UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL MENDOZA

INTRODUCCIÓN a las ESTRUCTURAS de DATOS y ARCHIVOS

Ing. Santiago C. PÉREZ

INDICE

INDICE

INTRODUCCIÓN

La recapitulación de bibliografía sobre el tema, ha permitido la generación del presente apunte, cuyo objeto fundamental es poner a disposición de cátedras con contenidos afines, docentes, estudiantes e interesados en general este material, como una referencia adicional, aunque breve, conceptual e informativa sobre el particular.

La lectura del texto requiere conocimientos de computación y programación, a fin de adquirir nociones fundamentales de Estructuras de Datos y Archivos.

Se proponen ejercicios a resolver, sencillos, aunque útiles para consolidar los conceptos adquiridos. Además, se deja a disposición de los lectores programas fuente directamente relacionados con los contenidos. Los mismos estarán a disposición como material complementario del presente texto.

El que suscribe estaría muy agradecido de aquellos lectores, a quienes el mismo haya resultado de utilidad, y que puedan aportar una critica o información para su mejora.

Finalmente, agradezco la colaboración de los Ayudantes de Segunda, Carlos Campos y Laura Noussan Lettry, en la revisión de los contenidos y el pasaje a formato electrónico.

Mendoza, Junio de 2004.

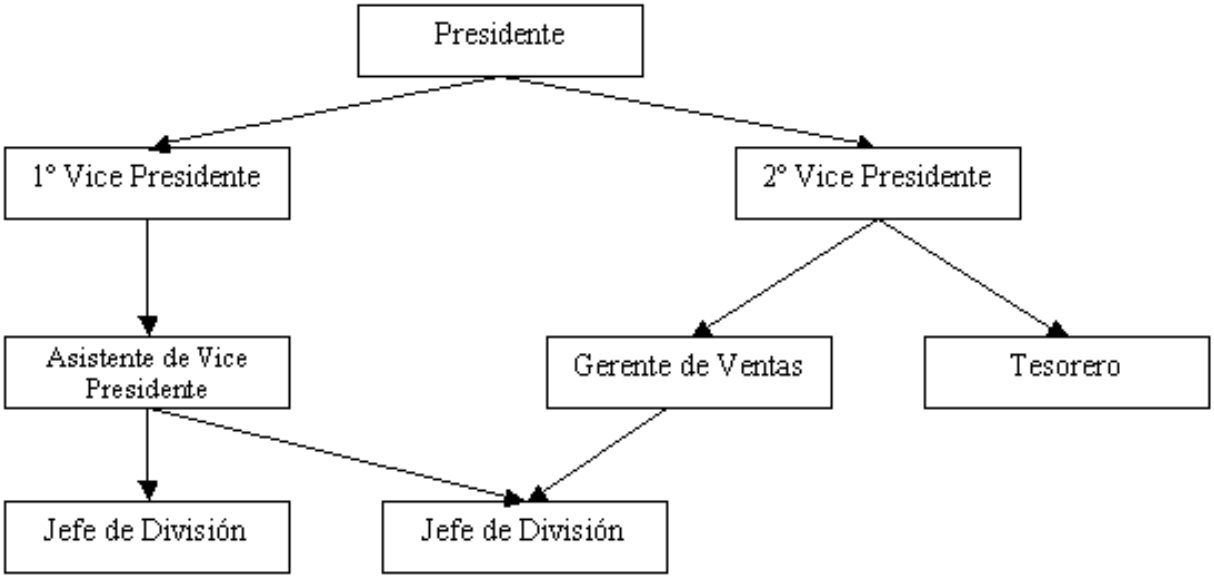
Ing. Santiago Pérez
Docente Cátedra Gestión de Datos
Email: santiagocp@frm.utn.edu.ar

TEMA 1: Modelos matemáticos y representaciones de computadora.

Estructuras de Datos y Archivos.

Comúnmente se define a los datos como un conjunto de hechos, de números, o de símbolos que usan y manipulan los programas de computadora. Una estructura de datos o archivos puede definirse como aquellos datos que denotan un conjunto de hechos, que pueden servir como operandos a un programa de computadora y cuya estructura es la manifestación de las relaciones (si las hay) entre elementos individuales de aquel conjunto. Por ejemplo, un conjunto de datos acerca de una empresa puede contener los nombres de todos los empleados, y los cargos de todos los niveles jerárquicos.

Así, podemos considerar la relación entre los niveles jerárquicos, es decir, que niveles jerárquicos son superiores a otros. Al resultado, frecuentemente se le llama estructura de organización de la empresa (Figura 1).



(Figura 1)

El estudio de estructuras de datos y archivos aproxima al interesado a encontrar resultados que son también aplicables a la teoría de sistemas operativos, de computación, y de lenguajes de programación, además de a las estructuras de datos y archivos en sí mismas.

Este estudio comprende dos objetivos complementarios. El primero es identificar y desarrollar modelos abstractos y operaciones útiles sobre las estructuras de datos, y determinar que clases de problemas pueden resolverse con ellos. El segundo objetivo es el de determinar las representaciones para aquellos modelos abstractos e implementar las operaciones sobre ellos con tipos de datos concretos y conocidos.

El primero de estos dos objetivos considera un tipo de datos superior, como una herramienta que puede ser utilizada para resolver nuevos problemas o mejorar otros; mientras que el segundo considera la implementación de dichos tipos de datos como un problema para ser resuelto utilizando tipos de datos ya existentes.

Por otro lado, deberemos distinguir que la teoría y las técnicas de las estructuras de datos y de archivos difieren entre sí. El estudio de las estructuras de archivos es, en esencia, la aplicación de las técnicas de estructuras de datos a problemas especiales asociados con el almacenamiento y la recuperación de datos en dispositivos de almacenamiento secundario.

Normalmente el estudio de las estructura de datos supone que éstos se encuentran almacenados en memoria RAM. Un archivo constituido por decenas o más de millones de registros es demasiado grande para cargarlo en memoria RAM a fin de efectuar operaciones sobre él. Dichas operaciones implican el uso de almacenamiento secundario, y por lo tanto, ya no se está en un ambiente de acceso aleatorio. Ahora, el costo de recuperar algunas partes de la información es desproporcionadamente mayor que el costo de recuperar otras. Un enfoque para las operaciones que se oriente al diseño de estructuras de archivos reconocería esto. Así minimizaría el número de acceso al disco y además intentaría adecuar el patrón de accedo de modo tal que, cuando se necesite otra parte de la información, ésta se localice donde el acceso resulte relativamente barato.

Por lo expuesto, salvando las dimensiones de los datos involucrados, que nos forzaría a reconocer las técnicas de estructuras de datos o de archivos, para su operación, usaremos el término estructura de datos para ambos casos.

Una consideración importante a la hora de la implementación es reconocer las capacidades propias, tipos de datos y operaciones disponibles en el lenguaje de programación utilizado. Existen lenguajes de programación de alto nivel que tienen muchos tipos de datos superiores implementados internamente, pero a su vez muchos de ellos son altamente ineficientes y poco utilizados. Hoy en día, es común utilizar lenguaje de programación orientados a objetos por su riqueza en estructuras de control, y por que permite concretar con sus tipos de datos disponibles con relativa facilidad los tipos de datos superiores.

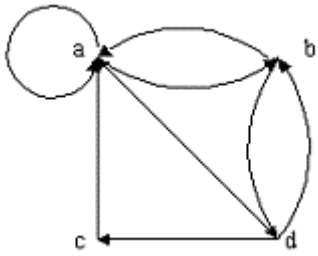
Modelos Matemáticos.

La herramienta más potente de la aproximación científica, es la descripción de un fenómeno por un modelo, que se discute y evalúa. Por ello, es común reemplazar el concepto intuitivo de estructura de datos por un modelo matemático preciso, conocido con el nombre de grafo dirigido con funciones de asignación. Los grafos dirigidos son modelos adecuados para la mayoría de las estructuras de datos, quedando refinaciones de este modelo para casos más complejos.

Sea P cualquier conjunto de elementos, puntos, números, etc.; y E una relación sobre P, cualquier conjunto de pares ordenados de elementos de P; por ejemplo:

$$P = \{a,b,c,d\}$$
$$E = \{(a,a), (a,b), (b,a), (b,d), (c,a), (d,a), (d,b), (d,c)\};$$

Se puede obtener la representación gráfica de la relación comúnmente llamada grafo de la relación (Figura 2).



(Figura 2)

Un grafo dirigido, simbolizado G = (P, E), significa simplemente una relación E sobre un conjunto de puntos P. No hay distinción entre los términos grafo y relación.

Un par ordenado (x, y) en la relación, que se representa en el grafo con una flecha, o línea, se conoce normalmente con el nombre de arco del grafo. En las aplicaciones normalmente, se asocian datos, en la forma de uno o más valores, en cada uno de los puntos del grafo y posiblemente en cada uno de los arcos.

Tales grafos se llaman grafos con asignación de puntos y/o asignación de arcos. Un string simbólico, llamado identificador, puede utilizarse para referenciar cada uno de los elementos individuales de la estructura, o la estructura entera.

Representaciones.

Los grafos dirigidos son un modelo abstracto muy útil para derivar resultados teóricos acerca de la naturaleza y uso de las estructuras de datos. Este modelo debe tener una versión concreta de la estructura de datos dentro de la computadora, y además, una correcta correspondencia entre la representación en la computadora y la estructura de datos abstracta.

En la búsqueda de una terminología común, llamaremos celda a la unidad direccionable más pequeña de una estructura de datos, en una aplicación dada, y campo a una porción de la celda. De hecho, una celda deber1a interpretarse como un bloque de una o más palabras de memoria consecutivas. Realmente, existe libertad para visual izar la celda de la manera más análoga a la implementación en la computadora y al lenguaje de programación utilizado, aunque lo más común es referirse a la celda con el término registro.

Con igual criterio, llamaremos campo link al campo en una celda cuyo valor es siempre una dirección de celda, y puntero a una variable común dentro de un programa que apunta a una celda particular de la estructura.

Las representaciones de computadora se construirán usando celdas, campos, links y punteros como sus componentes. Las celdas corresponderán a los puntos de los grafos, los campos link a los arcos, mientras que todos los otros campos, que pueden ser llamados campos de datos, representan valores de asignación asociados con los puntos. Finalmente, los punteros serán utilizados como identificadores para apuntar, o identificar, a elementos individuales, celdas o puntos de la estructura de datos.

Conceptos de teoría de grafos.

La teoría de grafos es una disciplina dentro de las matemáticas discretas cuyas definiciones, convenciones notacionales, y resultados se usan repetidamente en las discusiones de las estructuras de datos, desde su modelo abstracto.

Un grafo dirigido (o simplemente grafo), simbolizado G=(P,E), es una relación E sobre un conjunto P. los elementos de P se conocen como puntos (nodos o vértices), y los de E como arcos (aristas o líneas).

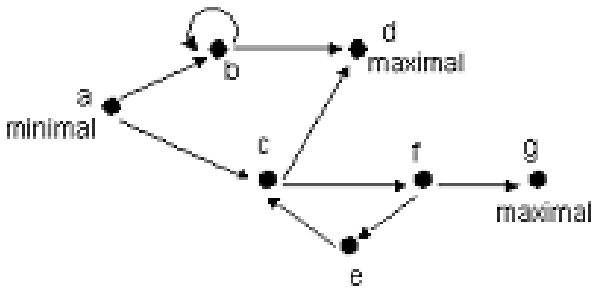
Los conjuntos P y E que constituyen un grafo pueden describirse por enumeración o por fórmula que haga posible determinar cualquier elemento en los conjuntos. Por ejemplo, un grafo puede definirse por enumeración de la siguiente manera:

G=(P,E); P={a,b,c,d,e,f,g}

E={ (a,b), (a,c), (b,b), (b,d), (c,d), (c,f), (e,c), (f,e), (f,g)}.

Sin embargo, esta representación da poca noción de la estructura del grafo.

Es más común ilustrar al grafo por un dibujo (Figura 3), identificado con el nombre de grafo, y que es simplemente una representación gráfica del grafo G. Otro método comúnmente usado de representar un grafo es por medio de notación matricial. Una matriz booleana A es la matriz de adyacencia del grafo G, si para todos los puntos ij que pertenecen a P, Aij=1, suponiendo que el par ordenado (i, j) pertenece a la relación, y Aij=0 de lo contrario.



	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	0	1	0	1	0	0	0
c	0	0	0	1	0	1	0
d	0	0	0	0	0	0	0
e	0	0	1	0	0	0	0
f	0	0	0	0	1	0	1
g	0	0	0	0	0	0	0

(Figura 3)

Un grafo G=(P,E) tiene asignación de puntos sobre un conjunto V, si existe una función f:P→V, y de asignación de arcos sobre un conjunto V', si existe una función g:E→V'.

De la representación gráfica de un grato, pueden reconocerse una serie de características importantes, y aplicables en la definición formal de las estructuras de datos. Por ejemplo, el hecho de que un elemento no sea primera o segunda componente de ningún par ordenado, tiene una significación particular. Estos puntos representan algo independiente no sólo de la semántica de la relación, sino también de la estructura particular de la relación.

Definiremos los puntos que no son segunda componente de ningún par ordenado, salvo eventualmente de si mismos, con el nombre de minimals. En símbolos, x es minimal en G, sí y sólo sí

$$(y, x) \in E \rightarrow y = x,$$

no exigiendo que haya un par ordenado (x, x), sino permitiendo que exista. Al par (x, x) lo llamaremos lazo, diciendo que en x hay lazo, sí y sólo sí

$$(x, x) \in E$$

De igual forma, definiremos los puntos que no son primera componente de ningún par ordenado, salvo eventualmente de sí mismos, con el nombre de maximals. En símbolos, x es maximal en G. si y sólo sí

$$(x, y) \in E \rightarrow y = x$$

Definiremos al conjunto de puntos que son segunda componente de un elemento dado, con el nombre de conjunto right. En símbolos,

$$right(x) = R(x) = \{y/(x, y) \in E\}$$

Igualmente, al conjunto de puntos que son primera componente de un elemento dado, con el nombre de conjunto left. En símbolos,

$$left(x) = L(x) = \{y/(y, x) \in E\}$$

Desde un punto de vista cuantitativo, nos interesará cuantas primeras o segundas componentes distintas tiene un elemento dado.

Es lo que se denomina grado de entrada o de salida (input o outpout degree). Definiremos el input degree como la cardinalidad del conjunto left. En símbolos,

$$input\ degree(x) = ID(x) = |L(x)|$$

Al output degree como la cardinalidad del conjunto right. En símbolos,

$$output\ degree(x) = OD(x) = |R(x)|$$

Es importante saber si existe alguna dependencia directa o indirecta, que permita el tránsito entre los puntos. Definiremos con el nombre de paso en el grafo, desde el punto x al punto y, al par ordenado (x, y) que cumpla con las siguientes condiciones:

1) $Zo = x$
   $Zn = y$

2) $Zi <> Zi + 1$; para todo $0 <= i < n$ excepto $Zo = Zn$

3) $(Zi, Zi + 1) \in E$; para todo $0 <= i < n$

La existencia de paso entre los puntos x e y se simbolizará e (x, y). La longitud de paso es un número n igual a la cantidad de arcos para pasar del punto inicial al final del paso, y denotada como la cardinalidad del paso:

$$| e (x,y) | = n$$

Al hecho de salir desde un punto, transitar por otros, y retornar al punto inicial, es algo muy particular. Definiremos como ciclo a un paso e (x, x) cuya longitud es >= a dos.

Este concepto de paso, permite plantear la necesidad de determinar cual es el conjunto de puntos que son primera o segunda componente que definen paso con un elemento dado. Es decir, hacia o desde que puntos hay paso, dado un elemento. Definiremos como ideal principal derecho de un elemento dado al conjunto de puntos que definen paso desde ese elemento. En símbolos,

$$\overline{R(x) = \{y / e (x, y)\}}$$

Igualmente, definiremos como ideal principal izquierdo de un elemento dado, al conjunto de puntos que definen paso hacia ese elemento. En símbolos,

$$\overline{L(x) = \{y / (e (y, x)\}}$$

El conjunto de pares ordenados (x, y) que definen paso, fijan una relación que tiene una ley de conformación propia, aunque dependiente de la relación E. Definiremos a dicha relación como relación de paso, y simbólicamente expresada como:

$$e_{\epsilon} = \{ (x, y) / e (x,y)\} ;$$

y al grafo G' = (P, e_{ϵ}), la clausura transitiva del grafo G = (P, E).

