir hasta el final de lista avanzando de nodo en nodo, y hacer que el el puntero *sgt* del último nodo apunte al nodo nuevo. Para avanzar de nodo en nodo se utiliza un puntero auxiliar que empieza con el nodo inicial de la lista y avanza mediante el puntero *sgt* de cada nodo.

La implementación sería:

```
 void insertar(TLE *lista, int x)
 {
    TNodo *nodo=crearNodo(x);
    TNodo *p=NULL;
    if(lista->inicio==NULL)
    {
        lista->inicio=nodo;
    }
    else
    {
        p=lista->inicio;
        while(p->sgt!=NULL)
        {
            p=p->sgt;
        }
        p->sgt=nodo;
    }
}
```

7.3.2 Reportar elementos

En este caso los pasos son:

- Obtener el primer nodo de la lista. Se utliza un puntero p a un nodo que empieza en el primer nodo.
- Mientras no se llegue al final de la lista (p diferente de NULL).
  - Mostrar el dato de p.
  - Avanzar p ($p = p->sgt$).

Tenemos la siguiente implementación:

```
void reportar(TLE lista)
{
    TNodo *p=NULL;
    p=lista.inicio;
    while(p!=NULL)
```

```
    {
        printf("%d \n",p->dato);
        p=p->sgt;
    }
}
```

Debemos notar la diferencia entre el parámetro *lista* de *insertar* con el de *reportar*. En el caso de insertar *lista* es un puntero, pues el proceso de inserción modificará la lista y queremos que ese cambio se refleje en el argumento. Mientras que el procedimiento *reportar* no afecta de alguna manera la lista.

A continuación se presenta un programa completo con las funciones implementadas:

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

typedef struct Nodo
{
    int dato;
    struct Nodo *sgt;
}TNodo;

typedef struct LE
{
    TNodo *inicio;
}TLE;

TNodo *crearNodo(int x)
{
    TNodo *nodo=(TNodo*)malloc(sizeof(TNodo));
    nodo->dato=x;
    nodo->sgt=NULL;
    return nodo;
}

TLE *crearLista()
{
    TLE *lista=(TLE*)malloc(sizeof(TLE));
    lista->inicio=NULL;
    return lista;
}

void insertar(TLE *lista, int x)
{
    TNodo *nodo=crearNodo(x);
    TNodo *p=NULL;
    if(lista->inicio==NULL)
    {
        lista->inicio=nodo;
    }
```

```
    else
    {
        p=lista->inicio;
        while(p->sgt!=NULL)
        {
            p=p->sgt;
        }
        p->sgt=nodo;
    }
}

void reportar(TLE lista)
{
    TNodo *p=NULL;
    p=lista.inicio;
    while(p!=NULL)
    {
        printf("%d \n",p->dato);
        p=p->sgt;
    }
}

int main()
{
    TLE *l=crearLista();
    insertar(l,1);
    insertar(l,2);
    insertar(l,3);
    reportar(*l);
    return 0;
}
```

7.3.3 Eliminar elemento

Supongamos que lo que queremos hacer es eliminar el nodo de la lista que contiene una cierto dato. La idea simplemente es encontrar el nodo correspondiente q y hacer que el nodo anterior a q apunte al siguiente de q. Con esto la lista deja de tener acceso a q. Finalmente, liberamos el espacio que ocupaba q usando la función $free$. Miremos la implementación:

```
 void eliminar(TLE *lista, int dato)
{
    TNodo *p=lista->inicio;
    TNodo *ant=NULL;
    int encontrado=0;

    while((p!=NULL)&&(!encontrado))
    {
        if(p->dato==dato)
```

```
        {
            encontrado=1;
        }
        else
        {
            ant=p;
            p=p->sgt;
        }
    }

