ESCUELA DE PREPARACIÓN DE OPOSITORES
E. P. O.
C/. La Merced, 8 – Bajo A Telf.: 968 24 85 54
30001 MURCIA

INF12 – SAI12

Organización lógica de los datos. Estructuras dinámicas.

Esquema.

1 INTRODUCCIÓN. 2
2 TIPOS DE DATOS RECURSIVOS. 2
3 PUNTEROS 2
4 LISTAS ENLAZADAS 4
4.1 OPERACIONES BÁSICAS. 4
4.1.1 Inserción en listas. 5
4.1.2 Borrado en listas. 6
4.1.3 Recorrido de listas. 6
4.1.4 Búsqueda en listas. 7
4.2 LISTAS ORDENADAS. 7
4.3 LISTAS DOBLEMENTE ENLAZADAS. 9
4.4 PILAS. 9
4.5 COLAS. 9
5 ESTRUCTURAS DE ÁRBOL. 10
5.1 CONCEPTOS Y DEFINICIONES BÁSICAS. 10
5.2 ÁRBOLES BINARIOS. 11
5.2.1 Recorrido del árbol. 12
5.2.2 Inserción en árboles binarios. 13
5.2.3 Eliminación en árboles binarios. 14
5.3 ÁRBOLES MULTICAMINO. 16
5.3.1 Árboles B 17
Búsqueda. 17
Inserción. 18
Eliminación. 18
6 GRAFOS. 19
6.1 CONCEPTOS Y DEFINICIONES BÁSICAS. 19
6.2 REPRESENTACIÓN DE GRAFOS. 21
6.2.1 Representación mediante matrices: matrices de adyacencia. 21
6.2.2 Representación mediante punteros: listas de adyacencia. 22
6.2.3 Representación mediante punteros: matrices dispersas. 23
6.3 RECORRIDO DE GRAFOS 23
6.3.1 Recorrido en anchura o BFS (Breadth First Search). 23
6.3.2 Recorrido en profundidad o DFS (Depth First Search). 23
7 CONCLUSIONES. 23

1 Introducción.

Las estructuras array y registro se llaman fundamentales debido a que constituyen los bloques elementales de los cuales se forman las estructuras más complejas y porque en la práctica ocurren muy frecuentemente. El objeto de definir un tipo de datos y por consiguiente especificar que ciertas variables son de ese tipo, es que la gama de valores tomados por estas variables y por tanto su patrón de almacenamiento, se fija una sola vez y para todos. En consecuencia, se dice que las variables declaradas en esta forma son estáticas.

Sin embargo, hay muchos problemas en los que intervienen estructuras de información mucho más complicadas. La característica de estos problemas es que no sólo los valores, sino también las estructuras de las variables, cambian durante el proceso de cálculo. Por lo tanto, se las llama estructuras dinámicas. Naturalmente, las componentes de estas estructuras son estáticas, es decir, de uno de los tipos de datos fundamentales. Este tema está dedicado a la construcción, análisis y manejo de estructuras de información dinámicas.

Este tema se dedica a la generación y manipulación de estructuras de datos cuyas componentes se vinculan por medio de punteros. Las estructuras con modelos específicos simples se destacan en particular; pueden derivarse fórmulas para manejar estructuras más complejas que aquellas que manipulan formaciones básicas. Estas son la lista enlazada y los árboles. Para los ejemplos presentados usaremos el lenguaje C.

2 Tipos de datos recursivos.

Los valores de este tipo de datos recursivo contendrán una o más componentes que pertenecerán al mismo tipo que éste. Un simple ejemplo de un objeto que se representará adecuadamente como un tipo definido en forma recursiva es la expresión aritmética que se encuentra en los lenguajes de programación. La recursión se utiliza para reflejar la posibilidad de anidamiento. En consecuencia, definiremos una expresión de manera informal como sigue: Una expresión consta de un término, seguido de un operador y después un término. Un término es una variable o bien una expresión entre paréntesis.

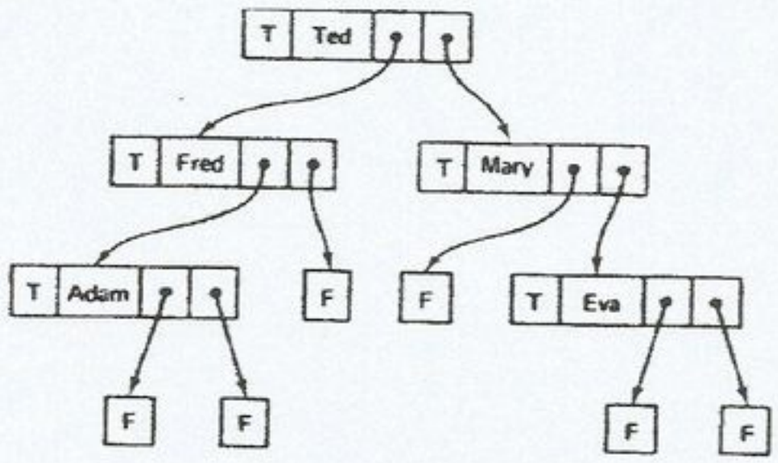
```
typedef struct expression
{
   operator op;
   union term
   {
      alfa id;
      struct expression *subex;
   } opd1, opd2:
};
```

El papel importante de la recursión variable se hace evidente; es el único medio por el cual una estructura de datos recursiva puede ser limitada y por lo tanto es una compañía inevitable de toda definición recursiva.

3 Punteros.

La propiedad característica de las estructuras recursivas que las distingue claramente de las estructuras fundamentales (arrays, registros) es su capacidad de cambiar de tamaño. En consecuencia, es imposible asignar una cantidad fija de espacio en la memoria a una estructura definida recursivamente y como consecuencia un

compilador no puede asociar direcciones específicas a las componentes de estas variables. La técnica que se utiliza de forma más común para este problema implica la asignación dinámica de almacenamiento, es decir, durante la ejecución de un programa. El compilador asigna después una cantidad fija de almacenamiento para contener la dirección de la componente asignada dinámicamente y no de la componente misma. Por ejemplo, el árbol genealógico que se ilustra en la siguiente figura será representado por registros individuales, uno por cada persona.



Gráficamente, esta situación se expresa de la mejor manera mediante el uso de flechas o punteros.

Es posible definir estructuras potencialmente infinitas o circulares y dictaminar que ciertas estructuras se comparten. En consecuencia se ha vuelto común en lenguajes de programación avanzados hacer posible la manipulación explícita de referencias de datos además de los datos mismos. Esto implica que debe existir una distinción notacional clara entre los datos y las referencias de éstos y que en consecuencia deben introducirse tipos de datos cuyos valores sean punteros de otros datos. La notación que se emplea con este objeto es la siguiente:

```
typedef t *t0;
```

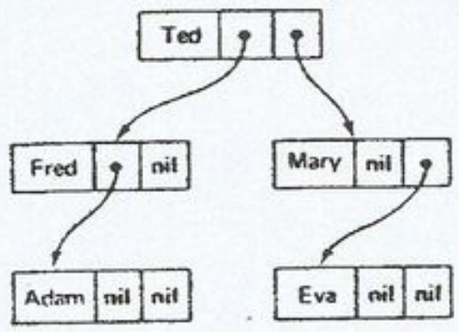
Los valores del tipo T son punteros de datos del tipo T_0. Los valores de los tipos punteros se generan siempre que un elemento de datos es asignado dinámicamente. Con este fin, se presenta un procedimiento *malloc*. Dada una variable puntero de tipo T, la proposición $p=(t_0 *)malloc(sizeof(t_0))$ asigna efectivamente una variable de tipo T_0 y asigna el puntero que hace referencia de esta nueva variable a p. El valor del puntero mismo puede referirse ahora como p (es decir, como el valor de la variable puntero p). En cambio, la variable que es referida por p se representa por $*p$.