Para culminar, definiremos como grafo conectado a aquel que satisface la inecuación:

$$|E| >= |P| - 1$$

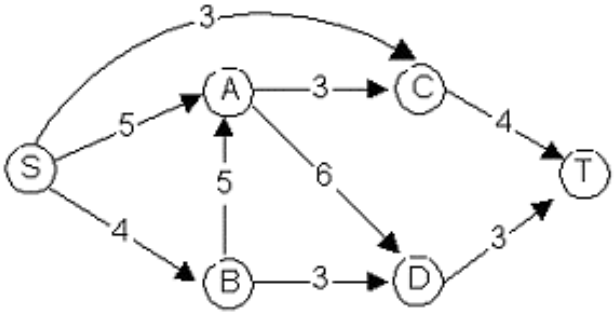
Como ilustración del uso de los conceptos previos, existen una serie de problema reales, cuyas soluciones pueden conducir a formulaciones similares.

Un problema interesante es evaluar un sistema de tuberías de transporte de agua con grafos (Figura 4). Cada arco puede representar un tubo y el número de encima de cada arco la capacidad de esa tubería en litros por segundo.

Los puntos representan lugares en los cuales las tuberías están unidas, y el agua es transportada de un tubo a otro. Los puntos, S y T, representan la fuente de agua, y un consumidor, respectivamente.

Un problema a plantear sería maximizar la cantidad de agua que fluye desde la fuente hasta el punto de consumo, dada la capacidad limitante de la tubería.

La solución requiere establecer las funciones capacidad c(a, b), y flujo f(a, b) donde a y b son puntos del grafo.



(Figura 4)

Ejercicios.

1. Dibuje dos grafos distintos con cinco nodos, siete arcos, por lo menos un nodo maximal y un nodo minimal, y sin ciclos.

2. Considerando un grafo G = (P, E) donde P es el conjunto de enteros positivos <= 8; es decir, P = {1,2,3,....,7,8} y E es la relación tal que E = {(x, y) / x divide exactamente a y},

a) Dibuje la representación de este grafo. ¿Qué arcos extras deberían dibujarse para ilustrar la relación clausura transitiva completa?

b) Represente el grafo con su matriz de adyacencia.

3. Para el grafo del ejercicio anterior,
a) Encuentre los pasos utilizando las potencias de la matriz de adyacencia.

b) Indique cuales son las características que debe cumplir la matriz de adyacencia para que un punto dado sea minimal o maximal.

TEMA 2: Estructuras lineales de Datos.

Propiedades formales. Relaciones lineales. Listas.

Las estructuras lineales de datos son las más comúnmente utilizadas, y se las conoce con una amplia variedad de nombres. Los grafos que le corresponden reciben el nombre de grafos lineales, dadas sus características gráficas (Figura 5).

(Figura 5)

Desde el punto de vista formal, es posible establecer que un grafo finito G = (P, E), que le corresponde a una estructura de datos lineal, es lineal sí y solo sí:

1. existe un punto x Є P tal que L(x) = 0;

2. para todo y Є P |L(y)| <= 1, |R(y)| <= 1, Y

3. G es conectado.

Este concepto formal define las características que debe tener un grafo que le corresponde a una estructura lineal, a la que a su vez denominaremos simplemente con el nombre de lista. El grafo lineal, como modelo abstracto de las listas, será siempre igual, salvo la cantidad de elementos y la naturaleza de las funciones de asignación definidas sobre los puntos y/o los arcos.

Tipos de listas.

Las estructuras lineales poseen una amplia variedad de representaciones de computadora alternativas, cuyas características están establecidas normalmente por la clase de procedimientos que. operarán sobre la estructura de datos. Por esta razón, las listas se clasifican de acuerdo a su futuro uso; algunos autores en particular, las clasifican de acuerdo a las operaciones de sumar o eliminar elementos del conjunto ordenado linealmente. Así, a una simple lista, dependiendo de su uso, se le llama:

1. N-upla (o vector, o arreglo simplemente subindicado) si no pueden sumarse o eliminarse elementos, es decir, la lista es de tamaño fijo.

2. Pila (o stack, o lista LIFO) si todas las sumas o eliminaciones de elementos ocurren en un extremo.

3. Cola (o fila, o lista FIFO) si todas las sumas ocurren en un extremo, y todas las eliminaciones en el otro.

4. Doble cola si las sumas o eliminaciones pueden ocurrir en cualquier extremo.

5. Listas irrestrictas (o simplemente listas) si se permite cualquier tipo de dinámica.

Las n-uplas corresponden al tipo predefinido arreglo (array) en los lenguajes de programación, cuando se utiliza un sólo subíndice. Estas son desde el punto de vista abstracto un conjunto ordenado, finito de elementos homogéneos. Por finito se quiere decir que existe un número especifico de elementos en la n-upla, y que además, no puede ser modificado; por ordenado que los elementos están organizados de tal manera que existe un primero, segundo, tercero, etc.. Las técnicas de asignación y extracción de contenidos a los elementos de las n-uplas se consideran conocidas.

Representaciones secuencial y linkeada.

Los restantes tipos de listas constituyen tipos de datos superiores o más complejos, y la implementación de los mismos deberá hacerse utilizando los tipos existentes, y/o recurriendo a los mecanismos de asignación dinámica de memoria. La elección en la representación computacional de estos tipos de datos superiores está regida por la eficiencia en tiempo y espacio. Si una aplicación en particular depende considerablemente de la manipulación de la estructura de datos, la velocidad a la cual estas manipulaciones pueden ser realizadas será el mayor determinante de la velocidad de toda la aplicación. Desde otro punto de vista, si un programa utiliza un gran número de tales estructuras, será impráctico una representación que utilice una cantidad muy alta de espacio para representar la estructura de datos. Lamentablemente, existe un balance entre estas dos eficiencias de tal manera que la representación que es rápida utiliza más almacenamiento que aquella que es más lenta, por lo que la elección requiere una evaluación cuidadosa. El eje de la elección está determinado por la forma en que se representen los arcos del grafo dirigido que les corresponde a las estructuras de datos.

La primera alternativa disponible es la representación secuencial, que usa el orden impuesto sobre las posiciones de memoria de la computadora por su convención de direccionamiento, para representar implícitamente los arcos del grafo. En términos de almacenamiento usado, está representación será la más eficiente.

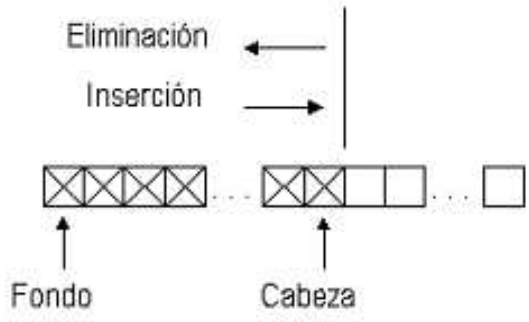
La segunda alternativa es la representación linkeada, donde los arcos del grafo son representados explícitamente al almacenar en el campo link de la celda que le corresponde a cada punto del grafo la dirección de almacenamiento de la celda del siguiente punto en el grafo. Esta representación será la más eficiente en términos del tiempo usado para las manipulaciones de la estructura de datos.

En el estudio de las estructuras de datos existe una regla general que dice: cualquier eficiencia obtenida por compactar la representación computacional de una estructura dada se compensa por un incremento correspondiente en el tiempo de computación de los procesos que operan sobre ella.

Pila.

Una pila (o stack, o lista LIFO) es un conjunto ordenado de elementos homogéneos, en el cual en un extremo se pueden sumar o eliminar elementos, llamado la parte superior (top) del stack. Según la definición los nuevos elementos deben colocarse en la parte superior de la pila, en cuyo caso la parte superior de la pila se mueve hacia arriba para dar lugar al nuevo elemento más alto, o los elementos que están en la parte superior pueden ser removidos, en cuyo caso la

parte superior de la pila se mueve hacia abajo para corresponder al nuevo elemento más alto (Figura 6).



(Figura 6)

Los dos cambios que se pueden hacer en una pila tienen nombres especiales: push y pop. Cuando se agrega un elemento a la pila, éste es empujado (pushed) dentro de la pila. Igualmente, el elemento puede ser retirado (poped) de la pila. No existe un límite en el número de elementos que se pueden mantener en una pila, en un todo de acuerdo a la definición de la misma. Sin embargo, si una pila contiene un solo elemento, y se realiza la operación de retiro, la pila resultante no contendrá elementos, y en este caso se denomina pila vacía. Por consiguiente, antes de aplicar la operación de retiro a una pila hay que asegurarse que no esté vacía. El resultado de tratar ilegalmente de retirar un. elemento de, una pila vacía se denominada underflow.

En general, una pila se puede utilizar en cualquier situación en donde el método de que el último que entra es el primero que sale. Un ejemplo característico lo constituye la validación de paréntesis de expresiones matemáticas, donde se requiere que exista igual número de paréntesis a la izquierda y a la derecha, y que cada paréntesis de la derecha esté precedido por el correspondiente paréntesis a la izquierda.

Este concepto abstracto de una pila, como caso particular de las listas y grafos lineales, puede ser representado en forma secuencial o linkeada, y dentro de cada una de ellas con varias opciones. Como introducción a las representaciones, asumiremos la más sencilla de las representaciones secuencial.

Una pila es una colección ordenada de elementos, y en los lenguajes de programación ya existe un tipo de dato que representa una colección ordenada de elementos y que ya vimos: el arreglo. Lamentablemente, sin embargo, una pila y un arreglo son dos conceptos completamente diferentes. El número de elementos en un arreglo es fijo, y determinado por la definición del mismo. Por otro lado, una pila es fundamentalmente un objeto dinámico, cuyo tamaño está cambiando constantemente a medida que los elementos son empujados o retirados en ella. A pesar de ello, aunque un arreglo no puede ser una pila, éste si puede ser el recipiente de ella. Durante el curso de ejecución del programa, la pila crecerá y se encogerá dentro del espacio reservado para ella (Figura 6). Un extremo del arreglo será el fondo fijo de la pila, mientras que la parte superior de la pila estará cambiando constantemente a medida que los elementos son empujados o retirados

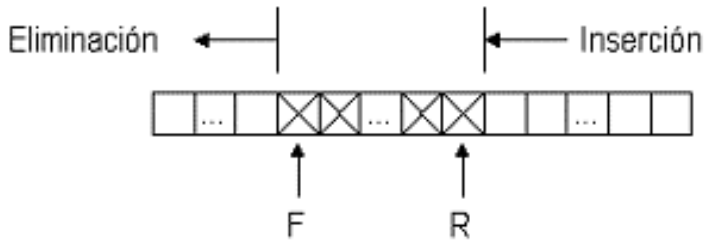
en ella. Además, se requiere otro campo adicional, el cual durante la ejecución del programa en cualquier momento llevará el registro de la posición actual de la parte superior de la pila.

Una pila en Pascal puede por consiguiente ser declarada como un registro que contiene dos objetos: un arreglo que contiene los elementos de la pila, y un entero que indica la posición de la parte superior de la pila dentro del arreglo.

Cola.

Una cola (o fila, o lista FIFO) es una colección ordenada de elementos homogéneos, en la que en un extremo (llamado el frente de la cola) se pueden sumar elementos, y en el otro extremo (llamado el fondo de la cola) se pueden eliminar elementos (Figura 7).

Según la definición, el primer elemento que se coloca en la cola es el primer elemento que será retirado, y por lo tanto, los elementos podrán eliminarse solo en el mismo orden que se sumaron a la cola.



(Figura 7)

Existen tres operaciones primarias que pueden ser aplicadas a una cola. Cuando se agrega un elemento a la cola, este es colocado (insertado) en el frente de la misma. Igualmente, un elemento puede ser eliminado (removido) en el fondo de la cola.

La operación de inserción se puede realizar siempre, puesto que no existe límite en cuanto al número de elementos que puede contener una cola. Por el contrario, la operación de retiro puede aplicarse únicamente si la cola no esta vacía; es decir, no existe forma de remover un elemento de una cola que no contiene elementos. El resultado de un intento ilegal para remover un elemento de una cola vacía se denomina underflow.

Este concepto abstracto de una cola, como para el caso de la pila, puede ser representado a través de un arreglo que contenga los elementos de la cola, y dos variables que contengan las posiciones del arreglo correspondiente al primer y último elemento de la cola.

Por supuesto, que el utilizar un arreglo para que contenga los elementos de una cola introduce naturalmente la posibilidad de un overflow si la cola contiene más elementos de los que fueron reservados, consideración que no es necesario para la representación de la pila si el arreglo es dimensionado adecuadamente. Una solución a este problema, es la de modificar la operación de retiro de elementos de tal manera que cuando se retire uno, se desplace toda la cola al principio del arreglo;

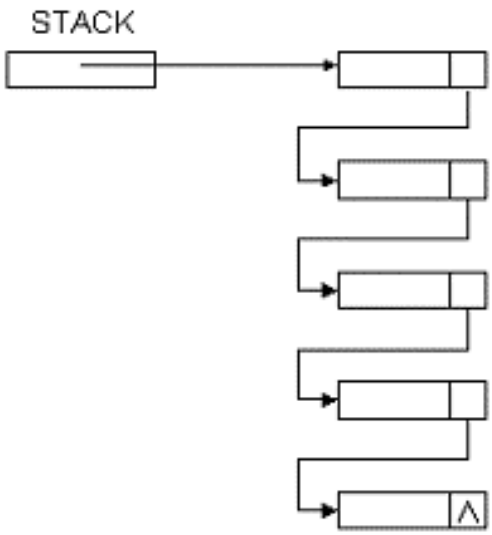
o considerar al arreglo que contiene a la cola como un circulo en lugar de una línea recta, tal que el primer elemento del arreglo esté inmediatamente después del último.

Una de las mayores desventajas de utilizar un sistema de almacenamiento secuencial, a través de arreglos, es que las cantidades fijas de almacenamiento permanecen asignadas a la pila o a la cola aún cuando la estructura esté realmente utilizando una pequeña cantidad de la misma.

La primera alternativa posible es la representación linkeada, donde los elementos contienen un campo link que indica la dirección del siguiente elemento (Figura 8), y una variable puntero que lo hace a algún elemento caracter1stico de la estructura.

Sin embargo, la representación linkeada con el uso de arreglos, suponiendo que existe un conjunto finito de espacios vacíos para los futuros elementos, sin necesidad de reservarlos a priori, también tiene sus ventajas y desventajas.

Al conjunto finito de espacios vacíos se lo denomina comúnmente stack de disponibilidades, y a su vez puede estar representado en forma linkeada en un arreglo. El mismo no puede ser accedido excepto a través de operaciones que retiren o coloquen un elemento, para su uso o reubicación para disponibilidad futura respectivamente, siendo de hecho la fuente que suministra espacio para los elementos de estructuras, como las pilas o colas, sólo cuando lo necesitan, y que se los retira de lo contrario.



(Figura 8)

Obviamente, un elemento de una pila o una cola, representada en forma linkeada, con el uso de un stack de disponibilidades, ocupa más espacio que el elemento correspondiente en un arreglo en representación secuencial, que contenga a aquellas estructuras. Sin embargo, dicho incremento no supera comúnmente el 10%.

Otra desventaja, es el tiempo adicional que se necesita en administrar el stack de disponibilidades. Cada suma o eliminación de un elemento de una pila o de una cola, involucra una eliminación o suma al stack de disponibilidades.

La ventaja de utilizar representación linkeada con el uso de un stack de disponibilidades es que todas las pilas y colas de un programa, tienen acceso al mismo stack de disponibilidades. El espacio que no es usado por una pila, puede ser utilizado por otra estructura, siempre y cuando el número total de elementos en uso en cualquier momento no sea mayor que el número total de disponibles.