    if(p!=NULL)//si lo encontro
    {
        if (ant==NULL)//primero
        {
            lista->inicio=(lista->inicio)->sgt;
        }
        else
        {
            ant->sgt=p->sgt;
        }
      free(p);
    }
}
```

Ejercicios:

1. Suponga que los nodos de una lista están numerados desde el 0 hasta el $n-1$ desde el inicio de la lista hasta el final. Llamaremos a esta numeración el **orden relativo**. Implmente una función para encontrar el orden relativo del nodo que contiene un cierto dato.

2. Implemente una procedimiento para insertar un nodo en una lista enlazada, de modo tal que los nodos siempre estén ordenados ascendentemente respecto del dato. Es decir, el proceso de inserción debe buscar el lugar más apropiado para insertar el nodo.

3. A diferencia del ejercicio anterior, usted tiene una lista enlazada desordenada, implemente una procedimiento para ordenar los nodos respecto del dato. Suponga una ordenación ascendente.

7.4 Listas doblemente enlazadas

En una lista enlazada simple los nodos mantienen un solo enlace al siguiente nodo, y la lista enlazada mantiene solamente un puntero al primer nodo. En una lista doblemente enlazada los nodos mantienen dos punteros, uno al siguiente nodo y otro al nodo anterior. De este modo, la lista se puede recorrer fácilmente de inicio al final o en sentido inverso desde el final hasta el primer nodo. En este caso, la lista debe mantener, además del puntero al primer nodo, un puntero al último nodo.

Los algoritmos para insertar, eliminar, o cualquier otro que requiere la actualización de los nodos tienen que tener en cuenta que ahora será necesario

actualizar dos enlaces por cada nodo.

Ejercicios:
Implemente estructuras apropiadas para crear una lista doblemente enlazada así como procedimientos de inserción, eliminación y reporte de elementos.

7.5 Listas circulares

Las lista circulares pueden ser implementadas como listas enlazadas simples con la característica que el último nodo debe apuntar al primero, de este modo se forma un ciclo con los nodos. La lista debe mantener un puntero al nodo inicial. Las listas doblemente enlazadas también pueden extenderse a listas circulares.

Ejercicios:
Extienda la implementación de una lista enlazada simple de modo que ahora tenga el comportamiento de una lista circular. Modifique los procedimientos de insertar, eliminar y reportar para que se comporten según las listas circulares.

8 Pilas y Colas

8.1 Pila

Una pila(stack) es un tipo abstracto de dato, cuya estructura es implementada comúnmente en forma dinámica, donde los elementos son insertados y sacados por un mismo lugar (la cima de la pila, o el frente). La pila es una estructura LIFO (First in First out), es decir el úlimo elemento insertados es el primero en ser sacado.

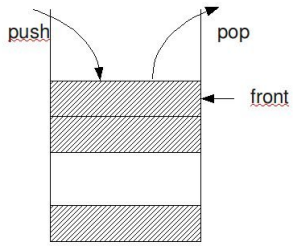

Figure 1: Esquema del TAD Pila

El TAD Pila se puede implementar fácilmente como una lista enlazada, con las siguiente operaciones:

- **crearPila**: Crea una pila vacía, devuelve un puntero a la pila.
- **push**: Inserta un elemento en el frente de la pila.
- **pop**: Saca elemento del frente de la pila, esto elimina el elemento.
- **front**: Consulta por el dato almacenado en el frente de la pila. El tipo de dato de retorno dependerá del dato almacenado en cada nodo.
- **isEmpty**: Consulta si la pila está vacía. Devolverá un dato booleano.

8.1.1 Implementación

Suponiendo que lo que necesitamos almacenar es un entero, una implementación usando una lista enlazada podría ser la siguiente:

```
 /*Implementacio'n de una pila de enteros*/
#include<stdio.h>
  #include<stdlib.h>
  /*----------------------------------*/
  typedef struct Nodo
  {
     int dato;
     struct Nodo *sgt;
  }TNodo;
  /*----------------------------------*/
  typedef struct Pila
  {
     TNodo *front;
  }TPila;
  /*----------------------------------*/
  TNodo *crearNodo(int x)
  {
     TNodo *nodo=(TNodo*)malloc(sizeof(TNodo));
     nodo->dato=x;
     nodo->sgt=NULL;
     return nodo;
  }
  /*----------------------------------*/
  TPila *crearPila()
  {
     TPila *pila=(TPila*)malloc(sizeof(TPila));
     pila->front=NULL;
     return pila;
  }
  /*----------------------------------*/
  void push(TPila *pila, int x)
  {
     TNodo *nodo=crearNodo(x);
     nodo->sgt=pila->front;
     pila->front=nodo;
  }
  /*----------------------------------*/
  void pop(TPila *pila)
  {

     TNodo *p=NULL;
     if(pila->front!=NULL)
     {
        p=pila->front;
        pila->front=p->sgt;
        free(p);
```