Se mencionó antes que una componente variante es indispensable en todo tipo recursivo para asegurar la cardinalidad finita. El ejemplo del árbol genealógico familiar es de un modelo que exhibe una constelación que ocurre con mayor frecuencia, es decir, el caso en el cual el campo identificador tiene dos valores y en el cual su valor con calidad de falso implica la ausencia de otras componentes.

La técnica de instrumentación que utiliza punteros sugiere una forma sencilla de ahorrar espacio para almacenamiento, haciendo que la información identificadora se incluya en el valor del puntero mismo. La solución común consiste en ampliar el intervalo de valores de todos los tipos punteros en un solo valor que no apunte a ningún elemento en absoluto. Este valor se representa por el símbolo especial NULL, y se entiende que NULL es automáticamente un elemento de todos los tipos punteros declarados. Esta extensión del intervalo de valores del puntero explica por qué pueden generarse estructuras finitas sin la presencia explícita de variantes en su declaración.

Para el ejemplo del árbol genealógico, la declaración quedaría como:

```
typedef struct Person
{
   alfa name;
   struct Person *father, *mother;
};
```



La estructura de datos que representa el árbol genealógico se vuelve a mostrar en la siguiente figura; en él los punteros de personas desconocidas se simbolizan por NULL. La mejora resultante en economía de almacenamiento es obvia.

Supongamos que Fred y Mary son hermanos, es decir, tienen el mismo padre y madre. Esta situación se expresa fácilmente reemplazando los dos valores NULL en los campos respectivos de los dos registros.

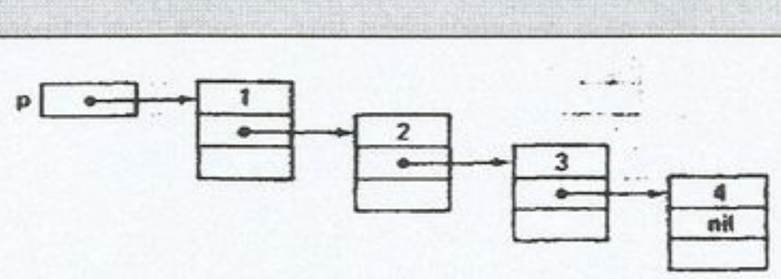
Una consecuencia adicional de la claridad de los punteros es que es posible definir y manipular estructuras de datos cíclicas. Esta flexibilidad adicional origina, desde luego, no sólo potencias crecientes sino también requiere mayor cuidado por parte del programador, dado que la manipulación de estructuras de datos cíclicas nos puede llevar fácilmente a procesos sin terminación.

4 Listas enlazadas.

4.1 Operaciones básicas.

La manera más simple de vincular un conjunto de elementos consiste en alinearlos en una sola lista o fila. En este caso, sólo se necesita un vínculo por cada elemento para hacer referencia a su sucesor. Supongamos que un tipo *Node* se define como sigue:

```
typedef struct s_node
{
   int key;
   struct s_node *next;
   ...
} Node;
```



Toda variable de este tipo consta de tres componentes, es decir, una clave (*key*) identificadora, el puntero de su sucesor (*next*) y posiblemente otra información asociada que se omite. Una lista de nodos, con un puntero a su primera componente asignada a una variable *p*, se ilustra en la figura.

4.1.1 Inserción en listas.

Quizá la operación más simple que se realiza con una lista como se muestra en la figura es la inserción de un elemento en su cabeza o parte superior. Primero, se asigna un elemento de tipo *Node* y su puntero se asigna a una variable puntero auxiliar *q*. Según esto una simple reasignación de punteros completa la operación, que se programa como:

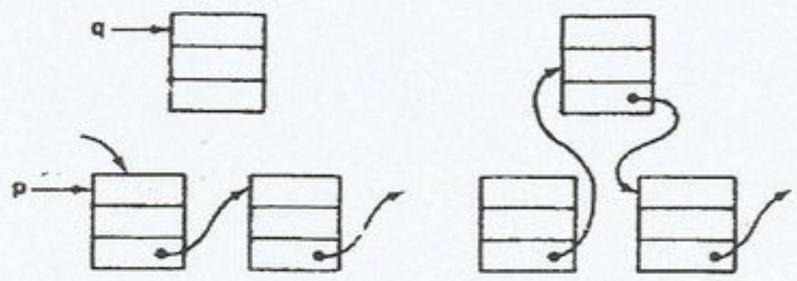
```
p = NULL; (*empezar con lista vacía*)
while (n>0)
{
   if((q=(Node *)malloc(sizeof(Node)))==NULL)
   { /*ERROR AL ASIGNAR MEMORIA*/
      ...
   }
   q->next = p;
   p = q;
   q->key = n;
   n--;
}
```

La operación de inserción de un elemento en la parte superior de una lista sugiere inmediatamente la forma en que una lista de este tipo puede ser generada: comenzando con la lista vacía, se añade varias veces. Esta es la forma más simple de formar una lista. Sin embargo, el orden resultante de los elementos es el recíproco del orden de su inserción. En algunas aplicaciones esto es indeseable y en consecuencia, deben agregarse nuevos elementos al final en vez de en la parte superior de la lista. Aunque el final se puede determinar fácilmente por medio de un examen de la lista, este punto de vista ingenuo implica un esfuerzo que bien puede evitarse utilizando un segundo puntero que siempre designe el último elemento. Su desventaja es que el primer elemento insertado tiene que ser tratado de forma diferente de los últimos.

La disponibilidad explícita de los punteros hace muy simples ciertas operaciones que en otras circunstancias son engorrosas: entre las operaciones de lista elementales se encuentran las de inserción y borrado de elementos y, desde luego, la revisión o recorrido de una lista. Primero investigaremos la inserción en lista.

Supongamos que un elemento designado por un puntero *q* se insertará en una lista después del elemento designado por el puntero *p*. Las asignaciones de punteros que se necesitan se expresan a continuación y su efecto se exhibe en la siguiente figura.

```
q->next = p->next;
```

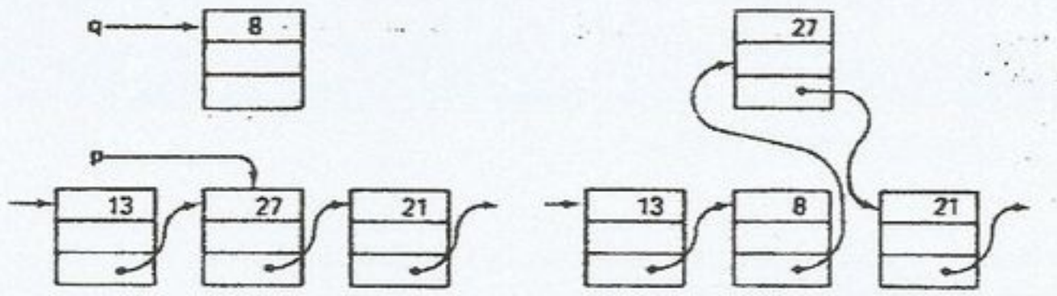



```
p->next = q;
```

Si se desea la inserción antes de en vez de después de *p*, la cadena de unión unidireccional parece ocasionar un problema, ya que no ofrece ninguna clase de trayectoria hacia los predecesores de un elemento. Sin embargo, un simple truco

resuelve nuestro dilema: éste se expresa a continuación y se ilustra en la siguiente figura.

```
tmp = q->key;
q->key = p->key;
```

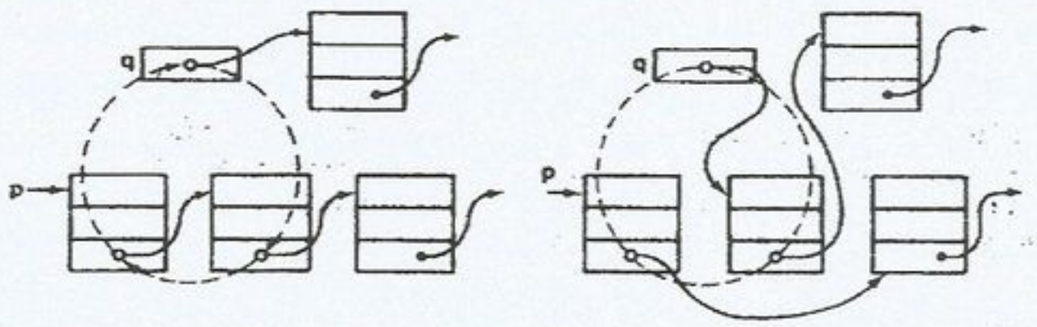



```
q->next = p->next;
p->next = q;
p->key = tmp;
```