Asignación dinámica de memoria.

El concepto de representación linkeada permite construir y manipular listas de diferentes tipos. El campo link introduce la posibilidad de ensamblar un conjunto de nodos de una estructura de datos, en estructuras flexibles. Mediante la alteración de los links, los nodos se pueden enganchar y desenganchar, estableciendo patrones que crecen o encogen a medida que progresa la resolución de un problema, con la ejecución de un programa.

Anteriormente se indicó que es posible utilizar un conjunto finito de nodos, representados por un arreglo o un stack de disponibilidades, para contener una representación linkeada. Un link a un nodo puede representarse por la posición relativa del nodo dentro del arreglo. Sin embargo, la desventaja de este método es doble. Primero, el número de nodos que se requieren no pueden predecirse en el momento en que se escribe el programa para ciertas aplicaciones. Generalmente, los datos con los cuales se debe ejecutar el programa determina el número necesario de datos. Por consiguiente, no interesa cuántos elementos contiene el arreglo o el stack de disponibilidades, siempre es posible que se presente la ejecución del programa con una entrada que requiera un número mayor. La segunda desventaja del método de la representación linkeada con el uso de arreglo o stack de disponibilidades es que cualquiera que sea el número de nodos declarados éstos permanecen asignados al programa a través de su ejecución. Por ejemplo, si se declaran 600 nodos de un tipo determinado, la cantidad de almacenamiento requerido para estos nodos es reservado para tal fin. Si el programa solo utiliza 100, o aún menos en su ejecución, los nodos adicionales permanecen reservados y su almacenamiento no puede ser utilizado para algún otro fin diferente.

La solución a este problema es el de tener variables que sean dinámicas en lugar de estáticas (clasificación en la que entrarían todos los tipos comunes conocidos). Es decir, que cuando se requiere un nodo, se reserva el almacenamiento para él, y cuando ya no se necesita, se libera el almacenamiento.

En los lenguajes de programación, si se utiliza asignación dinámica de almacenamiento, no hay un conjunto de nodos predefinidos que puedan identificarse con un nombre determinado, ni tampoco ninguna organización predefinida del conjunto de nodos. Por ello, cuando una parte de almacenamiento se reserva para ser usada como un nodo, simultáneamente se debe crear un método para acceder al nodo. Este método de acceso es un puntero, el cual puede considerarse como la dirección de la porción de almacenamiento asignado al nodo.

Una vez que se ha declarado una variable puntero a un tipo especifico de objeto, debe ser posible crear dinámicamente un objeto del tipo especifico y asignar su dirección al puntero. Esto se hace en los lenguajes de programación llamando a un procedimiento estándar new:

new(puntero_nodo);

Por el contrario, el procedimiento dispose libera almacenamiento de una variable asignada dinámicamente, siendo la declaración de la siguiente forma:

dispose{puntero_nodo);

La liberación de almacenamiento, puede lograrse también con el uso de los procedimientos estándar mark y release. En esencia, se llama a mark antes de usar new, y release después de usar new, cuando es el momento de desalojar la memoria. Release devuelve toda la memoria asignada entre la llamada de mark y release.

Este método devuelve un bloque de memoria al sistema, mientras que dispose sólo el que corresponde al nodo que direcciona el puntero.

Existe un valor especial que cualquier variable puntero puede tener, haciendo referencia que este no apunta a nada, llamado nil. De igual forma que en una representación linkeada con el uso de arreglos o stack de disponibilidades, hay que manifestar con un valor característico que a un nodo no le sigue otro (por ejemplo para el caso de una pila), en la representación linkeada con el uso de asignación dinámica, tal función la cumple la palabra reservada nil.

En el resto del apunte asumiremos que el termino representación linkeada se refiere a la misma con asignación dinámica de memoria (lo que no invalida las otras representaciones posibles).

Doble cola.

Un doble cola es un conjunto ordenado de elementos homogéneos, en la cual en ambos extremos se pueden sumar o eliminar elementos.

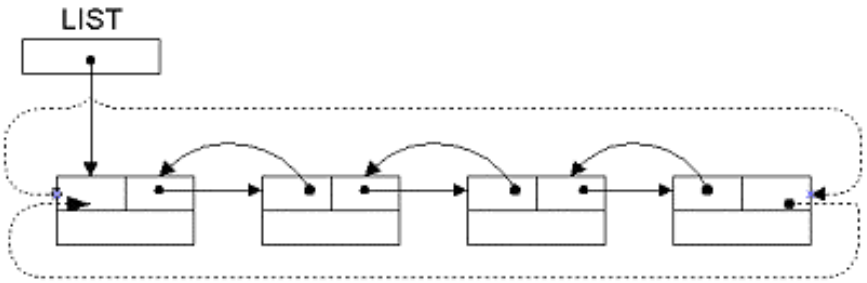
Si asumimos la representación linkeada de la doble cola, se la puede considerar como una simple cola con punteros a ambos extremos.

Según la definición, se pueden sumar o eliminar elementos en cualquiera de los extremos. Sin embargo, la eliminación de un elemento en el extremo opuesto al sentido del linkeado, requiere a dirección del elemento precedente, de modo que el puntero debe ajustarse al elemento precedente, y el campo link no contener dirección a sucesor alguno. Esto requiere un proceso de seguimiento a través de la doble cola. Para evitar este consumo de tiempo, varias dobles colas son doblemente linkeadas. Cada elemento en una doble cola doblemente linkeada contiene un link al elemento siguiente, y un segundo link al elemento inmediatamente precedente (Figura 9).

La inserción o eliminación de elementos en una representación doblemente linkeada es sencilla en cualquiera de los extremos. Además, con el doble linkeado se pueden sumar o restar elementos en el medio de la lista con un poco más de esfuerzo. Por ello, poco se gana al restringir las operaciones a los extremos.

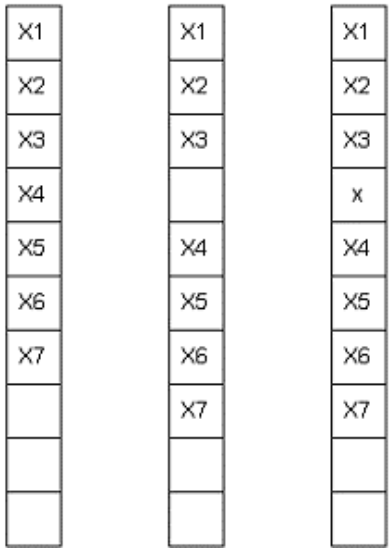
Lista irrestricta.

Un lista irrestricta o simplemente lista es un conjunto ordenado de elementos homogéneos, en la cual se permite cualquier tipo de dinámica, en el sentido que la suma o eliminación de elementos se puede realizar no sólo en los extremos.



(Figura 9)

En una representación secuencial de una lista, además de todas las consideraciones mencionadas en los temas tratados previamente, hay que tener en cuenta la desventaja evidente cuando se quiere insertar o eliminar un elemento en el medio de un grupo de otros elementos. Por ejemplo, supongamos que deseamos insertar un elemento x entre los elementos tercero y cuarto, en una representación secuencial con el uso de un arreglo de 10, el cual contiene actualmente 7 (Figura 10). Los elementos 4 hasta 7 deben ser movidos primero una casilla, y el elemento nuevo insertado en la nueva posición disponible 4. En este caso, el adicionar un elemento representa mover cuatro elementos, además del elemento que se está insertando. Si el arreglo contiene muchos elementos, tendrán que moverse un gran número de elementos. Igualmente, para retirar un elemento del arreglo, todos los elementos siguientes al elemento a ser retirado deben ser movidos una posición.



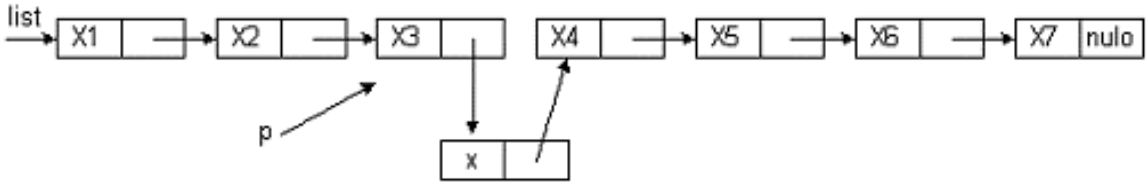
(Figura 10)

En una representación linkeada de la lista, como en cualquier estructura de datos, el programa podrá acceder al primer elemento por una variable puntero, y por los links a cada elemento posterior (Figura 11). Por ello, la variable puntero sirve realmente para identificar la estructura entera.

Cuando cada elemento es explícitamente linkeado al siguiente, pero no al inmediatamente precedente, se dice que tal lista es simplemente linkeada. Si se sumara un segundo campo link en cada elemento indicando al precedente, la lista se llamaría doblemente linkeada, tal cual se planteó en el caso de la doble cola.

(Figura 11)

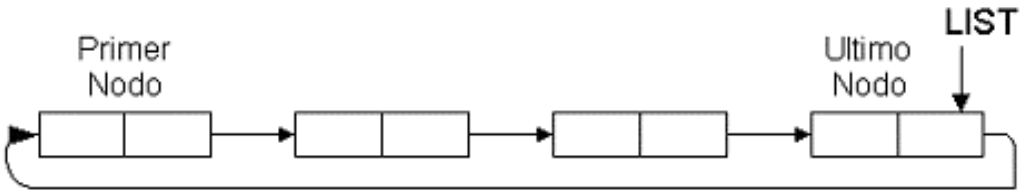
Retomando el problema de la inserción de un elemento en el medio de la lista, pero en este caso con representación linkeada, si p es una variable puntero para un elemento dado de la lista, la inserción representa asignar un elemento, insertar la información y ajustar dos punteros (Figura 12). la cantidad de trabajo requerido en independiente del tamaño de la lista.



(Figura 12)

Aunque la lista en representación simplemente linkeada es una estructura de datos muy útil, sin embargo tiene algunas desventajas. Una de ellas es que dada una variable puntero a un elemento en dicha lista, no se puede alcanzar cualquier otro elemento que preceda. Si se recorre una lista, el puntero original al comienzo de la lista debe ser guardado con el fin de que sea posible hacer referencia a esta lista de nuevo.

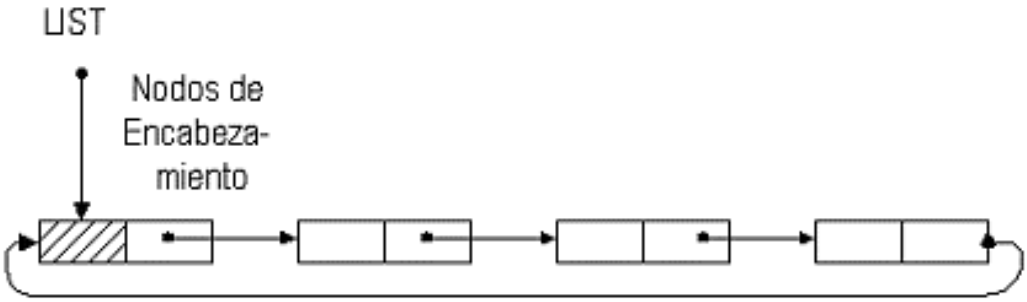
Una posibilidad para solucionar este problema es cambiar el campo link en el último elemento, de tal manera que contenga la dirección del primer elemento en vez de ser el puntero nulo. La lista en este caso recibe el nombre de lista circular. Desde cualquier punto en esta lista es posible alcanzar o llegar a cualquier otro punto de la lista. Una convención muy útil es la de permitir que el puntero externo de la lista circular apunte al último elemento, dejar que el elemento siguiente sea el primero (Figura 13), y que la variable puntero con el valor nulo representa una lista circular vacía. Sin embargo, como la lista es circular no sabemos cuándo se ha recorrido toda la lista al menos que exista otra variable puntero al primer elemento.



(Figura 13)

Una opción es el uso de un nodo de encabezamiento, que simplemente es un nodo extra al frente de la lista. Dicho elemento no representa un elemento en la lista, y el campo de información puede ser utilizado para mantener datos globales con respecto a toda la lista.

Por ejemplo, el número de elementos, o el valor de un puntero al elemento actual durante el proceso de su recorrido. Para el caso particular de una lista circular puede contener un valor no válido o una bandera marcando el comienzo, y así la lista puede ser recorrida usando una sola variable puntero que se detiene cuando se alcanza el nodo de encabezamiento. Dicha variable puede apuntar inicialmente al nodo de encabezamiento (Figura 14).



(Figura 14)

Aún cuando una lista circular con nodo de encabezamiento es una mejora con respecto a una lista común simplemente linkeada, ésta todavía tiene alguna desventaja. No se puede atravesar esta lista en dirección contraria ni tampoco se puede eliminar un elemento de una lista circular dado solo una variable puntero a ese elemento. En el caso que se requiere más flexibilidad, la estructura de datos apropiada nuevamente es la lista doblemente linkeada. Estas listas pueden ser circulares o no, y contener o no un nodo de encabezamiento.

Ejercicios.

1. Escribir la secuencia de pasos básicos de un algoritmo, para determinar con el uso de una pila si una cadena x de entrada es válida, tal que conste de las letras 'A' y 'B', Y genere una cadena inversa w de salida.

2. Escribir la secuencia de pasos básicos de un algoritmo, para
que:

a) Concatene dos listas linkeadas.

b) Invierta una lista, de tal manera que el último elemento pase a ser el primero, y así sucesivamente.

c) Retorne la suma de los números enteros que contiene los elementos de una lista.

3. Indicar como representar1a, con el uso de listas linkeadas, una planilla de cálculo electrónica de 999x999 celdas. ¿Cuál seria la característica estructural de los elementos que le corresponden a cada celda no nula de la planilla, suponiendo que puede almacenar números enteros, fórmulas o textos como cadenas.

TEMA 3: Árboles.

Propiedades formales.

Estas estructuras de datos son ampliamente usadas en distintas áreas, tales como: árboles para análisis de textos (teoría de compiladores, y lenguajes formales), árboles de búsqueda (recuperación de información), y árboles de directorios (manipulación de archivos y sistemas operativos).

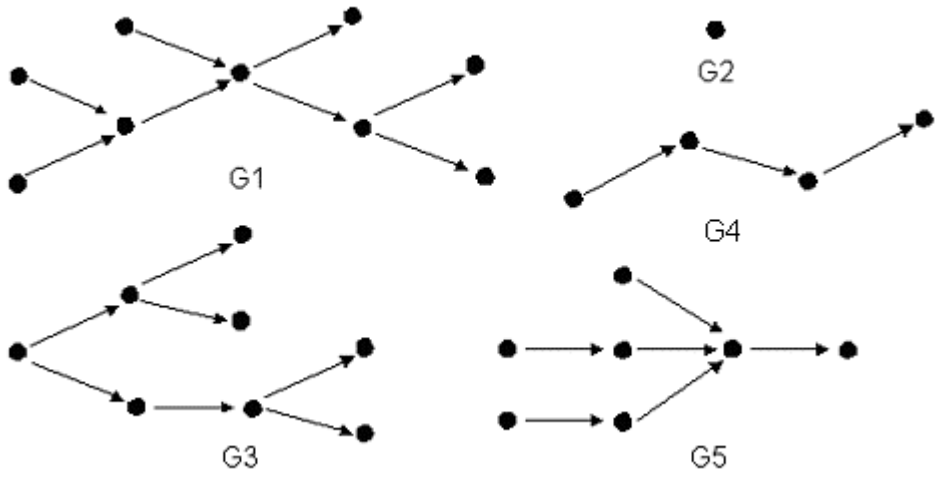
Según Pfaltz, un árbol es un grafo conectado finito G = (P, E), tal que:

$$|P| = |E| + 1$$

Como un corolario inmediato de esta definición, un grafo es un árbol, si y sólo si cada arco es un arco desconectante, es decir, su remoción desconecta el grafo (Figura 15).

A pesar de las definiciones previas, varios autores llaman sólo árboles a los grafos G3 y G4. Precisamente, esta forma más restringida de árbol es la que más frecuentemente se usa en aplicaciones de computación.