```
   }
}

/*---------------------------------*/
int front(TPila pila)
{
   return pila.front->dato;
}
/*---------------------------------*/
void recorrer(TPila pila)
{
   TNodo *p=pila.front;
   while(p!=NULL)
   {
      printf("%d \n",p->dato);
      p=p->sgt;
   }
}
/*---------------------------------*/
int isEmpty(TPila pila)
{
   int r=0;
   if(pila.front==NULL)
   {
      r=1;
   }
   return r;
}
```

Con la implementación anterior, podemos ejemplificar el uso de una pila mediante la siguiente implementación del *main*:

```
 int main()
{
    TPila *p=crearPila();
    push(p,1);
    push(p,2);
    push(p,3);
    printf("Primer recorrido \n");
    if(!isEmpty(*p))
    {
        recorrer(*p);
    }
    //dejamos la pila p vaci'a
    while(!isEmpty(*p))
    {
        pop(p);
    }
    printf("Segundo recorrido \n");
    recorrer(*p);
```

```
    return 0;
}
```

8.2 Cola

Una cola es un tipo abstracto de dato, implementado comúnmente mediante estructuras dinámicas, en el que los elementos son insertados y sacados por lados opuestos, el frente (front) y el fondo (tail). Así, una cola es una estructura FIFO (First in First out), es decir el primer elemento en ser insertado es el primer elemento en ser sacado. Se le considera una estructura justa.

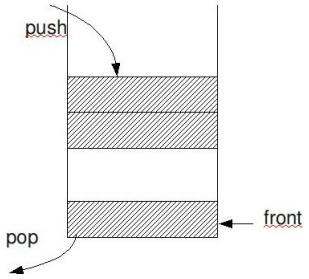

Figure 2: Esquema del TAD Pila

El TAD cola posee las siguiente operaciones:

- **crearCola**: Crea una pila vacía, devuelve un puntero a la cola.
- **push**: Inserta un elemento en el fondo de la cola.
- **pop**: Saca elemento del frente de la cola, esto elimina el elemento.
- **front**: Consulta por el dato almacenado en el frente de la cola. El tipo de dato de retorno dependerá del dato almacenado en cada nodo.
- **isEmpty**: Consulta si la cola está vacía. Devolverá un dato booleano.

8.2.1 Implementación

El TAD cola se puede implementar con una lista enlazada, pero al necesitar operar por dos extremos de la cola (front y tail) la cola necesita dos punteros, uno al nodo del frente y otro al nodo del fondo. Una posible implementación para una cola de enteros se muestra a continuación:

```
 /*Implementacio'n de una cola de enteros*/
#include<stdio.h>
#include<stdlib.h>
/*----------------------------------*/
typedef struct Nodo
{
    int dato;
    struct Nodo *sgt;
}TNodo;
/*----------------------------------*/
```

```
typedef struct Cola
{
    TNodo *front;
    TNodo *tail;
}TCola;
/*----------------------------------*/
TNodo *crearNodo(int x)
{
    TNodo *nodo=(TNodo*)malloc(sizeof(TNodo));
    nodo->dato=x;
    nodo->sgt=NULL;
    return nodo;
}
/*----------------------------------*/
TCola *crearCola()
{
    TCola *cola=(TCola*)malloc(sizeof(TCola));
    cola->front=NULL;
    cola->tail=NULL;
    return cola;
}
/*----------------------------------*/
void push(TCola *cola, int x)
{
    TNodo *nodo=crearNodo(x);

    if(cola->tail==NULL)
    {
        cola->front=nodo;
    }
    else
    {
        cola->tail->sgt=nodo;
    }
    cola->tail=nodo;

}
/*----------------------------------*/
void pop(TCola *cola)
{

    TNodo *p=NULL;
    if(cola->front!=NULL)
    {
        p=cola->front;
        cola->front=p->sgt;
        if(cola->front==NULL)
        {
            cola->tail=NULL;
        }
```

```
        free(p);
    }
}

/*-----------------------------------*/
int front(TCola cola)
{
    return cola.front->dato;
}
/*-----------------------------------*/
void recorrer(TCola cola)
{
    TNodo *p=cola.front;
    while(p!=NULL)
    {
        printf("%d \n",p->dato);
        p=p->sgt;
    }
}
/*-----------------------------------*/
int isEmpty(TCola cola)
{
    int r=0;
    if(cola.front==NULL)
    {
        r=1;
    }
    return r;
}
```

Ahora, un ejemplo del uso de una cola se muestra en la siguiente implementación del *main*:

```
 int main()
{
    TCola *p=crearCola();
    push(p,1);
    push(p,2);
    push(p,3);
    printf("Primer recorrido \n");
    if(!isEmpty(*p))
    {
        recorrer(*p);
    }
    //dejamos la cola p vaci'a
    while(!isEmpty(*p))
    {
        printf("%d\n",front(*p));
        pop(p);
```

```
    }
    printf("Segundo recorrido \n");
    recorrer(*p);
    return 0;
}
```

Costo: Es importante notar que el costo de las operaciones *push* y *pop* son constantes $O(1)$, esto represante una gran ventaja a la hora se ser aplicadas para resolver problemas concretos.

8.3 Aplicaciones con Pilas y Colas

8.3.1 Balanceando Paréntesis

Esta es una de las aplicaciones más sencillas que podamos implementar usando pilas. Balancear paréntesis consiste en que dada una expresión aritmética en el que se usen paréntesis como signos de agrupación, se debe determinar si los paréntesis que abren están en concordancia con lo que cierran. Si esto es así, entonces se dice que los paréntesis están balanaceados.
Ejemplos

1. La expresión: (a+(x*y)) + y *tiene los paréntesis balanceados.*
2. La expresión: (a+z*(3+x)-5))+(4y+4z) *no tiene los paréntesis balanceados.* Aquí, el tercer paréntesis que cierra no tiene su par que abre.
3. La expresión: (x+(4z-t)+5 *no tiene los paréntesis balanceados.* Aquí, falta cerrar el primer paréntesis.

Solución: La solución consiste en mantener una pila en la que se vayan insertando uno a uno los paréntesis que abren **'('**, leyendo la expresión (considerándola como una cadena) de izquierda a derecha. Cada vez que se encuentra un paréntesis que cierra **')'** se debe verificar que al frente de la pila haya uno que abra, si es así entonces sacar de la pila ese paréntesis. En este proceso pueden suceder los siguiente casos:

1. Caso correcto: Si la expresión se leyó completamente y la pila quedo vacía entonces la expresión tiene los paréntesis balanceados.
2. Casos incorrectos:
   (a) Si la expresión se leyó completamente, pero la pila no quedó vacía. Esto indica que faltan paréntesis por cerrar, por lo tanto la expresión NO tiene los paréntesis balanceados.
   (b) Si se leyó un paréntesis que cierra pero la pila está vacía. Esto indica que hay un inesperado paréntesis que cierra, al no haber el correspondiente paréntesis que lo abre. En este caso también los paréntesis NO están balanceados.

Implementación
Escribiremos la estructura de la pila en una archivo *header pila.h* y luego escribiremos el pograma *parentesis.c* con la solución del problema. El programa

debe incluir el header *pila.h*. En este caso implementaremos una pila de caracteres.

Empezamos escribiendo el *header pila.h*, que es idéntico a nuestra programa de pila anterior. Note que solamente debemos cambiar el tipo de dato del nodo.

```
 /*pila.h*/
#include<stdio.h>
#include<stdlib.h>
/*----------------------------------*/
typedef struct Nodo
{
    char dato;
    struct Nodo *sgt;
}TNodo;
/*----------------------------------*/
typedef struct Pila
{
    TNodo *front;
}TPila;
/*----------------------------------*/
TNodo *crearNodo(char x)
{
    TNodo *nodo=(TNodo*)malloc(sizeof(TNodo));
    nodo->dato=x;
    nodo->sgt=NULL;
    return nodo;
}
/*----------------------------------*/
TPila *crearPila()
{
    TPila *pila=(TPila*)malloc(sizeof(TPila));
    pila->front=NULL;
    return pila;
}
/*----------------------------------*/
void push(TPila *pila, char x)
{
    TNodo *nodo=crearNodo(x);
    nodo->sgt=pila->front;
    pila->front=nodo;
}
/*----------------------------------*/
void pop(TPila *pila)
{

    TNodo *p=NULL;
    if(pila->front!=NULL)
    {
        p=pila->front;
        pila->front=p->sgt;
```

```
        free(p);
    }
}

/*-----------------------------------*/
int front(TPila pila)
{
    return pila.front->dato;
}
/*-----------------------------------*/
void recorrer(TPila pila)
{
    TNodo *p=pila.front;
    while(p!=NULL)
    {
        printf("%c \n",p->dato);
        p=p->sgt;
    }
}
/*-----------------------------------*/
int isEmpty(TPila pila)
{
    int r=0;
    if(pila.front==NULL)
    {
        r=1;
    }
    return r;
}
```

El programa *parentesis.c* implementa la función *verificarParentesis* bajo el algoritmo descrito anteriormente. El programa completo se muestra a continuación:

```
 /*parentesis.c*/
#include<string.h>
#include "pila.h"

#define MAX_CAD 100

#define NO_ERROR 0
#define ERROR_1 1
#define ERROR_2 2

typedef char CADENA[MAX_CAD];

/*---------leer una cadena--------------------*/
void readLine(char s[])
{
    fgets(s,MAX_CAD,stdin);
```

```
    s[strlen(s)-1]='\0';
    while(strlen(s)==0)
    {
        fgets(s,MAX_CAD,stdin);
        s[strlen(s)-1]='\0';
    }

}

/*---------verifica--------------------*/
int verificarParentesis(CADENA exp)
{
    int i=0, n=0;
    char  car='\0';
    int ERROR=NO_ERROR;
    n=strlen(exp);
    TPila *pila=crearPila();
    while((i<n)&&(ERROR==NO_ERROR))
    {
       car=exp[i];
       if (car=='(')
       {
           push(pila,car);
       }
       if(car==')')
       {
           if(!isEmpty(*pila))
           {
               pop(pila);
           }
           else
           {
               ERROR=ERROR_2;
           }
       }
       i++;
    }

    if((ERROR==NO_ERROR)&&(!isEmpty(*pila)))
    {
        ERROR=ERROR_1;
    }
    return ERROR;
}

/*----------------main-----------------------*/

int main()
{
```

```
    CADENA expresion;
    int error=NO_ERROR;
    printf("Ingrese una expresion: ");
    readLine(expresion);
    error=verificarParentesis(expresion);

    switch(error)
    {
        case NO_ERROR:
            printf("La expresio'n tiene los par\'entesis balanceados!!\n");
            break;
        case ERROR_1:
            printf("ERROR: Quedaron pare'ntesis por cerrar !!\n");
            break;
        case ERROR_2:
            printf("ERROR: Hay demasiados pare'ntesis que cierran !!\n");
            break;
    }
    return 0;
}
```

8.3.2 Balanceando Operadores de Agrupación

Este problema es similar al problema anterior pero ahora puede usar tres tipos de operadores de agrupación: las llaves ({, }), los corchetes ([,]) y los parénteis ((,)).

En este caso, los operadores de apertura irán a la pila, y cuando encontremos un operador que cierra debemos verificar si en el frente de la pila está su correspondiente operador que abre. Si es así, aplicamo *pop* en la pila, de otro modo se trata de un error. La expresión será correcta si al finalizar de procesar toda la expresión de entrada la pila quedo vaciía.

Igual que en el caso anterior, usaremos una pila de caracteres, por lo tanto reusaremos la implmentación *pila.h*. Así, nos dedicaremos a implementar la función *verificarAgrupadores* en el programa *agrupación.c* que se muestra a continuación.

```
 /*agrupacion.c*/
#include<string.h>
#include "pilaCaracteres.h"

#define MAX_CAD 100

#define NO_ERROR 0
#define ERROR_1 1
#define ERROR_2 2
#define ERROR_3 3

typedef char CADENA[MAX_CAD];
```

```
/*---------leer una cadena--------------------*/
void readLine(char s[])
{
    fgets(s,MAX_CAD,stdin);

    s[strlen(s)-1]='\0';
    while(strlen(s)==0)
    {
        fgets(s,MAX_CAD,stdin);
        s[strlen(s)-1]='\0';
    }

}

/*---------verifica--------------------*/
int verificarAgrupadores(CADENA exp)
{
    int i=0, n=0;
    char  car='\0';
    int ERROR=NO_ERROR;
    n=strlen(exp);
    TPila *pila=crearPila();
    while((i<n)&&(ERROR==NO_ERROR))
    {
       car=exp[i];
       if ((car=='(')||(car=='[')||(car=='{'))
       {
           push(pila,car);
       }
       if ((car==')')||(car==']')||(car=='}'))
       {
          if(isEmpty(*pila))
           {
               ERROR=ERROR_2;//inesperado operador que cierra
           }
           else
           {
               if(((car==')')&&(front(*pila)=='('))||
                  ((car=='}')&&(front(*pila)=='{'))||
                  ((car==']')&&(front(*pila)=='[')))
               {
                   pop(pila);
               }
               else
               {
                   ERROR=ERROR_3;//incompatible operador que cierra
               }