4.1.2 Borrado en listas.

La eliminación del sucesor de un elemento *p* es directo. A continuación se muestra esto en combinación con la reinserción del elemento borrado que está en la parte superior de otra lista (designada por *q*). La figura ilustra el efecto de las proposiciones y muestra que se trata de un intercambio cíclico de tres punteros.

```
r = p->next;
p->next = r->next;
r->next = q;
q = r;
```



La eliminación de un elemento designado (en vez de su sucesor) es más difícil, ya que se encuentra el mismo problema que con la inserción: retornar al predecesor del elemento denotado es imposible. Pero la eliminación del sucesor después de adelantar su valor es una solución relativamente obvia y simple. Puede aplicarse siempre que *p* tenga un sucesor, es decir, no sea el último elemento en la lista.

4.1.3 Recorrido de listas.

Supongamos que tiene que realizarse una operación P(x) por todos y cada uno de los elementos de la lista cuyo primer elemento es apuntado por *p*. Esta tarea se puede expresar como sigue:

```
while (p!=NULL)
{
   P(p);
   p = p->next
}
```

4.1.4 Búsqueda en listas.

Una operación muy frecuente que se efectúa es la búsqueda en lista de un elemento con una clave dada *x*. A diferencia de los arrays, el proceso de búsqueda debe ser aquí puramente secuencial. La búsqueda termina si se halla un elemento o bien si se llega al final de la lista. Esto lo refleja una conjunción lógica que consta de dos términos. Una vez más, se supone que la parte superior de la lista se designa por medio de un puntero *p*.

```
while (p != NULL) && (p->key != x)
   p = p->next;
```

p=NULL implica que **p* no existe y, en consecuencia, que la expresión *p->key!=x* es indefinida. El orden de los dos términos es esencial por lo antes dicho.

4.2 Listas ordenadas.

Para dar un ejemplo, consideremos ahora un problema que ocurrirá en todo este tema a fin de ilustrar soluciones y técnicas alternativas. Se trata del problema de la lectura de un texto, colectando todas sus palabras y contando la frecuencia de su ocurrencia. A esto se le denomina construcción de una concordancia o bien generación de una lista de referencia cruzada.

Una solución obvia consiste en construir una lista de palabras halladas en el texto. La lista se examina para encontrar cada palabra. Si se encuentra la palabra, su contador de frecuencia se incrementa; en caso contrario la palabra se agrega a la lista.

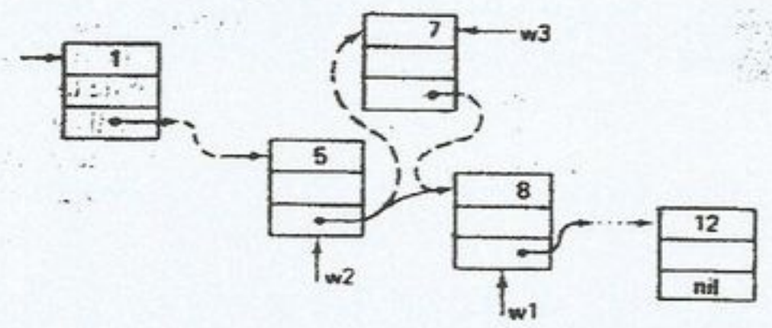
La variable *root* se refiere a la parte superior de la lista en la cual se insertan nuevas palabras.

```
Node *search(int x, Node *root)
{
   Node *w;

   w = root;
   while (w != NULL) && (w->key != x)
      w = w->next;
   if (w == NULL) /*nueva entrada*/
   {
      w = root;
      if((root=(Node *)malloc(sizeof(Node)))==NULL)
      { /*ERROR AL ASIGNAR MEMORIA*/
         ...
      }
      root->key = x;
      root->count = 1;
      root->next = w;
   }
   else
      w->count++;
   return root;
}
```

Sin embargo, una mejora sencilla se tiene fácilmente a la mano: la búsqueda de la lista ordenada. Si la lista está ordenada (por ejemplo, por claves crecientes), entonces la búsqueda puede terminarse hasta que se encuentre la primera llave que sea mayor que la nueva. El ordenamiento de una lista se logra insertando nuevos elementos en el sitio adecuado en vez de en la parte superior.

La búsqueda en lista ordenada es un ejemplo común de la situación en la cual debe insertarse un elemento delante de un elemento dado, es decir, en frente del primero cuya clave es demasiado grande. Sin embargo, la técnica que se muestra aquí difiere de las anteriores. En vez de calcar valores, dos punteros se llevan a todo lo largo del recorrido de la lista; *w2* se rezaga un paso antes que *w1* y, por lo tanto, identifica el sitio de inserción adecuado cuando *w1* ha hallado una clave demasiado grande. La etapa de inserción general se muestra en la figura.



El puntero del nuevo elemento (*w3*) se asignará a *w2->next*, excepto cuando la lista siga estando vacía. Por razones de simplicidad y claridad, preferimos evitar esta distinción utilizando una proposición condicional. La única manera de evitar esto consiste en introducir un elemento ficticio en la parte superior de la lista. La proposición de inicialización *root=NULL* se sustituye por

```
if((root=(Node *)malloc(sizeof(Node)))==NULL)
{ /*ERROR AL ASIGNAR MEMORIA*/
   ...
}
root->next=NULL;
```

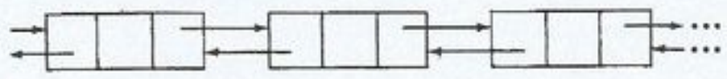
Haciendo referencia de la figura, se determina la condición en la cual el examen continúa para proceder hacia el siguiente elemento; consta de dos factores, es decir, *(w1 != NULL) & (w1->key < x)*. El procedimiento de búsqueda resultante es:

```
void search(int x, Node *root)
{
   Node *w1, *w2, *w3;

   w2 = root;
   w1 = w2->next;
   while (w1 != NULL) && (w1->key < x)
   {
      w2 = w1;
      w1 = w2->next;
   }
   if (w1 == NULL || w1->key > x)
   { /*nueva entrada*/
      if((w3=(Node *)malloc(sizeof(Node)))==NULL)
      { /*ERROR AL ASIGNAR MEMORIA*/
         ...
      }
      w2->next = w3;
      w3->key = x;
      w3->count = 1;
      w3->next = w1;
   }
   else
      w1->count++;
}
```

A fin de acelerar la búsqueda, la condición de continuación de la proposición while puede simplificarse una vez más utilizando un centinela. Esto requiere la presencia inicial de una cabeza ficticia así como de un centinela en la parte final.

4.3 Listas doblemente enlazadas.



En algunas aplicaciones puede ser deseable recorrer eficientemente una lista, tanto hacia adelante como hacia atrás. O, dado un elemento, podría desearse determinar con rapidez el siguiente y el anterior. En tales situaciones, quizá se quisiera poner en cada nodo de una lista un puntero al siguiente nodo y otro al anterior, como se sugiere en la lista doblemente enlazada de la figura.