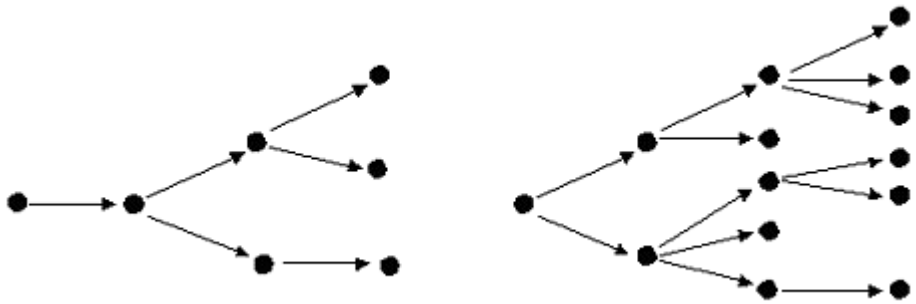
La definición de ideales principales es útil para caracterizar a esta clase limitada de árboles.



(Figura 15)

Dado un árbol T = (P, E), donde x Є P, llamamos subárbol principal derecho sobre x (Tx) al subgrafo R({x}). A partir de ello, T es un árbol principal derecho si para algún x Є P, T=Tx. Al punto x se lo llama el punto principal del árbol (Figura 16). El punto x es claramente minimal. Los puntos maximals son llamados los puntos extremos del árbol. Aunque los árboles podrían representarse gráficamente de cualquier forma, la mayoría de los autores ilustran los árboles creciendo hacia abajo.

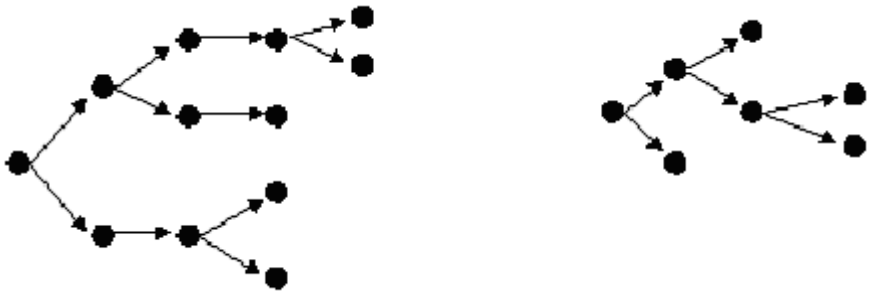
Por brevedad, asumiremos que los términos árbol y árbol principal derecho son sinónimos.



(Figura 16)

Árboles binarios. Representaciones secuencial y linkeada.

Los árboles están esencialmente caracterizados por la cardinalidad de grado de entrada, que es |L(y)| <= 1 para todo y, y por la cardinalidad del grado de salida para cualquier punto. Un árbol se llama árbol n-ario si para todo y Є P, |R(y)| <= n. En los casos especiales cuando n = 2 o n = 3, se llaman árbol binario o árbol ternario respectivamente. Un árbol es completo si |R(y)| = n para todos los puntos y no maximales (Figura 17), Y diremos que es lleno si además de ser completo, tiene sus maximales a la misma profundidad desde el punto principal.



(Figura 17)

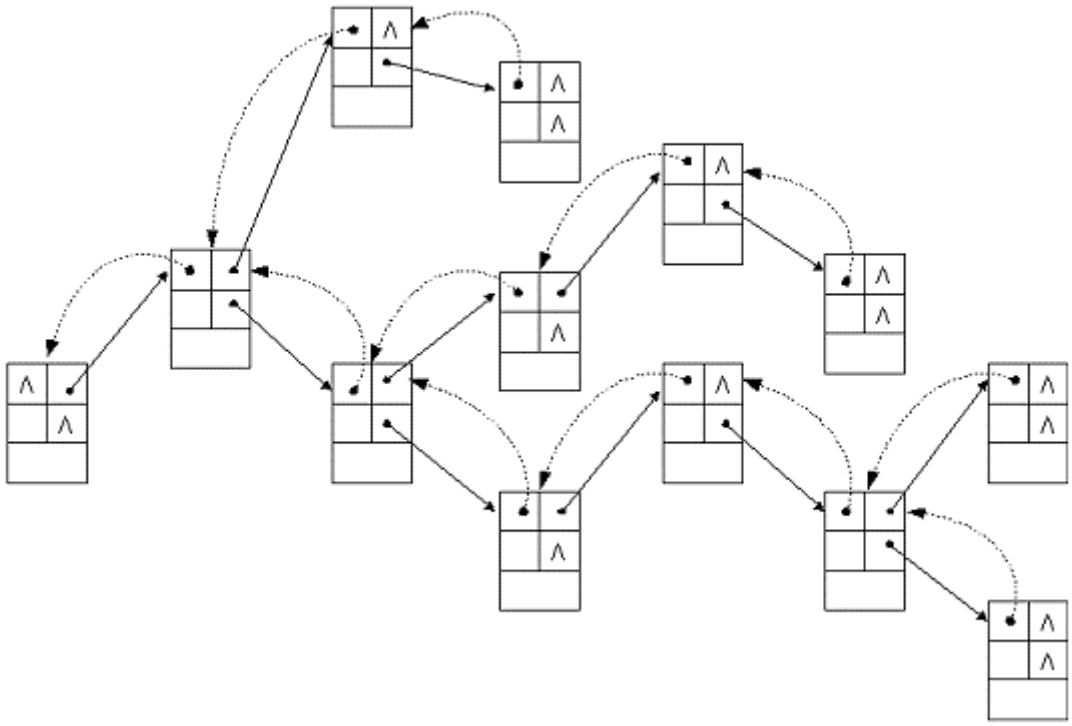
Los árboles binarios son particularmente fáciles de representar por una estructura linkeada, dado que no se necesitan más de tres campos link (Figura 18). Dos campos link se utilizan para linkear a los elementos que forman el conjunto R(y) (conocidos como los hijos de y), y uno opcional según la aplicación, al único elemento que forma el conjunto L(y) (conocido como el padre de y). La consideración importante a tener en cuenta es el orden impuesto por la aplicación a los hijos de un elemento dado, el que se basará en alguna propiedad del problema en cuestión. Justamente, los árboles binarios son especiales, porque cuando están ordenados se prestan a búsquedas, inserciones y eliminaciones rápidas. Tal es el caso del árbol binario de búsqueda, donde el subárbol de la izquierda contiene elementos cuyas claves son menores que la de la raíz, mientras que los de la derecha son mayores.

Por ejemplo, el proceso de búsqueda binaria, típicamente modela el ámbito de búsqueda de una clave sobre un arreglo con los elementos ubicados en orden ascendente o descendente. Esta técnica puede también ser abstractamente modelada por un árbol binario, explicitando en él, el orden de los elementos. Esta última técnica sobre un representación linkeada se valora cuando se trabaja con un conjunto dinámico de elementos. Un nuevo elemento ser sumado a una representación secuencial requeriría como promedio n/2 desplazamientos, mientras

que en una representación linkeda aquel será ubicado apropiadamente en el árbol binario explícito.

Desde otro punto de vista, si un árbol binario de búsqueda crece en una conducta dinámica, la desventaja se plantea debido a que la estructura del árbol se vuelve enteramente dependiente del orden en el cual los nuevos elementos ingresan a la misma.

Como hecho curioso, el caso más desfavorable sucede cuando los elementos ingresan ordenados, que de cualquier manera hace a la representación linkeada equivalente en perfomance a la representación secuencial.



(Figura 18)

Este ejemplo ilustra la siguiente regla general: las estructuras de datos linkeadas son generalmente más apropiadas para la representación de estructuras de datos dinámicas, mientras que las secuenciales tienden a ser superiores para estructuras de datos estáticas o casi estáticas.

En los lenguajes de programación, la implementación del árbol binario se puede efectuar a partir de la definición del tipo puntero al tipo de datos.

La representación secuencial de un árbol binario es efectiva si el mismo es estático y lleno. Si el árbol no está lleno, hay que sumar elementos extras vacíos, lo que para árboles de gran profundidad puede ser muy costoso en términos de almacenamiento.

Dado el punto principal x de árbol binario, primer elemento en la representación secuencial; y1, y2 Є R(x) segundo y tercer elemento; z1, z2 Є R(y1), y z3, z4 Є R(y2) cuarto, quinto, sexto y séptimo elemento respectivamente, y a sí sucesivamente; entonces, podemos representar un árbol binario secuencialmente.

Se puede luego definir la siguiente función de acceso a los elementos de la estructura: dado yi que define a un elemento con índice i en a representación secuencial, yj definirá al único elemento de L(yi) si j = i/2, e yk, ym los elementos de R(yi) si k=2·i y m=2·i+1. Se supone que i, j, k Y m deben satisfacer la condición:

$$0 < i, j, k, m <= |P|$$

Barridos de árboles binarios.

En la sintaxis de los lenguajes de programación, las expresiones computacionales se construyen a partir de la inserción de operadores binarios (+,-,*,1, **, etc.) entre dos operandos (variables, constantes, etc.). Esta forma de representar la expresión por una secuencia de operaciones computacionales se llama notación infija; un ejemplo típico podría ser la fórmula cuadrática para encontrar la raíz de una ecuación cuadrática:

(-B+SQRT(B**2-4*A*C))/2*A

En esta expresión parentizada es necesario especificar el orden de evaluación. Las expresiones alternativas sin paréntesis de esta expresión colocan el operador antes o después, más que entre los operandos, como se indica:

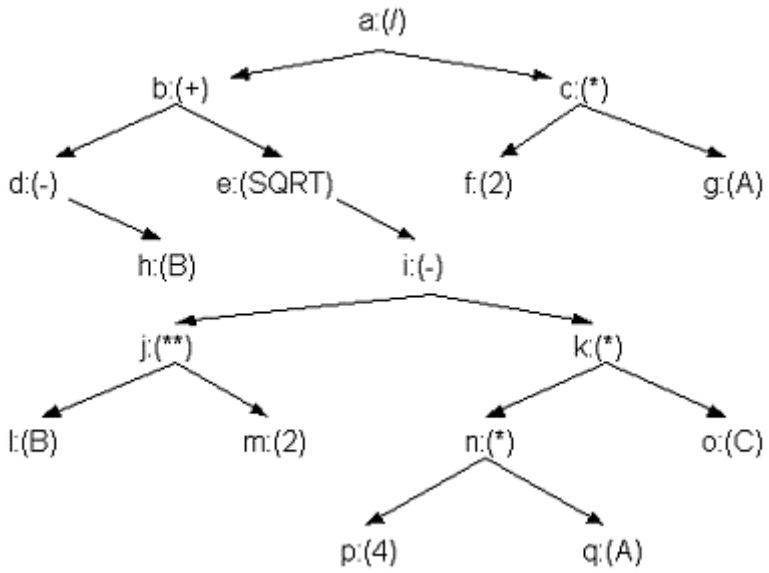
/ + - B SQRT - ** B 2 * * 4 A C * 2 A

B - B 2 ** 4 A * C * - SQRT + 2 A * /

A estas representaciones se les llama notación prefijo y notación postfijo, respectivamente. También la expresión infijo se puede representar como un árbol binario T (Figura 19), conocido como árbol de expresión.

El árbol de expresión T = (P, E) es realmente un grafo, de igual forma que lo son las expresiones anteriores. La diferencia es que son grafos lineales Li = (P, Ei), sobre el mismo conjunto de puntos.

Un barrido de un grafo G = (P, E) es cualquier relación lineal E' sobre P. Por lo tanto, las expresiones anteriores (realmente deberían observarse como simples cadenas) son barridos sobre T, como podría haberlo sido cualquier listado arbitrario de los puntos de P. Sin embargo, es preferible si el barrido E' refleja la estructura de arcos E del grafo original.

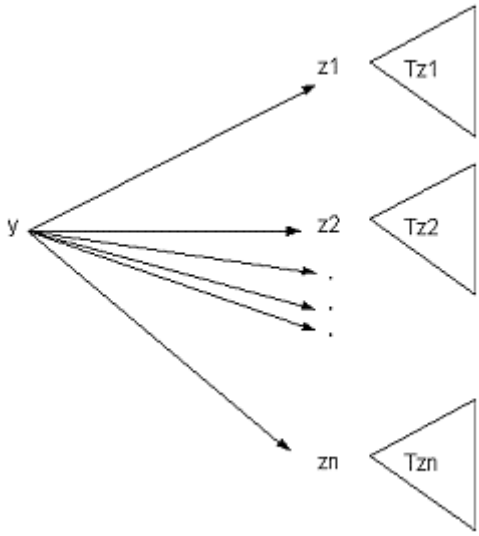


(Figura 19)

Previo a la definición de los tipos de barridos que nos interesan es conveniente redefinir el concepto de árbol desde el punto de vista recursivo.

Un subgrafo T = (P, E) es un árbol principal con punto principal y, llamado Ty

a) Si P = {y} es un simple elemento con Ep = 0, o
b) Si cada punto zi Є R(y) es el punto principal de Tzi en un conjunto Tzi de subárboles principales disjuntos (Figura 20).



(Figura 20)

Podemos definir recursivamente tres importantes barridos, en términos de un procedimiento establecido sobre el conjunto R(y):

- Barrido Preorden de $\overline{R}(y)$: a) Sumar y al orden lineal, siendo $\overline{R}(y)$= z1,z2,...,zn.
b) Barrer R(zl), R(z2),..., R(zn).

- Barrido Simétrico de $\overline{R}(y)$: a) Barrer $\overline{R}(zl)$,
b) Sumar y al orden lineal,
c) Barrer R (z2) .

- Barrido Postorden de $\overline{R}(y)$: a) Barrer $\overline{R}(z1)$, $\overline{R}(z2)$,..., $\overline{R}(zn)$,
b) Sumar y al orden lineal.

Justamente, a partir de estos barridos es posible la obtención de las cadenas de la expresión cuadrática de introducción al tema.

Los barridos son frecuentemente utilizados para linealizar una estructura de dato no lineal, para hacerla compatible con algún proceso secuencial sobre ella. El orden secuencial de evaluación de expresiones computacionales es un ejemplo.

Usando las herramientas recursivas del lenguaje Pascal, es posible generar estos barridos para la obtención de la representación lineal, dado un árbol binario. Realmente, la mayoría de los algoritmos o procesos que utilizan árboles binarios se realizan en dos fases.

En la primera se construye un árbol binario, y en la segunda se barre el árbol. Una aplicación típica es la construcción del árbol binario de búsqueda, y al mismo aplicarse barrido simétrico para obtener los elementos en orden ascendente. Nuevamente aquí la eficiencia del método depende del orden original de los elementos. El orden de complejidad es logarítmico en el mejor de los casos, y cuadrático en el peor.

Tipos de árboles binarios.

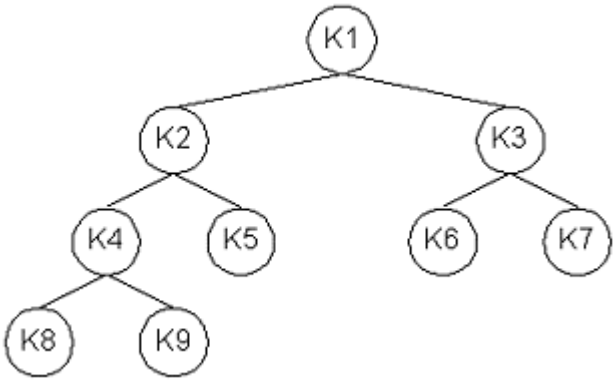
Hay un número interesante de variantes de las estructuras de datos árboles que se usan en algoritmos y aplicaciones. Estas presentan importantes propiedades, y frecuentemente habilitan algoritmos que poseen sorprendente eficiencia en tiempo de ejecución. Entre estas estructuras de datos árboles se reconocen como más destacadas los heaps, los árboles de búsqueda binaria y los árboles B.

La palabra heap se usa en dos sentidos en la literatura. En un caso se usa para determinar a los árboles binarios completos cuyos nodos contienen claves de búsqueda, los que están organizados en orden descendente en todos los pasos desde la raíz a los maximals. El término también se utiliza para reconocer la zona de memoria en la que se ubican nodos multilinkeados de tamaño variables.