           }
```

```
        }
        i++;
    }

    if((ERROR==NO_ERROR)&&(!isEmpty(*pila)))
    {
        ERROR=ERROR_1;//faltan cerrar
    }
    return ERROR;
}

/*----------------main-------------------------*/

int main()
{
    CADENA expresion;
    int error=NO_ERROR;
    printf("Ingrese una expresion: ");
    readLine(expresion);
    error=verificarAgrupadores(expresion);

    switch(error)
    {
        case NO_ERROR:
            printf("La expresio'n tiene los operadores de agrupaci'on balanceados!!\n");
            break;
        case ERROR_1:
            printf("ERROR: Quedaron operadores por cerrar !!\n");
            break;
        case ERROR_2:
            printf("ERROR: No se espera operadores de cierre !!\n");
            break;
        case ERROR_3:
            printf("ERROR: Incompatible operador que cierra !!\n");
            break;
    }
    return 0;
}
```

8.3.3 Evaluando Expresiones Aritméticas

Una expresión aritmética se compone de operandos y operadores. Existen tres formas clásicas de escribir expresiones aritméticas. Esta son:

1. Notación infija: Los operadores se escriben entre los operandos que afecta. Es importante la prioridad de los operadores.
   Ejemplo:
   $$5 + 4 * 7$$

2. Notación prefija o polaca: Los operadores se escriben antes de los operandos.

Ejemplo:

$$+5 * 47$$

3. Notación postfija o polaca inversa: Los operadores se escriben luego de los operandos.
   Ejemplo:

$$547 * +$$

La notación postfija es una de las formas más fáciles de evaluar expresiones aritméticas, pues no hay necesidad de preocuparse por la prioridad de los operadores. La evaluación se realiza en línea, conforme se ingresan los datos. La notación postfija ha sido ampliamente utilizada por caluladoras científicas.

Para la evaluación de una expresión en notación postfija se requiere un pila en donde se vayan almacenando los operandos, cuando se encuentra un operador, se extrae de la pila la cantidad de operandos que requiere el operador. Se evalúa la operación correspondiente y el resultado se inserta a la pila. Al final, la pila tendrá solamente un operando, será el resultado de la evaluación.

Antes de procesar la expresión, es necesario descomponerla en *tokens* (operandos y operadores) de modo que sean insertadas en una cola.

El algoritmo para evaluar una expresión en notación postfija es:

```
Algoritmo: Evaluar postfija
Input: Q:Cola de tokens
Salida: Resultado

P=crearPila
Mientras (!isEmpty(Q) AND (no haya ERROR))
   t=front(Q)
   pop(Q)
   Si (t es un operando)
      push(Q,t)
   sino //es un operador
         n=operandosNecearios(t)
         Si en P hay menos de n elementos
               ERROR: Hay pocos datos
         sino
              tomar n elementos de  P
              x=evaluar los n datos con el operador
              push(P,x)
         fin_si
   fin_si
fin_mientras

Si en  P queda un elemento
    r=front(P)
sino
    ERROR: Muchos datos
fin_si

devolver r
```

Conversión de infija a postfija
Comúnmente, nos acostumbramos a escribir expresiones en notación infija. Sin embargo, existe un algoritmo para convertirla a su equivalente en notación postfija de modo que su evaluación sea más sencilla. El algoritmo es el siguiente:

```
Algoritmo: Infija a Postfija

Entrada: Una cola (Q) de tokens (en infija).

Salida: Una cola (S) de tokens (en postfija).

Estructura Auxiliar: Una pila (P) de tokens.

P=crearPila
S=crearCola
Mientras (!isEmpty(Q))
    e=front(Q)
    pop(Q)

    Sea el caso e
        caso de e es un operando
             push(S,e)
        caso e es un '('
                 push(P,e)
        caso e es un ')'
             Mientras front(P) != '('
                 x=front(P)
                 pop(P)
                 push(S,x)
             fin_mientras
             pop(P)
        caso e es un operador
             Mientras !isEmpty(P) AND
                      front(P) != '('  AND
                      prioridad(front(P))>=prioridad(e)
                       x=front(P)
                       pop(P)
                       push(S,x)
             fin_mientras
             push(P,e)
     fin_caso
fin_mientras