4.4 Pilas.

Una Pila es un tipo especial de lista en la que todas las inserciones y supresiones tienen lugar en un extremo denominado tope. A las pilas se les llama también listas LIFO (*last in first out*) o listas último en entrar, primero en salir. El modelo intuitivo de una pila es precisamente una pila de fichas de póquer puesta sobre una mesa, o de libros sobre el piso, o de platos en una estantería, situaciones todas en las que sólo es conveniente quitar o agregar un objeto del extremo superior de la pila, al que se denomina tope. Un tipo de datos abstracto de la familia pila incluye a menudo las cinco operaciones siguientes:

1. TOPE(P). Devuelve el valor del elemento de la parte superior de la pila P.
2. SACA(P), en inglés POP. Suprime el elemento superior de la pila. Algunas veces resulta conveniente implementarla como una función que devuelve el elemento que acaba de suprimir.
3. 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.

Una aplicación importante de las pilas se da en la aplicación de procedimientos recursivos en los lenguajes de programación. La organización a tiempo de ejecución de uno de tales lenguajes es el conjunto de estructuras de datos usadas para representar los valores de las variables de un programa durante su ejecución. Todo lenguaje que permita procedimientos recursivos, utiliza una pila de registros de activación para registrar los valores de todas las variables que pertenecen a cada procedimiento activo de un programa. Cuando se llama a un procedimiento P, se coloca en la pila un nuevo registro de activación para P, con independencia de si ya existe en ella otro registro de activación para ese mismo procedimiento. Cuando P vuelve, su registro de activación debe estar en el tope de la pila, puesto que P no puede volver si no lo han hecho previamente todos los procedimientos a los que P ha llamado. Así, se puede sacar de la pila el registro de activación correspondiente a la llamada actual de P y hacer que el control regrese al punto en el que P fue llamado (este punto, conocido como dirección de retorno, se colocó en el registro de activación de P al llamar a este procedimiento).

4.5 Colas.

Una cola es otro tipo especial de lista en el cual los elementos se insertan en un extremo (el posterior) y se suprimen en el otro (el anterior o frente). Las colas se

conocen también como listas FIFO (*first-in first-out*) o listas primero en entrar, primero en salir. Las operaciones para una cola son análogas a las de las pilas; las diferencias sustanciales consisten en que las inserciones se hacen al final de la lista, y no al principio, y en que la terminología tradicional para colas y listas no es la misma. Se usarán las siguientes operaciones con colas:

1. FRENTE(C). Es una función que devuelve el valor del primer elemento de la cola C.
2. PONE_EN_COLA(x, C). Inserta el elemento x al final de la cola C.
3. QUITA_DE_COLA(C). Suprime el primer elemento de C.

5. VACIA(C). Devuelve verdadero si, y sólo si, C es una cola vacía.

Cualquier realización, de listas es lícita para las colas. No obstante, para aumentar la eficacia de PONE_EN_COLA es posible aprovechar el hecho de que las inserciones se efectúan sólo en el extremo posterior. En lugar de recorrer la lista de principio a fin cada vez que se desea hacer una inserción, se puede mantener un puntero al último elemento. Como en las listas de cualquier clase, también se mantiene un puntero al frente de la lista; en las colas, ese apuntador es útil para ejecutar mandatos del tipo FRENTE o QUITA_DE_COLA.

5 Estructuras de árbol.

5.1 Conceptos y definiciones básicas.

Una estructura de árbol con tipo base T es

1. La estructura vacía.
2. Un nodo de tipo T con un número finito de estructuras de árbol disjuntas asociadas de tipo de base T, llamadas subárboles.

Una lista es una estructura de árbol en la cual cada nodo tiene como mucho un subárbol. La lista se llama por tanto también árbol degenerado.

Un árbol ordenado es un árbol en el cual las ramas de cada nodo están ordenadas. Un nodo y que está directamente debajo del nodo x se denomina descendiente (directo) de x; si x está en el nivel i, entonces se dice que y es un nivel i+1. A la inversa, se dice que el nodo x es el ancestro (directo) de y. La raíz de un árbol se define como localizada en el nivel 0. Se dice que el nivel máximo de cualquier elemento de un árbol es su profundidad o altura.

Si un elemento no tiene descendientes, se le denomina nodo terminal o bien hoja, y un elemento que no es terminal es un nodo interior. El número de descendientes (directos) de un nodo interior se conoce como su grado. El grado máximo en todos los nodos es el grado del árbol.

El número máximo de nodos en un árbol de una altura dada h se alcanza si todos los nodos tienen d subárboles, excepto los situados en el nivel h, que no tienen ninguno. Para un árbol de grado d, el nivel 0 contiene por tanto 1 nodo (es decir, la raíz), el nivel 1 contiene sus d descendientes, el nivel 2 contiene los d^2 descendientes de los d nodos en el nivel 2, etc.

De particular importancia son los árboles ordenados de grado 2. A éstos se les llama árboles binarios. Un árbol binario ordenado se define como un conjunto finito de elementos (nodos) que es vacío o bien consta de una raíz (nodo) con dos árboles

binarios disjuntos llamados subárbol izquierdo y derecho de la raíz. En las secciones que siguen trataremos exclusivamente con árboles binarios y por lo tanto se utilizará la palabra árbol para referirnos a un árbol binario ordenado. Los árboles con grado mayor que 2 se denominan árboles multicamino y se estudian posteriormente en el tema.

5.2 Árboles binarios.

Es fácil que la ilustración de dichas estructuras recursivas en términos de estructuras de ramificación sugiera inmediatamente el uso de punteros. Evidentemente no tiene uso la declaración de variables con una estructura de árbol fija; en cambio, se definen los nodos como variables con una estructura fija, es decir, de un tipo fijo, en la cual el grado del árbol determina el número de componentes punteros que se refieren a los subárboles del nodo. Evidentemente, la referencia del árbol vacío se simboliza por NULL.

```
typedef struct s_Node
{
   int key;
   struct s_Node *left, *right;
} Node;
```

Supongamos que se generará un árbol con los valores de los nodos siendo n números que se leen de un archivo de entrada. A fin de hacer más desafiante el problema, sea que la tarea consista en la construcción de un árbol con n nodos y altura mínima. A fin de obtener una altura mínima para un número de nodos dado, se debe asignar el número máximo posible de nodos en todos los niveles excepto el más inferior. Esto puede lograrse con claridad distribuyendo los nodos entrantes de igual manera a la izquierda y derecha en cada nodo.

La regla de igual distribución con un número conocido n de nodos se forma mejor en forma recursiva:

1. Utilizar un nodo para la raíz.
2. Generar el subárbol de la izquierda con nl = n DIV 2 nodos en esta forma.
3. Generar el subárbol de la derecha con nr = n – nl - 1 nodos en esta forma.

La regla se expresa como un procedimiento recursivo. Anotamos la siguiente definición: Un árbol es perfectamente balanceado, si para cada nodo los números de nodos en sus subárboles izquierdo y derecho difieren como mucho en 1.

```
Node *tree(int n) /*construir un árbol perfectamente balanceado con n
nodos*/
{
   Node *newnode;
   int x, nl, nr;

   if (n == 0)
      newnode = NULL
   else
   {
      nl = n / 2;
      nr = n-nl-1;
      if((newnode=(Node *)malloc(sizeof(Node)))==NULL)
      { /*ERROR AL ASIGNAR MEMORIA*/
         ...
      }
      newnode->key = x;
```

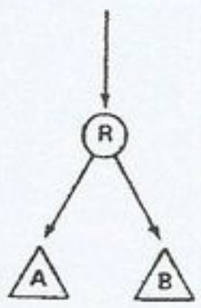
```
        newnode->left = tree(nl);
        newnode->right = tree(nr)
    }
    return newnode;
}
```

Es obvio que los algoritmos recursivos son particularmente adecuados cuando un programa debe manipular información cuya estructura se define por sí mismas de forma recursiva.