Siendo k1, k2,..., kn un conjunto de claves sobre las que existe un orden total, es posible asignar las mismas a los nodos de un árbol completo en un orden por nivel y de izquierda a derecha (Figura 21). Justamente, definimos a tal árbol binario como un heap si dada una clave Ki en cada nodo, en el árbol es mayor que o igual a las claves k2i y k2i+1 de sus nodos hijos. Definido de esta manera un heap, cada subárbol de unheap es un heap, y por transitividad la clave en la raíz de un heap es la clave mayor.

Esta es la base para establecer una serie de algoritmos propuestos por distintos autores para la manipulación de los heaps, fundamentalmente cuando los

nodos contienen claves que no satisfacen la propiedad heap, o existe una conducta dinámica de inserción de nodos y/o de eliminación de elementos desde el nodo raíz. El algoritmo heapsort es un ejemplo característico de este tipo de árboles binarios.



(Figura 21)

El árbol de búsqueda binario, anteriormente presentado, es un árbol binario que satisface la siguiente propiedad: en cada nodo Ni de clave ki, todas las claves en los nodos en el subárbol izquierdo del nodo Ni son menores que ki, y todas las claves en el subárbol derecho del nodo Ni son mayores.

Dada una clave k, el proceso de búsqueda para k en un árbol de búsqueda es directo. Si k es igual a la clave del nodo raíz, la búsqueda termina exitosamente. De otro modo, si k es menor que la clave en el nodo raíz, la búsqueda continúa en el subárbol izquierdo; mientras que si k es mayor que la clave en el nodo raíz, la búsqueda continúa en el subárbol derecho. Es proceso se repite en forma recursiva hasta encontrar la clave con éxito o no.

En algunas aplicaciones, un árbol binario puede tener un número fijo de nodos, sin permitirse inserciones y eliminaciones. A tal árbol binario de búsqueda se lo llama estático. Un ejemplo es aquel utilizado para suministrar un índice de búsqueda para los códigos de operación de un ensamblador, ya que los mismos permanecen fijos a través de todo el ensamblado.

Para un árbol de búsqueda binario es fundamental mantener el tiempo de búsqueda tan pequeño como sea posible. Distintos autores han propuesto una serie de algoritmos para minimizar el mismo cuando el árbol es estático, en algunos casos estableciendo distribuciones no uniformes dependientes de la frecuencia de uso de las claves.

Para el caso de los árboles de búsqueda binarios dinámicos, un constante objetivo ha sido minimizar dicho tiempo después de las inserciones o eliminaciones, manteniendo el árbol tan balanceado como sea posible.

Algunas consideraciones adicionales de los árboles binarios de búsqueda se plantearán en el TEMA 4.

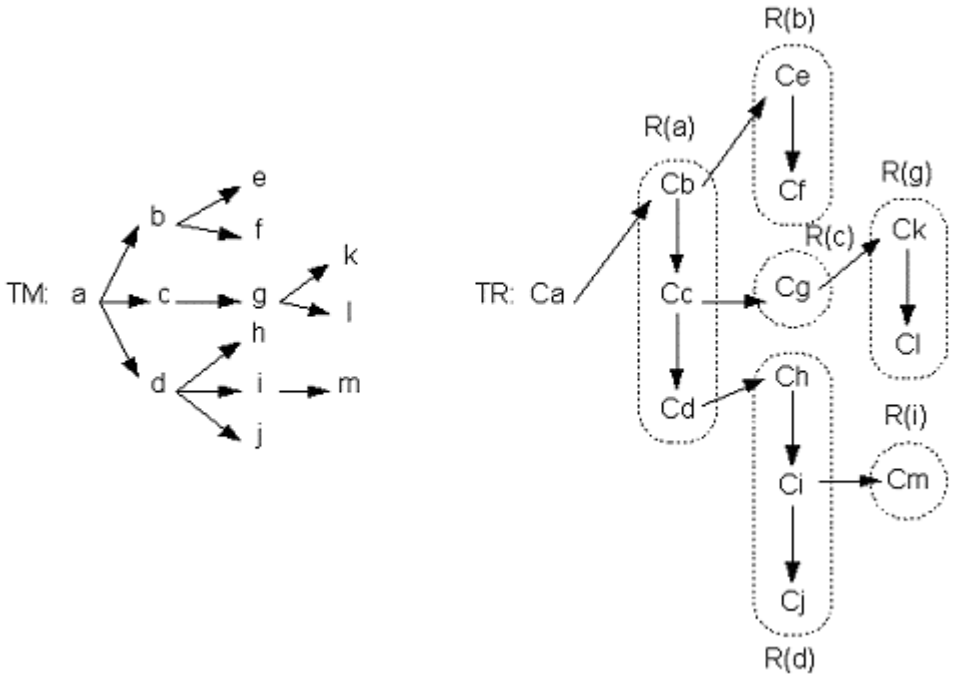
Árboles N-arios. Transformada de Knuth.

Un árbol es n-ario si para todos los puntos y; |R(y)| <= n. Una representación linkeada directa es fácil; simplemente habrá que suministrar a los elementos tantos

campos links como sean necesarios. Sin embargo, si la cardinalidad de n es muy grande, lo más probable es que se pierda una cantidad considerable de almacenamiento. Alternativamente, se podría intentar el uso de elementos que contengan sólo los campos link no nulos, cuyo número con estructuras dinámicas es algo indeterminado.

De igual forma, tal como se acaba de definir un árbol n-ario, sería impracticable el barrido simétrico, si fuera necesario su uso, como así también de mayor complejidad los algoritmos para su manipulación. El deseo evidente es disponer de un método de representación que emplee elementos de tamaño fijo, con un número limitado de campos links, que no obstante puedan representar árboles principales de grado arbitrario.

Dados los puntos z1, z2,...,zn que pertenecen al R(y) y observados como un conjunto, es posible encadenar las celdas que les corresponden como una simple lista linkeada. Ahora, la relación entre un nodo y sus hijos se puede representar por un simple link que indica el primer punto en el conjunto R(y) (Figura 22).



(Figura 22)

Si el árbol n-ario está ordenado, los puntos pueden ser linkeados en la misma secuencia ordenada como en el R(y), de lo contrario en cualquier orden.

La transformación precedente se conoce con el nombre de transformada de Knuth. Después de su aplicación cualquier árbol n-ario tendrá en cada punto no más que dos campos links, justamente como un árbol binario. Por eso, se puede decir que la transformada de Knuth mapea el conjunto de los árboles principales derechos en el conjunto de los árboles binarios. Así, con leves modificaciones debido a la manipulación diferente de los campos links, las rutinas de barridos para los árboles binarios son inmediatamente aplicables a las representaciones de Knuth de árboles principales arbitrarios.

De igual forma, los algoritmos generales planteados a árboles binarios tendrán su aplicación, salvando la posibilidad que la inserción o eliminación de puntos distorsione las propiedades de la estructura original.

Ejercicios.

1. Indique cuál es el máximo número de nodos de nivel n en un árbol binario, y un árbol n-ario.
2. Indique cuál es la cantidad de árboles que se pueden formar con n nodos, y un nivel máximo m.
3. Indique como implementaría el juego ta-te-ti con el uso de la estructura de datos árbol. ¿Por qué se puede considerar que haya programas para jugar ta-te-ti de una forma perfecta, pero no para el juego de ajedrez o de damas?

TEMA 4: Acceso de Datos

Tablas de símbolos. Costos de acceso.

Un problema clásico en computación, es obtener un valor f(x) relacionado funcionalmente con un valor x dado. La evaluación de la función puede observarse como un caso especial de problemas más generales de recuperación de información (dado un argumento x, recuperar un valor asociado f(x)). Si la relación funcional puede indicarse por una expresión algebraica, f(x) puede ser simplemente computada. Pero frecuentemente, la relación funcional es tal que aquella práctica es imposible. Como un simple ejemplo, dado x como el nombre de una ciudad y f(x) su población, es imposible computar la población de una ciudad desde su nombre. En ese caso la correspondencia funcional puede expresarse por una tabla que contenga dos arreglos indexados. En el arreglo argumento se busca el argumento dado, y la entrada correspondiente en el segundo arreglo obtiene el valor funcional. la codificación de un procedimiento para realizar el proceso puede considerarse un problema trivial, pero la implementación de uno rápido es el objetivo de este análisis.

Los métodos para analizar funciones arbitrarias, explícitamente definidas por medio de una tabla funcional, son variados dependiendo de la características de la misma. Normalmente los argumentos de la función son strings de caracteres simbólicos. Por esta razón, a la tabla de correspondencia funcional se le suele llamar tabla de símbolos. La terminología proviene del área de diseño de compiladores. Los analizadores, que leen un código de programa fuente, deben reconocer rápidamente una variedad de símbolos generados por el usuario, tales como identificadores de variables, rótulos de estamentos, y símbolos literales; determinar las propiedades asociadas que hayan sido declaradas con el símbolo; y pasar ambos al módulo de análisis sintáctico. Esto se logra con la construcción de una tabla de símbolos, en la que las propiedades se ingresan como fueron declaradas, y luego se verifica estas entradas en ocurrencias subsecuentes del símbolo. Ya que la velocidad de esta operación puede ser crítica para la velocidad total del compilador, se han desarrollado un gran número de técnicas para incrementarla.

De igual forma, los métodos para análisis de la tabla de símbolos dependen de las operaciones que se quieran establecer sobre ella. Por ejemplo, algunas tablas de símbolos (tales como una tabla de códigos de operación para un ensamblador), pueden ser estáticas, en el sentido que ellas solo se usan para la búsqueda de valores asociados con sus nombres, y nunca sujetas a inserciones o eliminaciones. Otras, pueden ser dinámicas, en el sentido que nuevos registros se insertan o eliminan durante su uso (tales como las tablas de símbolos para manipular definiciones de identificadores en lenguajes de programación estructurada como el Pascal a el C).

Así, para una tabla de símbolos de interés, puede aplicarse algún subconjunto de las siguientes operaciones:

- Recuperar los valores de algunos atributos de un nombre I dado
- Almacenar los valores de algunos atributos de un nombre I dado
- Insertar un nuevo registro consistente de un nombre y valores de atributos.
- Eliminar un registro que tenga un nombre I dado.
- Enumerar los registros en la tabla en orden alfabético de nombres.

Respecto a la organización de las tablas de símbolos, las representaciones posibles pueden ser al menos una de las siguientes:

- Secuencia ordenada: colocando los registros en orden secuencial, de acuerdo al orden alfabético de sus nombres.
- Árbol binario balanceado: creando un árbol binario de búsqueda AVL a partir de los nombres, y colocando los registros en los nodos o punteros a los componentes de registros que contengan valores de atributos en los nodos.
- Lista linkeada: linkeando los registros juntos en una lista según el orden de sus nombres.
- Tabla Hash: almacenando los registros en una tabla hash con hashing realizado sobre sus nombres.

La elección de cual de estos métodos se usará, está basada en un examen de los costos para realizar las operaciones requeridas (Figura 23). En algunas aplicaciones, es conveniente combinar las ventajas relativas, creando un representación híbrida.

	Acceso Aleatorio Recuperación / Actualización	Inserción / Eliminación	Enumeración Secuencial
Secuencia Ordenada	O(log n)	O(n)	O(n)
Árbol Binario Balanceado	O(log n)	Constante	O(n)
Lista Linkeada	O(n)	Constante	O(n)
Tabla Hash	Constante	Constante	O(n log n)

(Figura 23)

Árboles de búsqueda. Árboles balanceados. Árboles AVL.

Supongamos la necesidad de representación de una tabla de símbolos, donde los argumentos de la función serán strings, y la relación funcional dinámica, tal que podrán insertarse o eliminarse elementos durante la ejecución. Finalmente, que los valores funcionales serán enteros.

El mantenimiento y evaluación de la tabla de símbolos, podría llevarse acabo por tres procedimientos separados. La función BUSQUEDA(argumento) buscará en la tabla de símbolos la entrada correspondiente al argumento, y si la encuentra, retornará el valor funcional asociado. Como es común, adoptaremos el nombre de clave para el argumento. Justamente, a partir de esta definición, la tabla de símbolos puede definirse abstractamente de la siguiente manera:

TABLA = {(clave, valor) / clave Є strings simbólico y valor a cualquier conjunto V}

mientras que:

$$\text{BUSQUEDA} = \begin{cases} \text{Valor, si (clave, valor) Є TABLA} \\ \text{Vacío, si no es así} \end{cases}$$

Los procedimientos INSERCIÓN (clave, valor) y ELIMINACIÓN (clave, valor) insertarán y eliminarán respectivamente, entradas en la tabla. Si la tabla se representa por dos arreglos secuenciales, los procedimientos BUSQUEDA, INSERCION y ELIMINACION tienen las características de implementación ya evaluadas. Consideremos que L1, L2, I y E indican los costos esperados de ejecutar una búsqueda exitosa que encuentra el valor en la tabla, una búsqueda no exitosa que retorna el valor nil, la rutina INSERCION, y la rutina ELIMINACION, respectivamente.

Dicho costo, puede aproximarse al costo real considerando al mismo, como el número de comparaciones o accesos de almacenamiento. Por ello, L1, L2, I y E, expresaran el número esperado de accesos de almacenamiento como una función del tamaño de la tabla de símbolos. Si asumimos que el arreglo de claves esta desordenado, L1= n/2, L2= n, 1=1 y E=2, independientemente del tamaño de la tabla. Al definir V1, V2, V3 y V4 como el número de veces que cada uno de estos procesos se ejecuta respectivamente, el costo total de mantener y buscar en la tabla de símbolos puede expresarse como:

$$\begin{aligned} \text{Costo (tabla desordenada)} &= V1.L1 + V2.L2 + V3.I + V4.E \\ &= (n/2).V1 + n.V2 + V3 + 2.V4 \end{aligned}$$

Con un planteo similar, puede deducirse que el costo total de mantener y buscar en un tabla ordenada es:

$$\text{Costo (tabla ordenada)} = n/2.\ (V1 + V2 + V3 + V4)$$

Es común que si la clave no se encuentra en la tabla, ella se ingrese inmediatamente. Por ello,

$$\begin{aligned} \text{Costo (tabla ordenada)} &= (n/2).V1 + n.V2 + (n/2).V4 \\ &> (n/2).V1 + (n + 1).V2 + 2.V4 \\ &= \text{Costo (tabla desordenada)} \end{aligned}$$

Este ejercicio académico, define esencialmente que el costo para esta representación es una función lineal de n. Para algunos problemas, n es razonablemente pequeño y la representación secuencial desordenada tendrá un costo aceptable. En otros casos, el costo será inaceptable. Aunque debe reconocerse que hay una pequeña trampa teórica en el análisis. En este planteo existe un costo menor de la tabla desordenana respecto a la ordenada. Está en claro que se paga el costo del ordenamiento, pero no se aprovechan las ventajas al momento de la búsqueda (haciendo búsqueda secuencial y no búsqueda binaria).