Mientras !isEmpty(P)
     x=front(P)
     pop(P)
     push(S,x)
fin_mientras

devolver S
```

9 Árboles

Un árbol se define como una colección de nodos organizados en forma recursiva. Cuando hay 0 nodos se dice que el árbol está vacío, en caso contrario el árbol consiste de un nodo denominado raíz, el cual tiene 0 o más referencias a otros árboles, conocidos como subárboles. Por lo tantoo, cada nodo contiene información de interés (llave) y punteros o referencias a los subárboles.

Las raíces de los subárboles se denominan hijos de la raíz, y consecuentemente la raíz se denomina padre de las raíces de sus subárboles. En la Figura 3 se muestra en representación esquemática de un árbol:

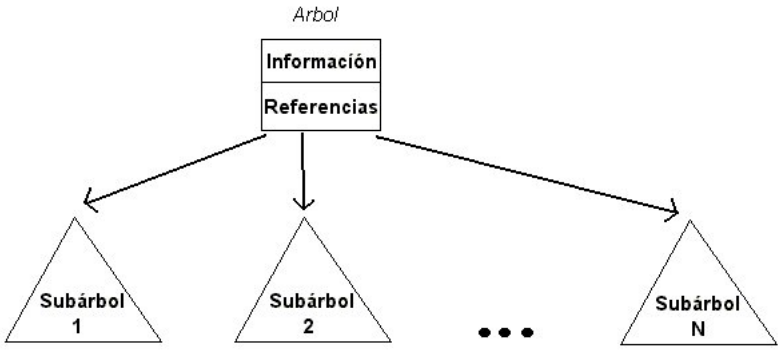

Figure 3: Esquema de un árbol.

Algunas definiciones:

- Nodos hojas: Nodos que no poseen hijos.
- Nodos hermanos: Nodos que tienen el padre en común.
- Camino entre nodos: Un camino entre un nodo x_1 y un nodo x_k está definido como la secuencia de nodos $x_1, x_2, \ldots, x_k$, tal que x_i es padre de x_{i+1}, $1 \leq i < k$. El largo del camino es el número de referencias que componen el camino, que para el ejemplo son $k - 1$. Existe un camino desde cada nodo del árbol a sí mismo y es de largo 0.

  En un árbol existe un único camino desde la raíz hasta cualquier otro nodo del árbol.
- Nodo ancestro: Un nodo x es ancestro de un nodo y si existe un camino desde x a y.
- Nodo descendiente: Un nodo x es descendiente de un nodo y si existe un camino desde y a x.
- Profundidad: Se define la profundidad del nodo x como el largo del camino entre la raíz del árbol y el nodo x. Esto implica que la profundidad de la

raíz es siempre 0. La profundidad de un árbol es la máxima profundidad de sus nodos.

- Altura: La altura de un nodo x es el máximo largo de camino desde x hasta alguna hoja. Esto implica que la altura de toda hoja es 0. La altura de un árbol es igual a la altura de la raíz, y tiene el mismo valor que la profundidad de la hoja más profunda. La altura de un árbol vacío se define como -1.

La Figura 4 muestra un ejemplo de un árbol.

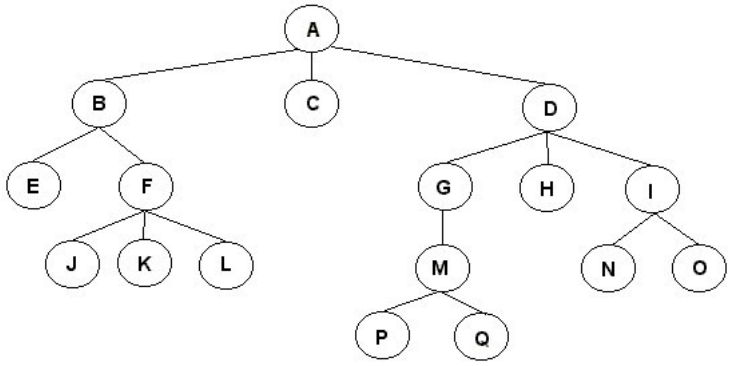

Figure 4: Ejemplo de un árbol.

En el ejemplo de la Figura 4 se tiene:

- A es la raíz del árbol.
- A es padre de B, C y D.
- E y F son hermanos, puesto que ambos son hijos de B.
- E, J, K, L, C, P, Q, H, N y O son las hojas del árbol.
- El camino desde A a J es único, lo conforman los nodos A-B-F-J y es de largo 3.
- D es ancestro de P, y por lo tanto P es descendiente de D.
- L no es descendiente de C, puesto que no existe un camino desde C a L.
- La profundidad de C es 1, de F es 2 y de Q es 4.
- La altura de C es 0, de F es 1 y de D es 3.
- La altura del árbol es 4 (largo del camino entre la raz A y la hoja más profunda, P ó Q).

9.1 Árboles Binarios

Un árbol binario es un árbol en donde cada nodo posee a lo más 2 referencias a subárboles (ni ms, ni menos). En general, dichas referencias se denominan izquierda y derecha, y consecuentemente se define el subárbol izquierdo y subárbol derecho del árbol.

Un esquema de un árbol binario se presenta en la Figura 5.

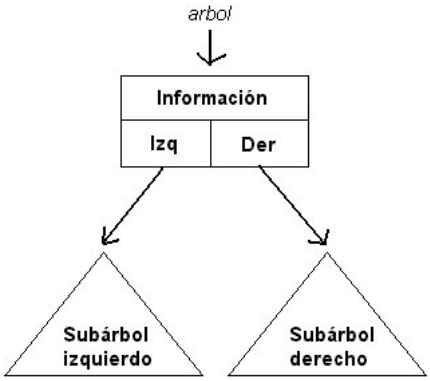

Figure 5: Esquema de un árbol binario.

9.1.1 Implementación

Un árbol binario puede implementarse usando la siguiente estructura, suponiendo que la llave o dato de inteŕes es un entero:

```
typedef struct AB
{
    int llave;
    struct AB *izq;
    struct AB *der;
}NodoAB;

typedef  NodoAB *BinaryTree;
```

Por otro lado, los nodos que conforman un árbol binario se denominan nodos internos, y todas los nodos con referencias que son null se denominan nodos externos (ver Figura 6).

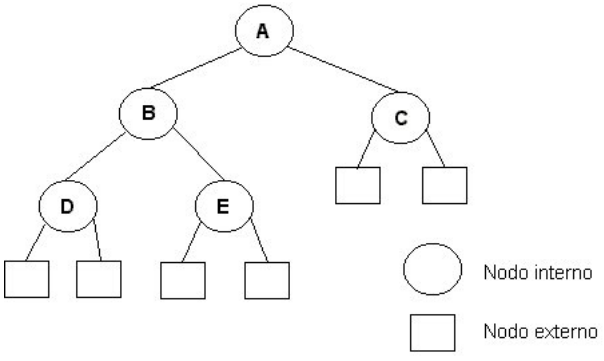

Figure 6: Nodos internos y externos en un árbol binario.

9.1.2 Recorridos en árboles binarios

Existen tres formas principales para recorrer un árbol binario en forma recursiva. Estas son:

- **Preorden**: raíz - subárbol izquierdo - subárbol derecho.
- **Inorden**: subárbol izquierdo - raíz - subárbol derecho.
- **Postorden**: subárbol izquierdo - subárbol derecho - raíz.

Por ejemplo, según el árbol binario de la Figura 7, se tiene los siguiente recorridos:

- **Preorden**: 2-7-2-6-5-11-5-9-4.
- **Inorden**: 2-7-5-6-11-2-5-4-9.
- **Postorden**: 2-5-11-6-7-4-9-5-2.

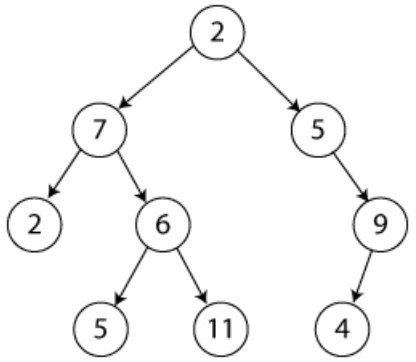

Figure 7: Ejemplo de árbol binario.

9.2 Árboles en General

En un árbol general cada nodo puede poseer un número indeterminado de hijos. La implementación de los nodos en este caso se realiza de la siguiente manera: como no se sabe de antemano cuántos hijos tiene un nodo en particular se utilizan dos referencias, una a su primer hijo y otra a su hermano más cercano. La ríz del árbol necesariamente tiene la referencia nula (null) a su hermano.

La implementación de un árbol sigue la siguiente estructura:

```
 typedef struct AG
{
    int llave;
    struct AG *hijo;
    struct AG *hno;
}NodoAG;

typedef  NodoAG *GeneralTree;
```

9.3 Úso de árboles en búsquedas

9.3.1 Árboles Binarios de Búsqueda (ABB)

9.3.2 Árboles AVL

9.3.3 Árboles Binarios Aleatorizados

9.3.4 Splay Tree