5.2.1 Recorrido del árbol.

Hay muchas tareas que pueden realizarse con una estructura de árbol; una común es la de ejecutar una operación P dada con cada elemento del árbol. Así pues, se entiende que P es un parámetro de la tarea más general de visitar todos los nodos o bien, como se llama generalmente, el recorrido del árbol. Si la tarea se considera como un solo proceso secuencial, entonces los nodos individuales son visitados en algún orden específico y pueden considerarse como colocados en una disposición lineal. De hecho, la descripción de muchos algoritmos se facilita considerablemente si se puede hablar acerca del procedimiento del siguiente elemento en el árbol con base en un orden subyacente. Existen tres ordenamientos principales que surgen en forma natural de las estructuras de los árboles. Al igual que la estructura del árbol, se expresan adecuadamente en términos recursivos. Refiriéndose al árbol binario de la figura en la cual R representa la raíz y A y B simbolizan los subárboles izquierdo y derecho, los tres ordenamientos son:

1. Preorden: R, A, B (visitar el nodo antes de los subárboles)
2. En orden: A, R, B
3. Postorden: A, B, R (visitar la raíz después de los subárboles)



Ahora formularemos los tres métodos de recorrido por medio de tres programas concretos con el parámetro explícito t que simboliza el árbol en que se operará y con el parámetro implícito P que representa la operación que se efectuará con cada nodo.

```
void preorder(Node *t)
{
   if (t != NULL)
   {
      P(t);
      preorder(t->left);
      preorder(t->right);
   }
}
```

```
void inorder(Node *t)
{
   if (t != NULL)
   {
      inorder(t->left);
      P(t);
      inorder(t->right);
   }
}
```

```
void postorder(Node *t)
{
   if (t != NULL)
   {
      postorder(t->left);
      postorder(t->right);
      P(t);
   }
}
```

Los árboles binarios se usan para representar un conjunto de datos cuyos elementos se recuperarán a través de una clave única. Si un árbol se organiza en tal forma que para cada nodo t_i todas las claves en el subárbol izquierdo de t_i sean menores que la clave de t_i y aquellas en el subárbol derecho sean mayores que la clave de t_i, entonces a este árbol se le llama árbol de búsqueda. En un árbol de búsqueda es posible localizar una clave arbitraria comenzando en la raíz y prosiguiendo a lo largo de una trayectoria de búsqueda cambiando a un subárbol izquierdo o derecho de un nodo por medio de una decisión basada en la inspección de esa clave del nodo solamente. Como se ha observado, n elementos pueden organizarse en un árbol binario de una altura tan pequeña como *log* n. Por lo tanto, una búsqueda entre n elementos puede realizarse con tan pocas como *log* n comparaciones si el árbol está perfectamente balanceado. Como esta búsqueda sigue una sola trayectoria de la raíz al nodo deseado, puede ser programada fácilmente mediante iteración.

```
Node *locate(int x; Node *t)
{
   while (t != NULL && t->key != x)
      if (t->key < x)
         t = t->right;
      else
         t = t->left;
   return t;
}
```

La función *locate(x, t)* produce el valor NULL, si no se halla ninguna llave con el valor x en el árbol con raíz t.

5.2.2 Inserción en árboles binarios.

Consideremos el caso de un árbol que crece de forma constante pero que nunca se contrae. Un ejemplo común es el problema de concordancia que ya se investigó en relación con las listas enlazadas. Comenzando con un árbol vacío, cada palabra se busca en el árbol. Si se encuentra, su contador de ocurrencias se incrementa; en caso contrario, se inserta como una nueva palabra. Se supone las siguiente definición de tipos de datos:

```
typedef struct s_Word
{
   int key, count;
   struct s_Word *left, *right;
} Word;
```

Toda la operación se muestra a continuación.

```
Word *search(int x, Word *p)
{
   if (p == NULL)
   {
      if((p=(Word *)malloc(sizeof(Word)))==NULL)
      { /*ERROR AL ASIGNAR MEMORIA*/
         ...
      }
      p->key = x;
      p->count = 1;
      p->left = NULL;
      p->right = NULL;
   }
   else
      if (x < p->key)
         p->left = search(x, p->left);
      else
         if (x > p->key)
            p->right = search(x, p->right);
         else
            p->count++;
   return p;
}
```

El proceso de búsqueda se formula como un procedimiento recursivo.

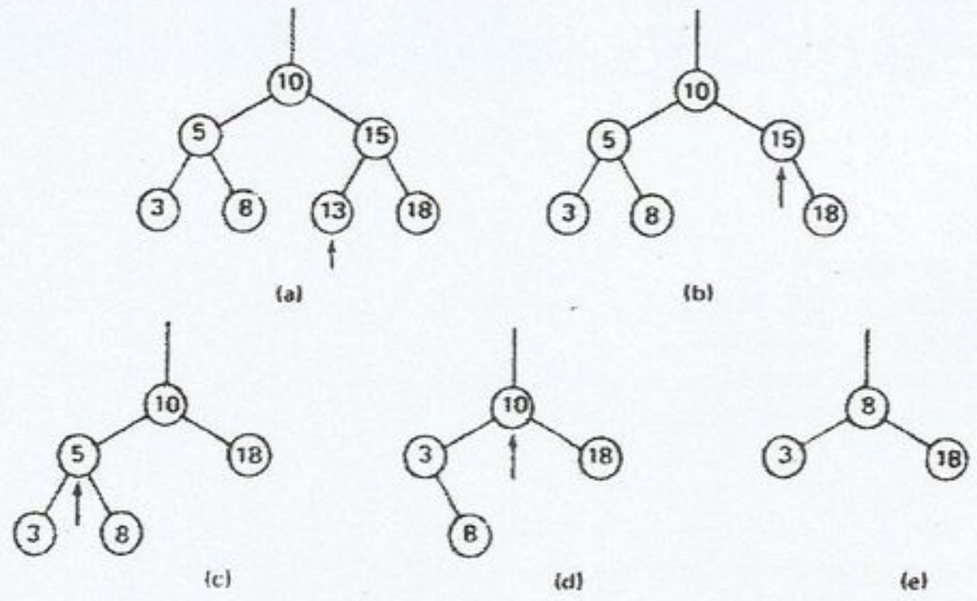
5.2.3 Eliminación en árboles binarios.

La eliminación de un elemento en general no es tan simple como la inserción. Es directo si el elemento a eliminar es un nodo terminal o uno con un solo descendiente. La dificultad radica en la eliminación de un elemento con dos descendientes. En esta situación, el elemento eliminado será sustituido por el elemento de más a la derecha de su subárbol izquierdo o bien por el nodo de más a la izquierda de su subárbol derecho, los cuales tienen a lo más un descendiente. Los detalles se muestran en el procedimiento recursivo llamado delete. Este procedimiento distingue tres casos:

1. No hay una componente con clave igual a x.
2. La componente con la clave x tiene a lo más un descendiente.
3. La componente con la clave x tiene dos descendientes.

```
Word *delete(int x, Word *p)
{
   if (p != NULL)
   {
      if (x < p->key)
         p->left = delete(x, p->left);
      else if (x > p->key)
         p->right = delete(x, p->right);
      else /*Hay que eliminar p*/
      {
         Word *q;
         q = p;
         if (q->right == NULL) /*Caso 2*/
            p = q->left;
         else if (q->left == NULL) /*Caso 2*/
            p = q->right;
         else /*Caso 3*/
         {
            Word *r, *ant;
            r = q->left;
            ant = q;
            while (r->right != NULL)
            {
               ant = r;
               r = r->right;
            }
            q->key = r->key;
            q->count = r->count;
            if (ant == q)
               ant->left = r->left;
            else
               ant->right = r->left;
            q = r; /*Es el nodo que realmente se elimina*/
         }
         free(q);
      }
   }
   return p;
}
```

En el caso 3 se desciende a lo largo de la rama de más a la derecha del subárbol izquierdo del elemento *q* que se eliminará y luego sustituye la información relevante (clave y contador) en el nodo apuntado por *q* por los valores correspondientes de la componente de más a la derecha *r* de ese subárbol izquierdo, siendo precisamente el nodo apuntado por *r* el eliminado.



A fin de ilustrar el funcionamiento del procedimiento, nos referimos a la figura. Consideremos el árbol (a); después eliminamos sucesivamente los nodos con las claves 13, 15, 5, 10. Los árboles resultantes se muestran en la figura.