La opción que presentaremos será la representación linkeada de la tabla de símbolos en forma de árbol, aunque un análisis de su conducta la obtendremos al evaluar su crecimiento dinámico a partir de una entrada de texto natural (figura 24). El árbol desbalanceado indicado tiene 72 puntos, y la longitud de paso promedio es

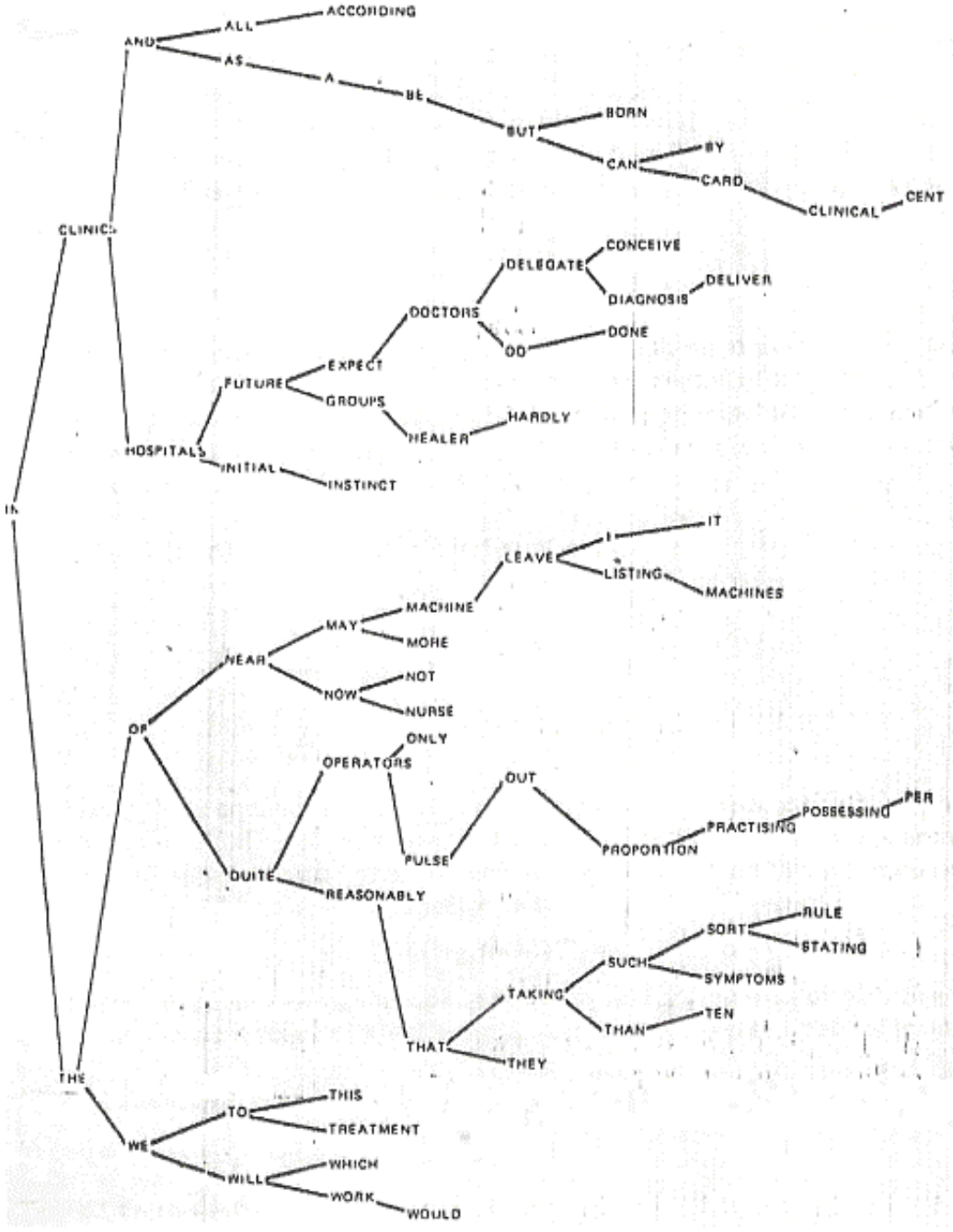
5,375. A la longitud de paso se le llama comúnmente profundidad del punto y en el árbol, y simbolizada por d(y). Para encontrar el elemento y se requieren d(y)+1 accesos de almacenamiento y comparaciones. Por ello, el número de puntos que deben ser examinados para buscar una entrada en la tabla de símbolos es 6,375. Esto es mucho mejor que un costo esperado de 72/2=36 accesos de almacenamiento para una búsqueda secuencial.

Sin embargo, la longitud de paso promedio será mínimo, y por lo tanto, el número de veces L1 y L2 será óptimo, en un árbol binario completo y balanceado. Un árbol es balanceado si para todos los puntos y con R(y) = {z1, z2}, ||R(z1) | - | R(z2)|| <= 1 (Figura 25).

Por ello, los costos de búsqueda L1 y L2 en términos de accesos de almacenamiento, para una tabla de símbolos estructurada en un árbol binario completo y balanceado, teniendo en cuenta que ello requiere k+1 accesos de almacenamiento para barrer un paso de longitud k, pueden establecerse a partir de las siguientes expresiones:

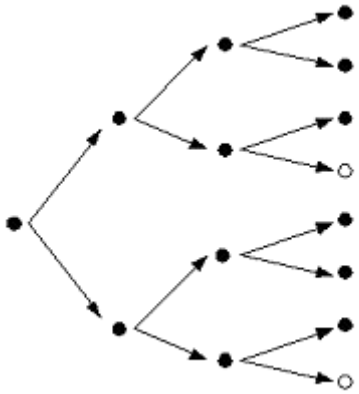
$$L1 = [\log(n+1)] - 1 \qquad L2 = [\log(n+1)]$$

In the clinics and hospitals of the near future we may quite reasonably expect that doctors will delegate all the initial work of diagnosis to machine operators as they now leave the taking of a pulse to a nurse. Such machine work may be only listing of symptoms, but I can conceive machines which would sort out groups of symptoms and deliver a card stating the diagnosis and treatment according to rule. It would not do the work done by the clinical instinct of the born healer, but the proportion of practising doctors possessing this instinct can hardly be more than ten per cent.]



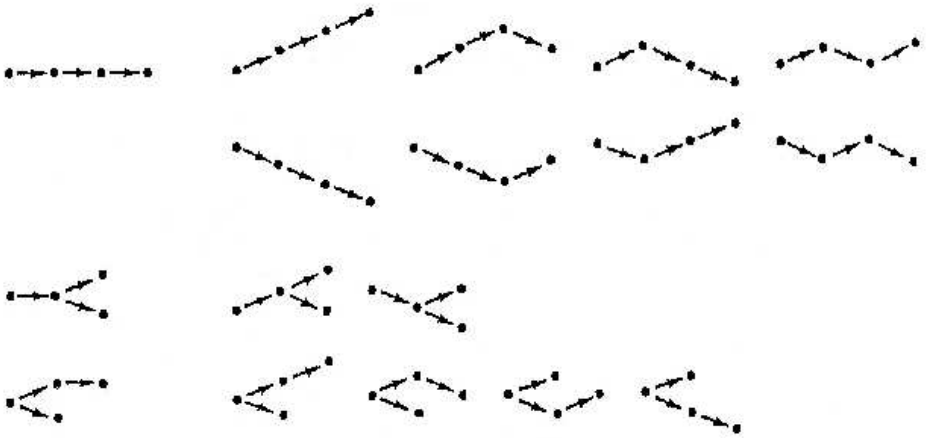
(Figura 24)

Los costos esperados de inserción I y de eliminación E son mucho más difíciles de calcular. Los costos de simplemente insertar un elemento a un árbol binario es despreciable, y para eliminarlo levemente mayor. Sin embargo, cada una de estas acciones con seguridad originará un árbol desbalanceado. Dependiendo de la aplicación, el costo de mantener un árbol de búsqueda dinámico en balance, puede efectivamente compensar las ventajas de la búsqueda óptima propuesta.



(Figura 25)

La conclusión anterior sugiere que para varias aplicaciones de tablas de símbolos, los árboles desbalanceados son eficaces. Realmente, como trabajo previo debería determinarse cual es el valor de L1 y L2 para todos los árboles binarios posibles de n puntos. Dicha tarea no es sencilla, lo que queda a la vista al considerar el caso para n=4 (Figura 26). Se pueden contar las longitudes de los pasos, para establecer que la longitud de paso esperado para los árboles binarios ordenados posibles es 1,3214. Para un caso general, cada representación se generará por una secuencia de n inserciones, cuya estructura depende del orden en que los valores ingresen. Por ello, si la tabla de símbolos se creara por el ingreso de un conjunto n de valores ordenados aleatoriamente, la longitud de paso esperada está en relación directa a la probabilidad de que se genere una representación dada. Del análisis estadístico se puede deducir que comparada con la longitud de paso esperada en un árbol balanceado completo de n puntos, la longitud correspondiente a árboles desbalanceados no es demasiado mala. Esto favorecería la representación con árboles no balanceados; más aún si se tiene en cuenta que los costos I y E son un poco mayores que los de ubicar un elemento.

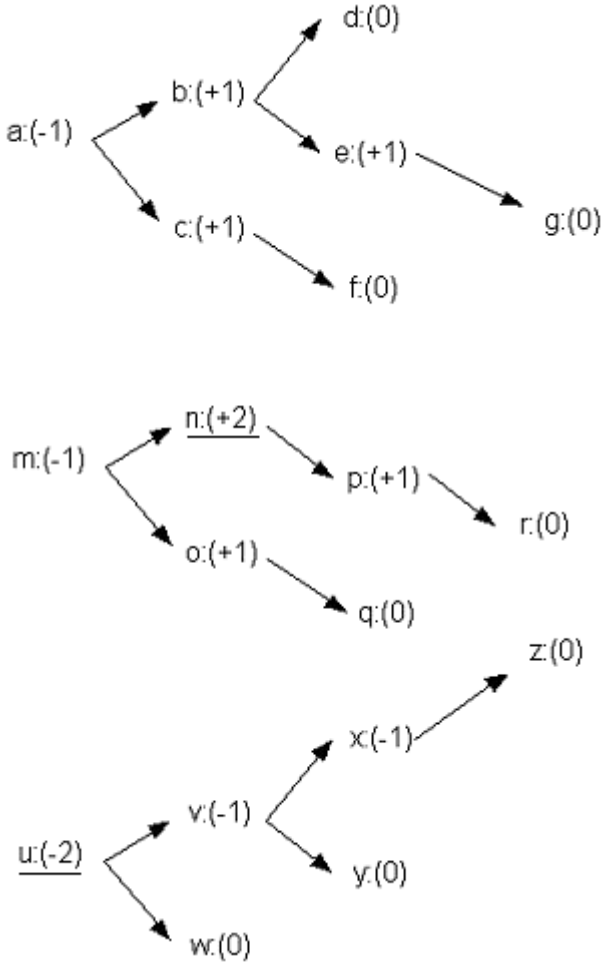


(Figura 26)

Como conclusión entonces, si se sabe que las claves de los elementos a insertar tienen un orden razonablemente aleatorio, un árbol binario desbalanceado puede considerarse una estructura atractiva de recuperación. Sin embargo, si las claves muestran un orden pronunciado, será necesario el esfuerzo de mantener dinámicamente el árbol binario completamente balanceado.

La última opción, realmente fija un costo normalmente prohibitivo. El compromiso podría ser un árbol casi balanceado, llamado árbol AVL (Adelson, Velskii y Landis) o árbol altamente balanceado.

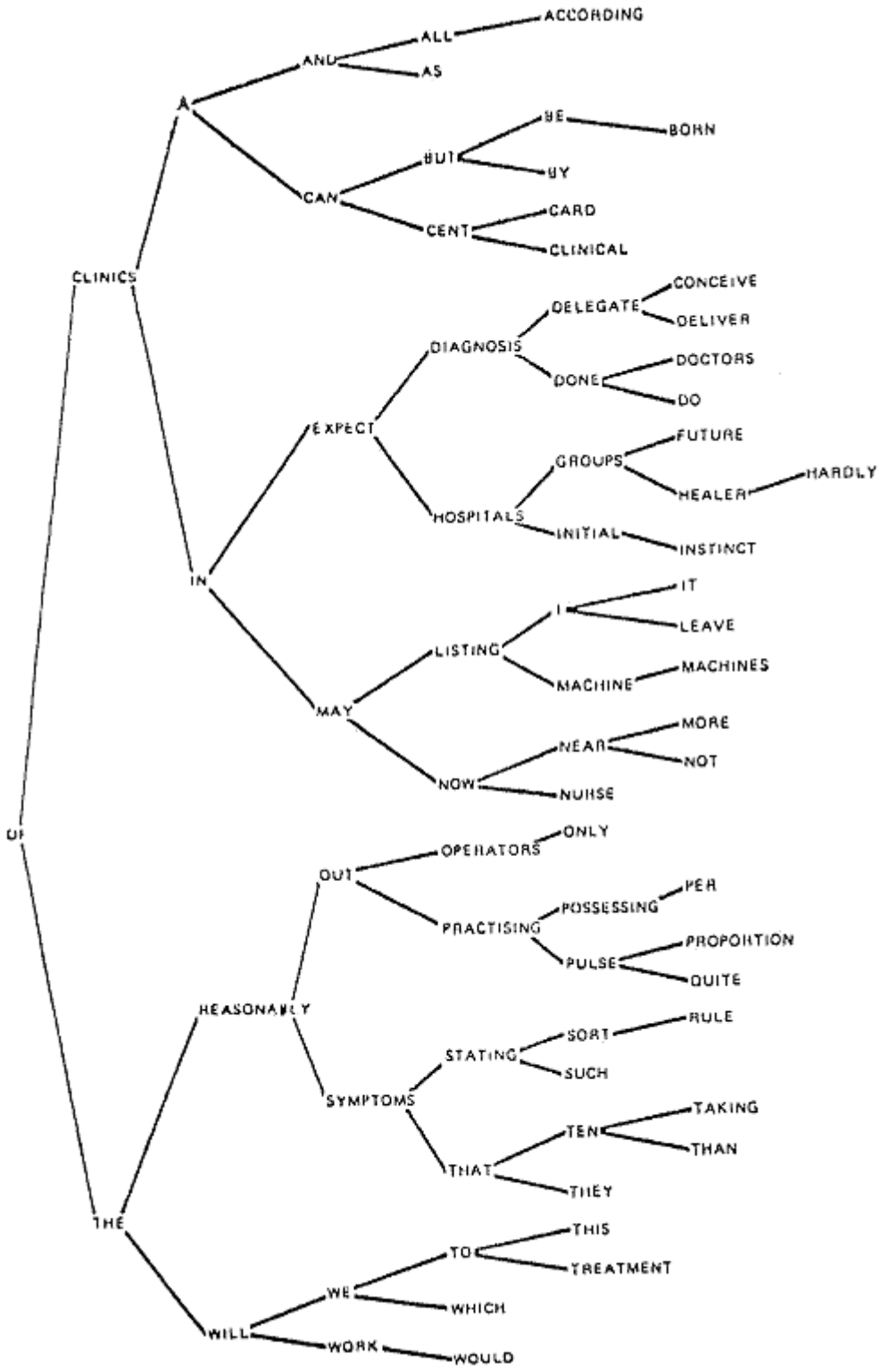
Si definimos a la longitud de paso mayor coma la altura, para un subárbol Tz1 con punto principal z1, y simbolizada por h(Tz1), un árbol se dice que es altamente balanceado o AVL si para toda y, |h(Tz2)-h(Tz1)| <= 1. Al resultado de la diferencia se le suele denominar el balance en el punto y, y cuyos valores podrán ser -1, 0 ó +1. Si el balance en un punto excede dicho valor, el misma está desbalanceado y se lo considera crítico (Figura 27).



(Figura 27)

La búsqueda en un árbol AVL es idéntica a la de cualquier árbol binario, salvo que los procedimientos deben reasignar valores de balance y observar aquellos puntos que se vuelven críticos. En aquel caso, la estructura local del árbol debe reconfigurarse. Justamente, dos características importantes de esta reconfiguración han establecido la popularidad del árbol AVL.

La primera es que la reconfiguración es local, ya que a los sumo tres o cuatro arcos deberán alterarse. La segunda es que a lo sumo una reconfiguración es necesaria para rebalancear el árbol entero. Finalmente, es conveniente observar los resultados de aplicar estos conceptos del árbol AVL sobre el mismo texto de introducción al tema (Figura 28).



(Figura 28)

Árboles de búsqueda óptimos.

Hasta ahora, todas las consideraciones realizadas sobre la forma de organizar árboles de búsqueda, se han hecho sobre la hipótesis básica de que todos los nodos tienen la misma frecuencia de acceso. Esta es la hipótesis que se puede hacer cuando no se conoce la distribución de accesos. Sin embargo, existen casos en los que se dispone de información sobre las probabilidades de acceso a los nodos individuales. En estos casos, es una característica que se mantenga una estructura constante. Un ejemplo típico es el analizador de léxico de un compilador que determina si una palabra (identificador) es una palabra clave (palabra reservada) o no. En este caso, se puede obtener información bastante aproximada sobre la frecuencia relativa con que se presentan las claves individuales y, por lo tanto, sobre su frecuencia de acceso, realizando medidas estadísticas en la compilación de cientos de programas.