5.3 Árboles multicamino.

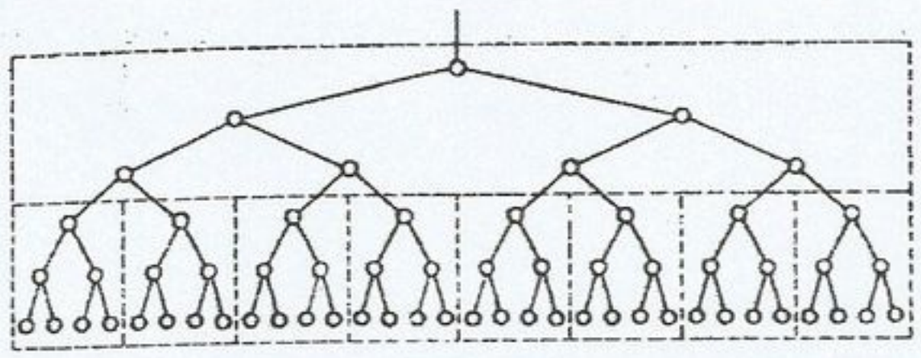
Hasta ahora nos hemos limitado a los árboles binarios. Los árboles con grado mayor que 2 se denominan árboles multicamino. Una definición posible de tipo para estos árboles aparece en la figura.

```
typedef struct Person
{
   char *name;
   struct Person *sibling, *offspring;
}
```

Los algoritmos que operan sobre tales estructuras están íntimamente ligados a sus definiciones de datos, por lo que no tiene sentido especificar reglas generales ni técnicas de aplicación amplia.

Sin embargo, hay un área muy práctica de aplicación de los árboles multicamino que tiene un interés general. Se trata de la construcción y mantenimiento de árboles de búsqueda a gran escala, en los cuales se necesitan las inserciones y eliminaciones, pero la memoria primaria de una computadora no es lo suficientemente grande o económica para utilizarse para el almacenamiento a largo plazo.

Así pues, supongamos que los nodos de un árbol deben ser guardados en un medio de memoria secundaria, digamos en un disco. La principal innovación consiste en que los punteros están representados por direcciones de almacenamiento en disco. Usar un árbol binario para un conjunto de datos de, digamos, un millón de elementos requiere en promedio aproximadamente $log\ 10^6$ pasos de búsqueda. Puesto que cada paso requiere un acceso al disco (con el tiempo inherente de latencia), una organización de memoria que emplee menos accesos será conveniente en extremo. El árbol multicamino es una solución perfecta de este problema. Si se accede a un elemento situado en una memoria secundaria, se puede acceder a un grupo entero de elementos sin mucho costo adicional. Ello significa que un árbol puede subdividirse en subárboles y que éstos se representan como unidades a las cuales se accede simultáneamente. Llamaremos páginas a esos subárboles. La figura muestra un árbol binario subdividido en páginas, cada una formada por 7 nodos.

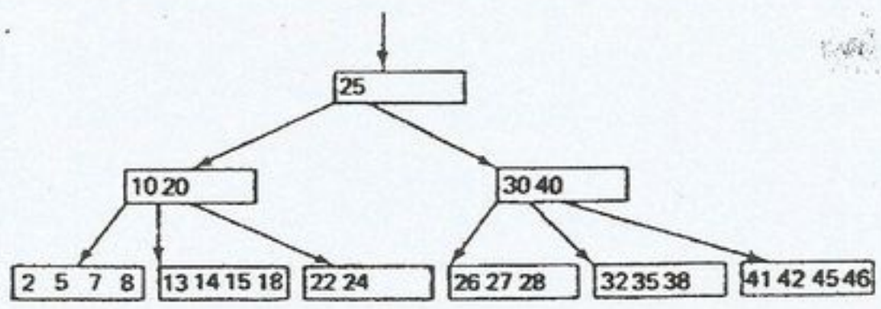


Puede ser considerable el ahorro en el número de accesos al disco: cada acceso de página requiere ahora un acceso al disco. Supongamos que decidimos colocar 100 nodos en una página; entonces el árbol de búsqueda de un millón de elementos requerirá, en promedio, apenas $log_{100}\ 10^6$ accesos de página. Pero, por supuesto, si se deja al árbol crecer aleatoriamente, el peor caso será todavía hasta de 10^4. Es evidente que un plan del crecimiento controlado resulta casi obligatorio en el caso de árboles multicamino.

5.3.1 Árboles B.

Los árboles B tienen las siguientes características:

1. Cada página contiene a lo sumo 2n elementos (claves).
2. Cada página, excepto la de la raíz, contiene n elementos por lo menos.
3. Cada página es una página de hoja, o sea que no tiene descendientes o tiene m+1 descendientes, donde m es su número de claves en esta página.
4. Todas las páginas de hoja aparecen al mismo nivel.



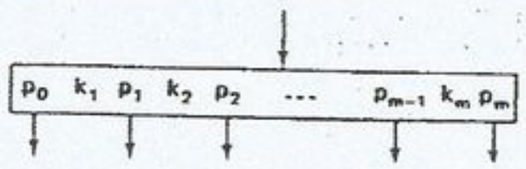
La figura muestra un árbol B de orden 2 con 3 niveles. Todas las páginas contienen 2, 3 o 4 elementos; la excepción es la raíz que puede contener un solo elemento únicamente. Todas las páginas de hoja aparecen en el nivel 3. Las claves aparecen en orden creciente de la izquierda a la derecha si el árbol B es introducido forzadamente en un solo nivel, insertando los descendientes entre las claves de su página madre. Este arreglo representa una extensión natural de los árboles binarios y determina el método con que se busca un elemento que tiene una clave determinada.

Búsqueda.

Examinemos una página como la de la figura y un argumento de búsqueda x. Suponiendo que la página ha sido metida en la memoria primaria, podemos aplicar los métodos ordinarios de búsqueda entre las claves $k_1 ... k_m$. Si m es suficientemente grande, puede recurrirse a la investigación binaria; si es bastante pequeña, bastará con una búsqueda secuencial ordinaria. Si la búsqueda fracasa, nos encontraremos en una de las siguientes situaciones:

1. $k_j < x < k_{i+1}$, para $1 \leq i < m$. Proseguimos la búsqueda en la página apuntada por p_i.

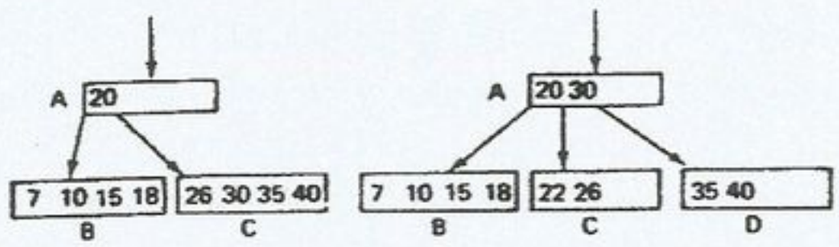
2. $k_m < x$. La búsqueda prosigue en la página apuntada por p_m.
3. $x < k_1$. La búsqueda prosigue en la página apuntada por p_0.



Si en algún caso el puntero designado es NULL, esto es, si no hay página de hijo, entonces tampoco existe un elemento con la clave x en todo el árbol y la búsqueda finaliza.

Inserción.

Es interesante señalar que la inserción en un árbol B es relativamente sencilla. Si hay que insertar un elemento en una página con m<2n elementos, el proceso de inserción queda limitado a esa página. Es sólo la inserción en una página ya llena la que tiene consecuencias en la estructura del árbol, pudiendo ocasionar la asignación de páginas nuevas. Para entender lo que sucede en ese caso, veamos la siguiente figura que ilustra la inserción de la clave 22 en un árbol B de orden 2.



La acción se realiza en los siguientes pasos:

1. Se descubre que falta la clave 22; la inserción en la página C es imposible porque C ya está llena.
2. La página C se divide en dos páginas (esto es, se asigna una nueva página D).
3. Las 2n+1 llaves se distribuyen uniformemente en C y D, y la clave de la mitad se sube un nivel hacia la página madre A.

Este plan tan elegante preserva todas las propiedades típicas de los árboles B. En particular, las páginas divididas contienen exactamente n elementos. Desde luego, la inserción de un elemento en la página madre puede hacer que ésta se desborde, con lo cual ocasiona que la división se propague. En el caso extremo, puede propagarse hasta la raíz. Es decir, la única manera en que el árbol B puede aumentar su altura. Tiene, pues, una manera singular de crecer: crece de las hojas hacia la raíz.

Eliminación.

La eliminación de elementos en un árbol B es en teoría bastante sencilla, pero se complica en sus detalles. Podemos distinguir dos circunstancias:

1. El elemento que debe suprimirse se halla en una página de hoja; entonces su algoritmo de eliminación será fácil y sencillo.
2. El elemento no se encuentra en la página de hoja; hay que sustituirlo por uno de dos elementos lexicográficamente contiguos, que resultan estar en las páginas de hoja y son fáciles de suprimir.

En el caso 2 encontrar la clave contigua es semejante a encontrar la que se usó en la eliminación en árboles binarios. Bajamos por los punteros situados al extremo derecho hasta la página de hoja P, reemplazamos el elemento a eliminar con el del extremo derecho en P y luego reducimos en 1 el tamaño de P. En todo caso, la disminución del tamaño debe acompañarse de una verificación del número de elementos en la página reducida, pues se violaría la característica primaria de los árboles B si m<n.

El recurso consiste en tomar o anexar un elemento a partir de una de las páginas vecinas, digamos Q. Puesto que para ello se requiere introducir Q en la memoria principal, sentimos la tentación de salir airosos de esa situación inconveniente y agregar más de un elemento de inmediato. La estrategia habitual consiste en distribuir los elementos en las páginas P y Q de manera uniforme en ambas. A esto se le llama balanceo de páginas.

Por supuesto, puede suceder que no queden elementos por agregar, pues Q ya ha alcanzado su tamaño mínimo n. En este caso, el número total de elementos en las páginas P y Q es 2n-l; podemos combinar las dos páginas en una, agregando el elemento intermedio de la página madre de P y Q y luego prescindir completamente de la página Q. Este es el proceso inverso de la división de página. El proceso puede visualizarse si se examina la eliminación de la clave 22 de la figura anterior. También en este caso, la supresión de la clave de la mitad en la página madre puede hacer que su tamaño baje más allá del límite permisible n, con lo cual requeriría que en el siguiente nivel se tomase una medida especial (balanceo o mezcla). En el caso extremo, la combinación de páginas puede probarse hasta llegar a la raíz. Si el tamaño de ésta se reduce a 0, también se suprime ocasionando una disminución en la altura del árbol B. De hecho, ésa es la única manera en que un árbol B puede disminuir de tamaño.

6 Grafos.

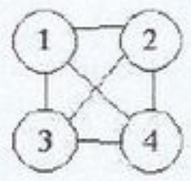
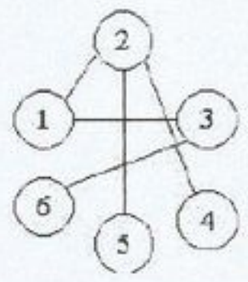
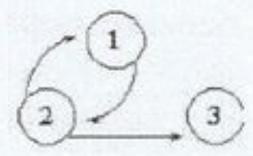
Los grafos podemos considerarlos como árboles no jerarquizados, o lo que es lo mismo, los árboles son un caso concreto de grafos.

6.1 Conceptos y definiciones básicas.

Un grafo, G, es un par, compuesto por dos conjuntos V y A. Al conjunto V se le llama conjunto de vértices o nodos del grafo. A es un conjunto de pares de vértices que se conocen habitualmente con el nombre de arcos o ejes del grafo. Se suele utilizar la notación G = (V, A) para identificar un grafo.

Los grafos representan un conjunto de objetos donde no hay restricción a la relación entre ellos. Son estructuras más generales y menos restrictivas. Podemos clasificar los grafos en dos grupos: dirigidos y no dirigidos. En un grafo no dirigido el par de vértices que representa un arco no está ordenado. Por lo tanto, los pares (v1, v2) y (v2, v1) representan el mismo arco. En un grafo dirigido cada arco está representado por un par ordenado de vértices, de forma que y representan dos arcos diferentes.

Ejemplos de grafos (dirigidos y no dirigidos):

G1 = (V1, A1) V1 = {1, 2, 3, 4} A1 = {(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)}	
G2 = (V2, A2) V2 = {1, 2, 3, 4, 5, 6} A2 = {(1, 2), (1, 3), (2, 4), (2, 5), (3, 6)}	
G3 = (V3, A3) V3 = {1, 2, 3} A3 = { <1, 2>, <2, 1>, <2, 3> }	

Los grafos permiten representar conjuntos de objetos arbitrariamente relacionados. Se puede asociar el conjunto de vértices con el conjunto de objetos y el conjunto de arcos con las relaciones que se establecen entre ellos.

El número de distintos pares de vértices (v(i), v(j)), con v(i) <> v(j), en un grafo con n vértices es n*(n-1)/2. Este es el número máximo de arcos en un grafo no dirigido de n vértices. Un grafo no dirigido que tenga exactamente n*(n-1)/2 arcos se dice que es un grafo completo. En el caso de un grafo dirigido de n vértices el número máximo de arcos es n*(n-1).

Algunas definiciones básicas en grafos:

- Orden de un grafo: es el número de nodos (vértices) del grafo.
- Grado de un nodo: es el número de ejes (arcos) que inciden sobre el nodo
- Grafo simétrico: es un grafo dirigido tal que si existe la relación (v, u) entonces existe (u, v), con u, v pertenecientes a V.
- Grafo no simétrico: es un grafo que no cumple la propiedad anterior.
- Grafo reflexivo: es el grafo que cumple que para todo nodo u de V existe la relación (u, u) de A.
- Grafo transitivo: es aquél que cumple que si existen las relaciones (u, v) y (v, z) de A entonces existe (u, z) de A.
- Grafo completo: es el grafo que contiene todos los posibles pares de relaciones, es decir, para cualquier par de nodos u, v de V, (u <> v), existe (u, v) de A.
- Camino: un camino en el grafo G es una sucesión de vértices y arcos: v(0), a(1), v(1), a(2), v(2), ... , a(k), v(k); tal que los extremos del arco a(i) son los vértices v(i-1) y v(i).

- Longitud de un camino: es el número de arcos que componen el camino.
- Camino cerrado (circuito): camino en el que coinciden los vértices extremos (v(0) = v(k)).
- Camino simple: camino donde sus vértices son distintos dos a dos, salvo a lo sumo los extremos.
- Camino elemental: camino donde sus arcos son distintos dos a dos.
- Camino euleriano: camino simple que contiene todos los arcos del grafo.
- Grafo euleriano: es un grafo que tiene un camino euleriano cerrado.
- Grafo conexo: es un grafo no dirigido tal que para cualquier par de nodos existe al menos un camino que los une.
- Grafo fuertemente conexo: es un grafo dirigido tal que para cualquier par de nodos existe un camino que los une.
- Punto de articulación: es un nodo que si desaparece provoca que se cree un grafo no conexo.
- Componente conexa: subgrafo conexo maximal de un grafo no dirigido (parte más grande de un grafo que sea conexa).

6.2 Representación de grafos.

Existen tres maneras básicas de representar los grafos: mediante matrices, mediante listas y mediante matrices dispersas.