Como ejemplo, considérese el conjunto de claves 1, 2, 3, con probabilidades de acceso p1=1/7, p2=2/7 y p3=4/7. Estas claves pueden organizarse como árboles de búsqueda de 5 maneras diferentes (Figura 29). Las longitudes de camino ponderadas, establecidas como la suma de todos los pasos de la raíz a cada nodo multiplicados por la probabilidad de acceso a cada nodo, para cada una de las configuraciones es:

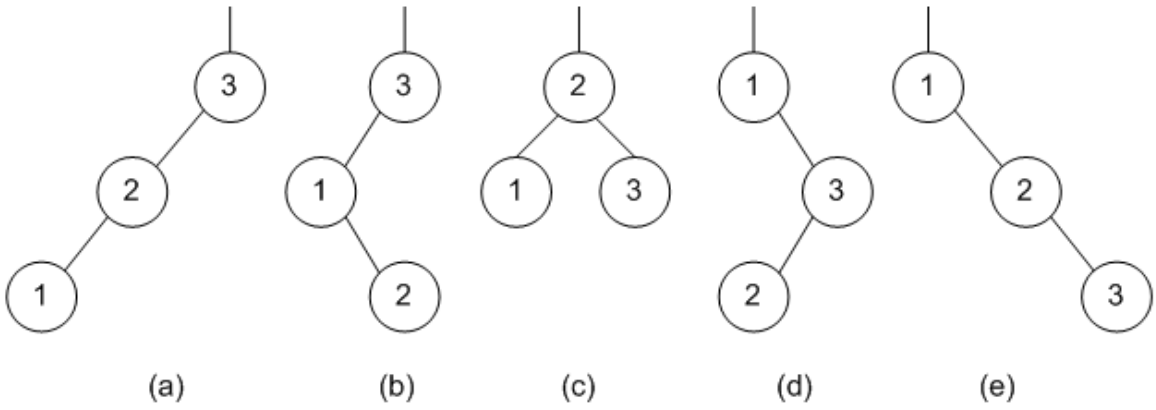
$$C^{(a)}_I = 1/7(1\cdot 3 + 2\cdot 2 + 4\cdot 1) = 11/7$$

$$C^{(b)}_I = 1/7(1\cdot 2 + 2\cdot 3 + 4\cdot 1) = 12/7$$

$$C^{(c)}_I = 1/7(1\cdot 2 + 2\cdot 1 + 4\cdot 2) = 12/7$$

$$C^{(d)}_I = 1/7(1\cdot 1 + 2\cdot 3 + 4\cdot 2) = 15/7$$

$$C^{(e)}_I = 1/7(1\cdot 1 + 2\cdot 2 + 4\cdot 3) = 17/7$$



(Figura 29)

En este ejemplo el árbol perfectamente balanceado no es la organización óptima, sino el árbol del caso a.

Sin embargo, el problema del analizador léxico sugiere inmediatamente que el problema debe contemplarse con una óptica más amplia, ya que las palabras que aparecen en el texto no siempre son palabras claves, y por lo tanto, debe generalizase el problema teniendo en cuenta las búsquedas infructuosas.

El árbol de búsqueda cuya estructura de el mínimo costo, entre todos los árboles con un conjunto de claves dado ki, y probabilidades pi <de acceso a cada nodo> y qi <de que un argumento a buscar se encuentre entre las claves ki y ki+1>, recibe el nombre de árbol óptimo.

Para encontrar el árbol óptimo, no hace falta que las probabilidades p y q sumen la unidad. De hecho, se determinan frecuentemente por medio de experimentos en los que se cuenta el número de accesos a los nodos.

Considerando que el número de configuraciones posibles de n nodos crece exponencialmente, parece poco menos que imposible poder encontrar el árbol óptimo cuando n es grande. Por ello, distintos algoritmos serán eficaces en cuanto a su resultado, cuando el n no sea muy grande. Por el contrario, otros en vez de buscar el árbol óptimo, simplemente prometerán encontrar un árbol casi óptimo. Pueden, por lo tanto, estar basados en principios heurísticos.

Hashing. Funciones de Transformación. Colisiones.

Dada una clave de recuperación k, los procedimientos de búsqueda anteriores encuentran el valor asociado f(k), a través del barrido de una estructura lineal o un árbol binario, en el registro que representa el elemento (k, f (k)). Cuando el conjunto de claves es demasiado grande, el costo de búsqueda debe necesariamente incrementarse, aún aunque se adopten varias técnicas para minimizar el ritmo de crecimiento. La técnica de recuperación hashing, idealmente elimina el tiempo de búsqueda.

La técnica del código hash es completamente sencilla, y aunque las claves pueden representar strings simbólicos o algún otro conjunto de valores, en realidad todas las claves adoptan en la computadora un valor entero. Así entonces, podemos definir una función arbitraria h(k) sobre el conjunto S de claves, llamada función hash, la que mapea S en el intervalo de enteros [Li, Lf]. Note que h puede ser cualquier clase de función sujeta sólo a la condición que h:-->[Li, Lf].

Cuando un elemento (k, f(k)) se inserta en la tabla de símbolos o sistema de recuperación, debe ocupar un espacio de almacenamiento del bloque [Li, Lf]. Así, dada la clave k, el procedimiento de búsqueda necesita solo reevaluar la función h(k) para obtener la posición del registro que estará vacío, en cuyo caso f(k)=vacío, o representa el elemento deseado (k, f(k)). Es decir, ninguna búsqueda en principio sería necesaria.

Sin embargo, es completamente posible que dos diferentes claves k1 y k2 en S, puedan tener la misma posición, h(k1)=h(k2). Lo imposible, es que el mismo registro contenga ambos elementos (k1, f(k1)) y (k2, f(k2)). A tal situación se le llama una colisión. Una función hashing h correctamente elegida, puede minimizar la probabilidad de colisiones, pero no eliminarlas. Igualmente, debe incluir algún mecanismo para resolver las colisiones. La primera característica podría obtenerse al considerar una función h que sea la antitesis de una función continua. De hecho, esto ha sido la causa de la considerable investigación sobre el tema.

Comentaremos aquí solo dos de las varias técnicas conocidas para seleccionar funciones hash. Uno de los métodos usa la división y el otro la multiplicación. El método de la división, simplemente obtiene h(k) del resto de k después de dividirlo por el tamaño de la tabla M:

$$h(k) = k \bmod M.$$

Sin embargo, debería elegirse M más cuidadosamente. Por ejemplo, el método de la división con un valor primo próximo al tamaño de la tabla, ha obtenido en distintos análisis la mejor perfomance promedio respecto a otras técnicas.

El método de la multiplicación podría ejemplificarse con una de las técnicas más interesantes. Suponiendo la relación:

$$O = \sqrt{5-1/2} = 0.61803399,$$

y el examen de su multiplicación con i, para 1 <= i <= 20, la parte fraccional de estas multiplicaciones, y el primer dígito de 10 veces la parte fraccional de las mismas (Figura 30), es posible observar que los primeros diez valores de la última columna son una permutación de los números 0, 1,...,9. Esta es la base para establecer una propuesta por el método de la multiplicación, con la siguiente fórmula general:

$$h(k) = M.(((A/w).k) \bmod 1)$$

donde M es el tamaño de la tabla, w es el tamaño de la palabra de la computadora, y A es primo de w.

i	Múltiple $i\Phi^{-1}$	Parte fraccionaria $\{i\Phi^{-1}\}$	Piso de 10 x parte fraccionaria $[10 \times \{i\Phi^{-1}\}]$
1	0.618034	.618034	6
2	1.236068	.236068	2
3	1.854102	.854102	8
4	2.472136	.472136	4
5	3.090170	.090170	0
6	3.708204	.708204	7
7	4.326238	.326238	3
8	4.944272	.944272	9
9	5.562306	.562306	5
10	6.180340	.180340	1
11	6,798374	.798374	7
12	7.416408	.416408	4
13	8.034442	.034442	0
14	8.652476	.652476	6
15	9.270510	.270510	2
16	9.888544	.888544	8
17	10.506578	.506578	5
18	11.124612	.124612	1
19	11.742646	.742646	7
20	12.360680	.360680	3

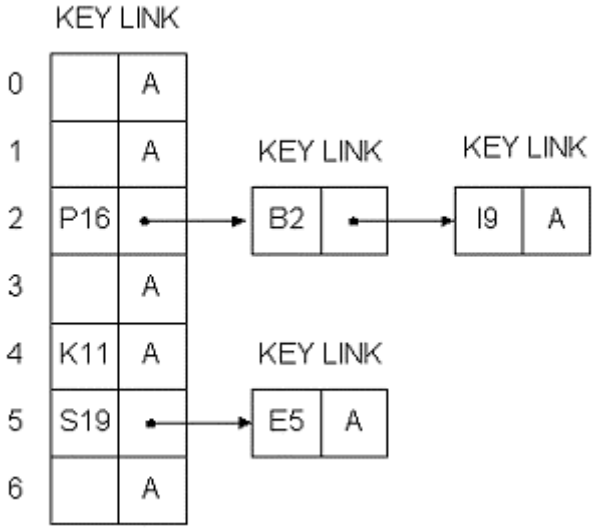
(Figura 30)

Respecto a la política de resolución de colisiones, tres técnicas han recibido más atención en la bibliografía: linkeado, el uso de buckets o compartimentos, y direccionamiento abierto.

Un camino para resolver colisiones es mantener M listas linkeadas, una por cada dirección posible en la tabla hash. A una clave K, le corresponderá una dirección h(k) en la tabla, donde encontraremos la cabeza de una lista que contendrá todos los registros con claves que han generado la misma dirección. Luego, la lista es inspeccionada para ubicar un registro que contenga la clave k. Por ejemplo, consideremos una tabla hash con siete registros numerados del 0 al 6, donde las claves son de la forma Ln donde n es el número que le corresponde a la posición de la letra L en el alfabeto (Figura 31). Además, que la función h es:

$$h(Ln) = n \bmod 7.$$

Así, h(P16) = 16 mod 7 = 2, valor de función que también le corresponde a las letras I y B, y que determina que estas claves se almacenen en la misma lista linkeada cuya cabeza comienza con la dirección 2. Si N claves van a ser almacenadas en las listas de una tabla de longitud M, el número promedio de claves en una lista es N/M. Por ello, la longitud de la lista promedio es completamente breve, y disminuye la cantidad promedio de trabajo para una búsqueda por un factor M, aunque es altamente probable la existencia de colisiones.



(Figura 31)

El uso de buckets, como técnica para resolver colisiones, consiste en dividir la tabla hash en J grupos de registros, de B registros cada uno, llamados buckets. La función hash computa h(k) para obtener un número de bucket desde la clave k, tal que la misma se almacena en el bucket cuyo número es h(k). Si un bucket está lleno, habrá que invocar a un rutina para resolver el overflow. Esta técnica también es útil para la búsqueda en archivos, a partir de la recuperación de un grupo de registros y el posterior seguimiento de un registro determinado con una técnica conocida como la búsqueda binaria, respecto a la búsqueda individual de registros en los archivos. Si se produce un overflow en un bucket, puede efectuarse un linkeado a un bucket de overflow. Este link puede ubicarse en el extremo del bucket que ha sufrido el overflow. Resulta claro que es necesario una elección adecuada

del tamaño de los buckets. El análisis de esta técnica permite indicar que el número promedio de accesos requeridos, como una función del porcentaje de ocupación, es muy bueno. Por ejemplo, se necesitan menos que tres accesos para todos los buckets con tamaños comprendidos entre B=1 a B=50, aún en un 95% de saturación de la tabla.

Finalmente, la técnica de direccionamiento abierto resuelve las colisiones por la búsqueda de una entrada vacía a la tabla hash, en una dirección distinta a la h(k), que le corresponde a la clave k. Al considerar V0 = h(k), y V0, V1,...,VM-1 una permutación de las direcciones de la tabla, llamada secuencia de prueba, se podría inspeccionar las entradas de la tabla en el orden dado por la secuencia de prueba, hasta encontrar un registro que contenga la clave k (en cuyo caso la búsqueda será exitosa), o hasta encontrar un registro vacío, o concluir que la tabla está llena (en cuyo caso la búsqueda no será exitosa).

El camino más simple para elegir la secuencia de prueba, o más genéricamente función secuencia de prueba p(k), es definir que p(k)=l para todas las claves. A este método de búsqueda se le llama por prueba lineal, ya que la secuencia para cualquer clave k es siempre de la forma:

h(k), k(k)-1, h(k)-2,..., 1, 0, M-1, M-2,..., h(k)+1.

Así, primero se inspecciona la entrada en la dirección h(k) para ver si contiene la clave k; sino la contiene la búsqueda continúa en orden decreciente hasta cerrar circularmente la tabla. El análisis demuestra que este método incrementa considerablemente los tiempos de inserción y búsqueda cuando la tabla se aproxima a la saturación total.

Otro método que define una secuencia de prueba distinta, es el denominado búsqueda por prueba pseudo aleatoria, donde la secuencia de prueba es de la forma:

h(k)+0, h(k)+r1, k(k)+r2,..., h(k)+rM-1,

siendo los números 0, r1,..., rM-1, permutaciones de los números 0, 1,..., M-1. Estos números pueden ser generados por un generador de números aleatorios.

La alternativa podría ser el método denominado búsqueda por residuo cuadrático, que utiliza una secuencia de prueba tal como:

h(k), h(k)+i·i, h(k)-i·i,...

con módulos reducidos al tamaño de la tabla, para valores de i crecientes en el intervalo 1<=i<=(M-1)/2. Para números primos adecuadamente elegidos, se puede garantizar que la secuencia de prueba es una permutación del espacio de direcciones de la tabla.

El hashing doble, considerado como una mejor opción que los anteriores, usa una secuencia de prueba de la forma:

h(k) – i·h2(k)

con módulo reducido al tamaño de la tabla, para 0 <= i <= M-1, donde h2(k) es una segunda función hash que da para la clave k un primo entero y próximo al tamaño de la tabla M. La diferencia entre este método y el de la prueba pseudo aleatoria, es que en el doble hash claves k y k' diferentes, producen secuencias de prueba posiblemente diferentes, aún aunque ellas colisionen en la misma dirección de tabla; mientras que en la prueba pseudo aleatoria y de búsqueda por residuo cuadrático, dos claves que colisionen en la misma dirección de tabla conducen a la misma secuencia de prueba.

De estas variaciones de direccionamiento abierto, los métodos de hashing doble son superiores, aunque sus rendimientos pueden mejorarse bajo circunstancias favorables.

La eficiencia de las técnicas de código hash, la que puede medirse en términos del número esperado de registros que deben examinarse en orden para recuperar un elemento deseado, es una función de tres factores básicos:

1. La distribución de las claves de búsqueda,
2. Las características de la función hashing h, y
3. La densidad de los valores hash h(k) en el rango potencial [L1, L2].

Un programador tiene normalmente poco control sobre el primer parámetro. La característica esencial de la función hashing es su propiedad de distribuir los valores hash h(k) de forma uniforme sobre el rango, lo que es usualmente posible con las técnicas existentes. De cualquier forma, el programador tiene la posibilidad de correr unas pocas simulaciones para determinar la uniformidad de la función elegida. El tercer parámetro es el más critico para determinar la eficiencia de la técnica, pero a su vez sobre el cual se tiene el mayor control.

Como en el sub tema anterior, los números L1, L2, I y E son funciones puras del número de elementos del sistema, salvo que con las funciones hash se pueden alterar dichos números incrementando o disminuyendo la cantidad de almacenamiento usado.

Consideremos a r como el número total de registros disponibles en el rango de la función hash h, que puede ser el número total de registros disponibles en el almacenamiento libre o el número total de buckets. A n como el número de elementos, cada uno con una clave distinta, que ingresará al sistema y se almacenará. Finalmente, como factor de densidad a la relación f = n/r.

Dada una clave particular k, habrá un conjunto de uno o más elementos que tengan el mismo valor hash h(k). Para buscar y recuperar el elemento (k, f(k)), al menos debe hacerse un acceso o prueba. Según la bibliografía, puede demostrarse suponiendo una distribución perfectamente uniforme h, que el número esperado de accesos es:

$$L1 = 1 + f/2$$

Algunos valores típicos de L1 como una función del factor de densidad f dará valores bastantes pequeños (Figura 32). Para los buckets, los valores de L1 son mayores que 2.

Aunque el número L1, para las función hash es largamente una función de la densidad f, otros factores también afectan la eficiencia:

* la distribución real de las claves,

* la secuencia particular en la que los elementos se insertan y son posteriormente recuperados, etc.

Φ	L1
0.1	1.05
0.5	1.25
0.75	1.38
0.9	1.45

Figura 32

Árboles de búsqueda n-arios. Árbol B.

Hay un área muy práctica en la que se utilizan árboles n-arios, y que se refiere a la construcción y el mantenimiento de árboles de búsqueda en gran escala, donde hay que hacer inserciones y eliminaciones, pero en los que la memoria principal es demasiado costosa, o no suficientemente grande, para ser utilizada como almacenamiento permanente.

Entonces, los nodos de un árbol deberán guardarse en un dispositivo de almacenamiento secundario, como un disco rígido, y los punteros representarán direcciones de disco, en vez de direcciones de memoria principal.

Si se utilizara un árbol binario para un conjunto de datos que tenga un millón de elementos, se necesitarían 20 accesos de búsqueda como valor promedio, lo que es una cantidad apreciable. Los árboles de búsqueda n-arios se ajustan a este problema. Si se accede a un elemento que está almacenado en disco, también se puede acceder a un grupo completo de elementos, sin que sea preciso mucho esfuerzo adicional. Esto sugiere dividir un árbol en subárboles, llamados páginas, y representarlos como unidades a las que se accede en bloque (Figura 33).

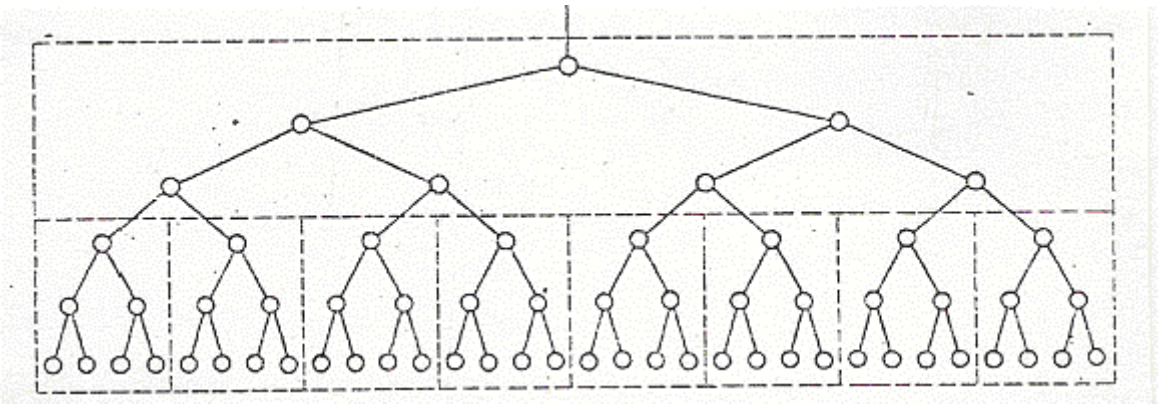


Figura 33

El ahorro en el número de accesos al disco puede ser considerable. Sin embargo, resulta imperativo que exista un método que controle el crecimiento del árbol.

El concepto de árbol B introducido por Beyer y Mc Creight, se refiere a un árbol n-ario (P,E) con punto principal t, tal que:

1. |R(t)| <= 2,
2. para todo y **Є** P, tal que y <> t, y no es maximal, |R(y)| <= n/2,
3. para todos los maximals zi, zj, | ρ (t, zi)| = | ρ (t, zj)|, es decir, todos los maximals a la misma profundidad.

Dado que el punto principal o raíz tiene al menos dos hijos, y los puntos no maximals y al menos n/2, puede demostrarse que la longitud de paso esperada para cualquier punto maximal z en un árbol B de orden n es:

$$< = \log n/ 2 ((| \max (T) | + 1) /2) ,$$

donde max(T) representa el número total de puntos maximals o claves en T. Esta relación demuestra que el tiempo de acceso vía un árbol B es una función relativamente estable de P.

Los puntos en un árbol B n-ario pueden representarse por células que contengan n campos link, separados por n-1 claves, sobre las cuales se efectuarán n comparaciones para efectuar saltos a otros puntos.

A partir de las definiciones previas, un árbol B de orden 4 con 3 niveles, tendrá todas las páginas con 2, 3 o 4 elementos, salvo la raíz que puede tener un elemento. Todos los maximals se encontrarán en el nivel 3 (Figura 34). Los elementos de las páginas ordenan sus elementos de izquierda a derecha en orden creciente. Esta organización supone una extensión natural de la de los árboles binarios de búsqueda, y determina el método a utilizar para encontrar un elemento con clave dada. Por ejemplo, considérese una página (Figura 35), Y un argumento a buscar, x. Suponiendo que se ha trasladado la página a la memoria principal, se pueden utilizar métodos de búsqueda convencionales entre las claves k1...km. Si m es suficientemente grande, se puede utilizar la búsqueda binaria; si m es bastante pequeño, una búsqueda secuencial puede ser suficiente. Si la búsqueda es infructuosa, se estará en una de las siguientes situaciones:

1. ki <= x < ki+1, para 1 <= i < m; la búsqueda continúa en la página pi .
2. km < x; la búsqueda continúa en la página pm.
3. x < k0; la búsqueda continúa en la página p0.

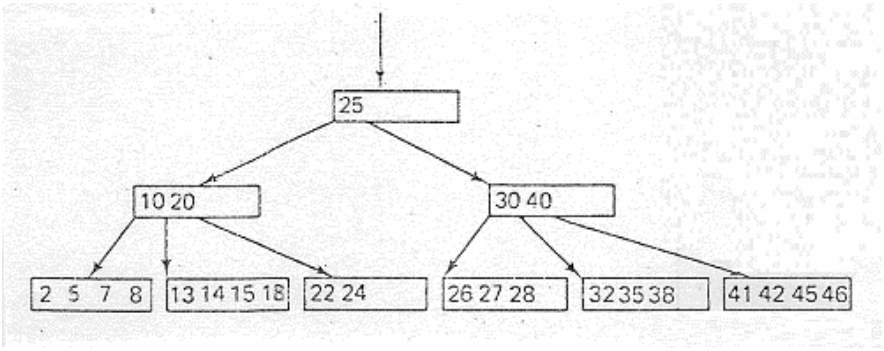


Figura 34

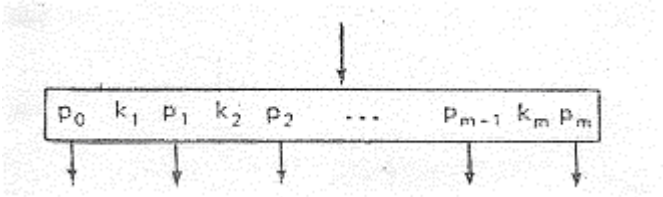


Figura 35

Si en algún caso el puntero es nil, es decir, si no hay página descendiente, entonces no hay elemento en todo el árbol que tenga clave x, y se acaba la búsqueda.

La inserción en una página de un árbol B que no esté llena, involucra a esa página únicamente, de lo contrario, hay consecuencias que afectan la estructura del árbol, y puede necesitarse la creación de nuevas páginas. En un caso extremo, el

proceso de partición puede propagarse hasta la raíz. Esta es, de hecho, la única forma en que un árbol B puede aumentar su profundidad, y a la inversa de lo imaginable, lo hace desde los maximals en dirección a la raíz. La Figura 36 muestra el resultado de construir un árbol B, insertando la siguiente secuencia de claves:

20; 40 10 30 15; 35 7 26 18 22; 5; 42 13 46 27 8 32; 38 24 45 25;

donde los punto y comas marcan los lugares en los que se ha tomado las instantáneas de creación de nuevas páginas. El planteo del algoritmo de inserción puede ser determinante en la cantidad de páginas simultáneas en la memoria principal, que en el peor de los casos será igual numéricamente a la profundidad del árbol. Una conveniencia adicional del planteo de dicho algoritmo, será que la raíz esté siempre en memoria, ya que todas las consultas al árbol arrancarán desde ella.

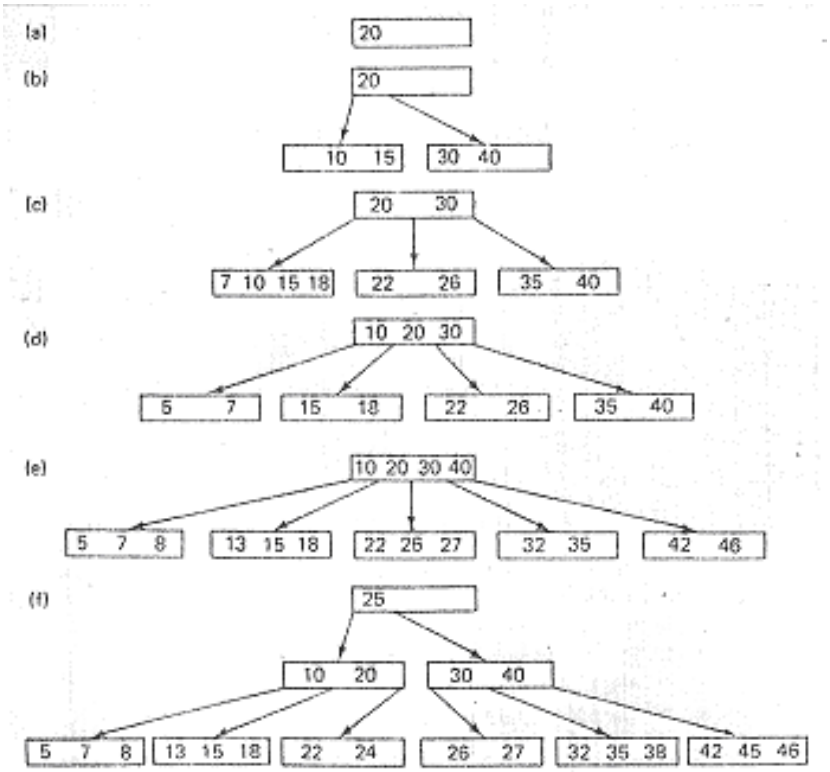


Figura 36

La eliminación de ítems en un árbol B tiene dos opciones:

1. El ítem a eliminar se encuentra en una página maximals, en cuyo caso el planteo es claro.
2. El ítem a borrar no se encuentra en una página maximals; debe sustituirse por uno de los dos ítems que le son adyacentes, que resultan estar en páginas maximals y pueden ser borrados fácilmente.

La reducción del tamaño, por la eliminación de un ítem, exige comprobar el número de ítems en la página en cuestión, a efectos de satisfacer constantemente las características del árbol B. Esto puede obligar a distribuir equitativamente los ítems de páginas adyacentes, para equilibrar el árbol. En un caso extremo, cuando esta operación no pueda realizarse, deberán unirse las dos páginas en una, y

eliminarse la restante, fenómeno que puede propagarse a la raíz. Esta es, de hecho, la única manera en que puede reducirse la altura de un árbol B.

La Figura 37 muestra el proceso gradual de reducción de tamaño del árbol B de la Figura 36, cuando se borran en secuencia las claves:

25 45 24; 38 32; 8 27 46 13 42; 5 22 18 26; 7 35 15;

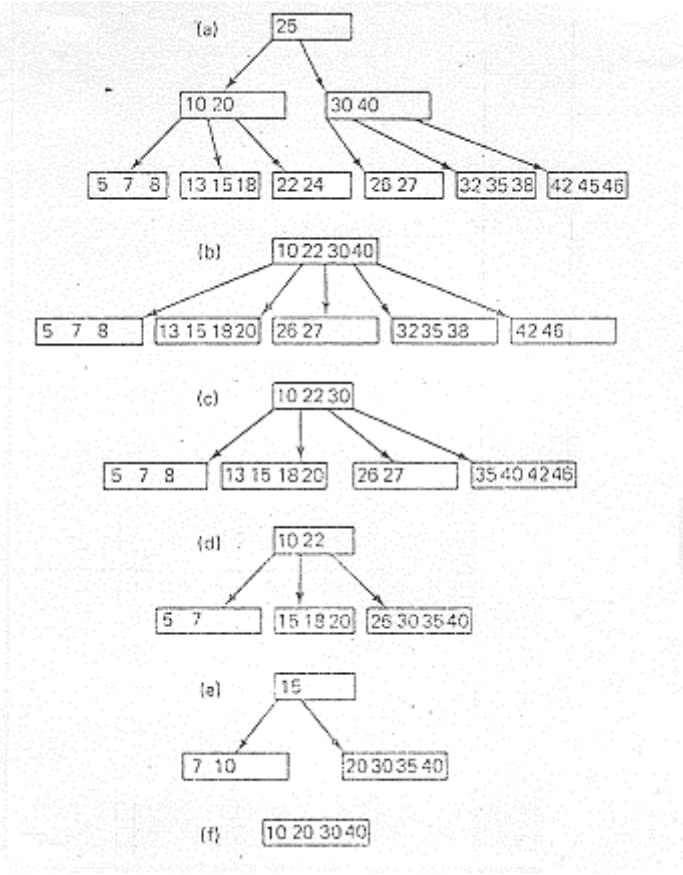


Figura 37

Como antes, los punto y comas marcan los lugares en los que se han tomado las instantáneas, que son los puntos donde se eliminan páginas.

Finalmente, puede considerarse que la organización árbol provee acceso aleatorio muy efectivo, pero leves modificaciones habilitan la posibilidad de procesos secuenciales en orden ascendentes de sus claves.

Ejercicios.

1. Dibuje el árbol binario de búsqueda desbalanceado, que se obtiene al ingresar las siguientes claves con el orden indicado:
vida, computadora, muy, pizza, algún, todos, mujer, soy, las, ver, matemáticas, una, profesor, grande, binario, de, universidad, rojo, antes, estudio, después, hombre.
Indique la longitud de paso promedio en este árbol.

2. Dibuje el árbol binario de búsqueda balanceado AVL, para la misma secuencia de claves del ejercicio anterior. Indique la longitud de paso promedio en este árbol.

3. Por el método hash de división con m=100, obtener los valores hash de los siguientes conjuntos de claves:
'Mendoza' 'Catamarca ' 'Formosa' 'Tucumán'

4. Suponiendo que se insertan las claves 1, 2, 3,...12, en un árbol B de orden 2, indique que claves originan divisiones de página, y que claves hacen que la profundidad del árbol crezca.

Listado de programas fuentes disponibles

- Evaluación de empleados de empresas.
- Evaluación de pasos de menor costo de empresa de aeronavegación.
- Conversión de expresión algebraica a lenguaje assembler.
- Clasificación por la base de numeración.
- Cálculo factorial exacto.
- Lista secuencial alumnos U.T.N..
- Clasificación por barrido simétrico.
- Clasificación por montículo (heapsort).
- Conversión de árbol n-ario a árbol binario.
- Generador de palabras cruzadas.
- Generador de árbol de búsqueda AVL.
- Generador de árbol de búsqueda óptimo.
- Hashing por hash doble.
- Hashing por linkeado.
- Generador de árbol B.

Bibliografía

- COMPUTER DATA STRUCTURES de John l. Pfaltz - 1987.
- FILE STRUCTURES de Folk-Zoellick-Riccardi - 1998
- DATA STRUCTURES de T. Standhis - 1978.
- ALGORITMOS + ESTRUCTURAS DE DATOS = PROGRAMAS de Niklaus Wirth - 1985.
- INTRODUCCION A LOS SISTEMAS DE BASE DE DATOS de C. Date - 2000
- SISTEMAS OPERATIVOS de Andrew Tanenbaum - 1988
- COMPILADORES de Alfred Aho, Ravi Sethi y Jeffrey Ullman - 1990