6.2.1 Representación mediante matrices: matrices de adyacencia.

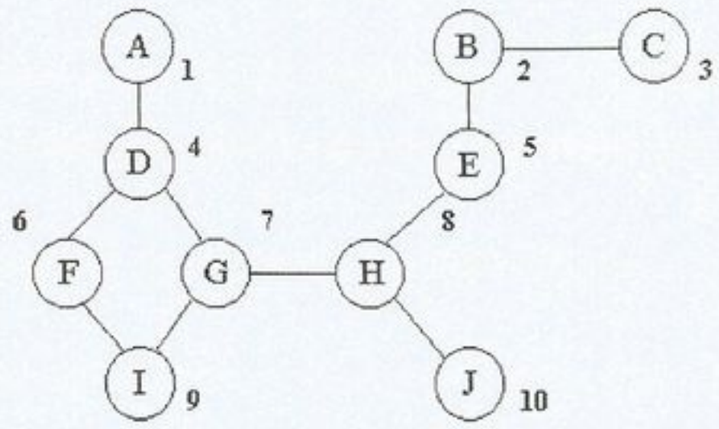
Un grafo es un par compuesto por dos conjuntos: un conjunto de nodos y un conjunto de relaciones entre los nodos. La representación tendrá que ser capaz de guardar esta información en memoria.

La forma más fácil de guardar la información de los nodos es mediante la utilización de un vector que indexe los nodos, de manera que los arcos entre los nodos se pueden ver como relaciones entre los índices. Esta relación entre índices se puede guardar en una matriz, que llamaremos de adyacencia.

Con esta representación tendremos que reservar al menos del orden de (n^2) espacios de memoria para la información de los arcos, y las operaciones relacionadas con el grafo implicarán, habitualmente, recorrer toda la matriz, con lo que el orden de las operaciones será, en general, cuadrático, aunque tengamos un número de relaciones entre los nodos mucho menor que (n^2).

En cambio, con esta representación es muy fácil determinar, a partir de dos nodos, si están o no relacionados: sólo hay que acceder al elemento adecuado de la matriz y comprobar el valor que guarda.

Ejemplo: Supongamos el grafo representado en la figura siguiente.



A partir de ese grafo la información que guardaríamos, con esta representación, sería:

GRAFO

Nodos

	1	2	3	4	5	6	7	8	9	10	11
Existe	T	T	T	T	T	T	T	T	T	T	F
	A	B	C	D	E	F	G	H	I	J	

Arcos

	Existe	1	2	3	4	5	6	7	8	9	10	11
1		F	F	F	V	F	F	F	F	F	F	
2		F	F	F	F	V	F	F	F	F	F	
3		F	F	F	F	V	F	F	F	F	F	
4		V	F	F	F	F	V	V	F	F	F	
5		F	V	V	F	F	F	F	V	F	V	
6		F	F	F	V	F	F	V	F	F	F	
7		F	F	F	V	F	F	F	V	V	F	
8		F	F	F	F	V	F	V	F	F	V	
9		F	F	F	F	F	V	V	F	F	F	
10		F	F	F	F	V	F	F	V	F	F	
11												
12												

6.2.2 Representación mediante punteros: listas de adyacencia.

En las listas de adyacencia se intenta evitar justamente el reservar espacio para aquellos arcos que no contienen ningún tipo de información. El sustituto obvio a los vectores con huecos son las listas.

En las listas de adyacencia lo que haremos será guardar por cada nodo, además de la información que pueda contener el propio nodo, una lista dinámica con los nodos a los que se puede acceder desde él. La información de los nodos se puede guardar en un vector, al igual que antes, o en otra lista dinámica.

En general con esta representación sólo se reservará memoria para aquellos arcos que efectivamente existan, pero como contrapartida estamos guardando más espacio para cada uno de los arcos (estamos añadiendo el índice destino del arco y el puntero al siguiente elemento de la lista de arcos).

Las tareas relacionadas con el recorrido del grafo supondrán sólo trabajar con los vértices existentes en el grafo, que puede ser mucho menor que (n^2). Pero comprobar las relaciones entre nodos no es tan directo como lo era en la matriz, sino que supone recorrer la lista de elementos adyacentes perteneciente al nodo analizado.

Además, sólo estamos guardando realmente la mitad de la información que guardábamos en el caso anterior, ya que las relaciones inversas (las relaciones que llegan a un cierto nodo) en este caso no se guardan, y averiguarlas supone recorrer todas las listas de todos los nodos.

6.2.3 Representación mediante punteros: matrices dispersas.

Para evitar uno de los problemas que teníamos con las listas de adyacencia, que era la dificultad de obtener las relaciones inversas, podemos utilizar las matrices dispersas, que contienen tanta información como las matrices de adyacencia, pero, en principio, no ocupan tanta memoria como las matrices, ya que al igual que en las listas de adyacencia, sólo representaremos aquellos enlaces que existen en el grafo.

6.3 Recorrido de grafos.

Recorrer un grafo supone intentar alcanzar todos los nodos que estén relacionados con uno dado que tomaremos como nodo de salida. Existen básicamente dos técnicas para recorrer un grafo: el recorrido en anchura; y el recorrido en profundidad.

6.3.1 Recorrido en anchura o BFS (*Breadth First Search*).

Supone recorrer el grafo, a partir de un nodo dado, en niveles, es decir, primero los que están a una distancia de un arco del nodo de salida, después los que están a dos arcos de distancia, y así sucesivamente hasta alcanzar todos los nodos a los que se pudiese llegar desde el nodo salida.

La diferencia a la hora de implementar el algoritmo para cada una de las implementaciones residirá en la manera de averiguar los diferentes nodos adyacentes a uno dado. En el caso de las matrices de adyacencia se tendrán que comprobar si los enlaces entre los nodos existen en la matriz. En los casos de las listas de adyacencia y de las matrices dispersas sólo habrá que recorrer las listas de enlaces que parten del nodo en cuestión para averiguar qué nodos son adyacentes al estudiado.

6.3.2 Recorrido en profundidad o DFS (Depth First Search).

Trata de buscar los caminos que parten desde el nodo de salida hasta que ya no es posible avanzar más. Cuando ya no puede avanzarse más sobre el camino elegido, se vuelve atrás en busca de caminos alternativos, que no se estudiaron previamente.

7 Conclusiones.

Siempre que se utilice un dato en un programa debe estar determinado su tipo. En el caso de las estructuras estáticas y datos de tipos elementales, el tipo del dato determina el espacio que se usa en memoria. Esto puede no ocurrir si el dato es de un tipo estructurado. Algunos tipos estructurados (listas y árboles) se declaran sin especificar el número de componentes que van a tener. En este caso el compilador les reserva el espacio de memoria mínimo que necesitan. Durante la ejecución del programa la estructura de datos puede ir creciendo, es decir, ocupando más memoria. En cualquier caso, el máximo espacio al que pueden llegar está limitado por el espacio libre en el programa. Si se necesitase más memoria de la disponible en el programa, este terminaría por error. Una estructura de datos que es gestionada de esta forma se dice que es dinámica, ya que la memoria que necesita se asigna dinámicamente. Por el contrario, una estructura de datos que siempre ocupa el mismo espacio se dice que es estática.

Las estructuras dinámicas se implementan utilizando punteros. Un puntero es un dato que contiene una dirección de memoria.

Una lista está formada por un número variable de datos (elementos) de un mismo tipo, ordenados según una secuencia lineal. Cada elemento, salvo el primero, tiene un predecesor en la lista. Todos los elementos, salvo el último, tienen un sucesor.

Un caso particular de lista se denomina pila, donde cualquier elemento añadido pasa a ser el primero de la lista. Además, sólo puede se puede acceder o eliminar el elemento que ocupa la primera posición de la lista.

Se denomina cola a una lista en que las inserciones se realizan sólo en el final y sólo se puede acceder o eliminar el primer elemento de la lista.

Un árbol es una estructura de datos formada por elementos del mismo tipo, llamados nodos, relacionados de tal modo que el árbol puede descomponerse en un nodo llamado raíz y un conjunto finito de objetos de tipo árbol, llamados subárboles de nodo raíz.

Los grafos representan un conjunto de objetos donde no hay restricción a la relación entre ellos. Son las estructuras más generales y menos restrictivas.