UNIVERSIDAD DE ORIENTE.

NUCLEÓ ANZOÁTEGUI.

ESCUELA DE INGENIERÍA Y CIENCIAS APLICADAS

DEPARTAMENTO DE COMPUTACIÓN Y SISTEMAS

MATERIA: TEORIA DE ESTRUCTURA DE DATOS

**PROFESOR:**

**MANUEL LARA**

**GRUPO:**

**LUIS CORREA C.I: 19.840.230**

**GABRIEL MOVILIO C.I:21.173.595**

**JONATHAN FERNANDEZ C.I:19.315.715**

**CARLA GOMEZ C.I:16.926.242**

**MIGUEL GALAVIS C.I:20.054.471**

**MANUEL DUN C.I:19.257.821**

**JOSE CARABALLO C.I:21.013.733**

**FRANCISCO QUIJADA C.I:19.190.821**

**Sección: 01**

TRABAJO DE ARBOLES

Barcelona, 20 Septiembre de 2012

**Índice**

|  |
| --- |
| [Introducción 1](#_Toc335941348)  [1.-Árbol: 2](#_Toc335941349)  [2.-Tipos de árboles: 4](#_Toc335941350)  [3.-Operaciones de árboles: 28](#_Toc335941351)  [4.-Algoritmos de operación árboles: 29](#_Toc335941352)  [Conclusiones 42](#_Toc335941353)  [Bibliografía 43](#_Toc335941354) |

# Introducción

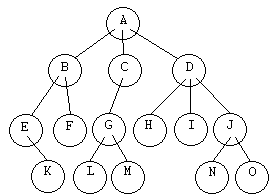
En el siguiente trabajo se describen los denominados árboles y sus aplicaciones en las ciencias de la computación.

Entre los algoritmos de operaciones para los arboles en computación, se describirán los algoritmos de Prim y el de Kruskal.

Estos son [algoritmo](http://es.wikipedia.org/wiki/Algoritmo)s pertenecientes a la [teoría de los grafos](http://es.wikipedia.org/wiki/Teor%C3%ADa_de_los_grafos), los cuales permiten encontrar un [árbol de recubrimiento mínimo](http://es.wikipedia.org/wiki/%C3%81rbol_recubridor_m%C3%ADnimo) en un [grafo](http://es.wikipedia.org/wiki/Grafo) [conexo](http://es.wikipedia.org/wiki/Teor%C3%ADa_de_los_grafos#Grafos_conexos) no dirigido y cuyas [aristas](http://es.wikipedia.org/wiki/Arista_(Teor%C3%ADa_de_grafos)) están etiquetadas.

## 1.-Árbol:

Un árbol es una estructura no lineal en la que cada nodo puede apuntar a uno o varios nodos, primero hay que explicar varios términos que se utilizaran.



* **Nodo hijo:** cualquiera de los nodos apuntados por uno de los nodos del árbol.
* **Nodo padre:** nodo que contiene un puntero a otro nodo.

Los árboles con los que trabajaremos tienen otra característica importante: cada nodo sólo puede ser apuntado por otro nodo, es decir, cada nodo sólo tendrá un padre. Esto hace que estos árboles estén fuertemente jerarquizados, y es lo que en realidad les da la apariencia de árboles.

En cuanto a la posición dentro del árbol:

* **Nodo raíz:** nodo que no tiene padre. Este es el nodo que usaremos para referirnos al árbol.
* **Nodo hoja:** nodo que no tiene hijos.
* **Nodo rama:** aunque esta definición apenas la usaremos, estos son los nodos que no pertenecen a ninguna de las dos categorías anteriores.

Se llama árbol completo al de profundidad K que tiene todos los nodos posibles hasta el penúltimo nivel (profundidad K-1), y donde los elementos del último nivel están colocados de izquierda a derecha sin dejar huecos entre ellos.

Existen otros conceptos que definen las características del árbol, en relación a su tamaño:

* **Orden**: es el número potencial de hijos que puede tener cada elemento de árbol. De este modo, diremos que un árbol en el que cada nodo puede apuntar a otros dos es de orden dos, si puede apuntar a tres será de orden tres, etc.
* **Grado**: el número de hijos que tiene el elemento con más hijos dentro del árbol. En el árbol del ejemplo, el grado es tres, ya que tanto 'A' como 'D' tienen tres hijos, y no existen elementos con más de tres hijos.
* **Nivel**: se define para cada elemento del árbol como la distancia a la raíz, medida en nodos. El nivel de la raíz es cero y el de sus hijos uno. Así sucesivamente. En el ejemplo, el nodo 'D' tiene nivel 1, el nodo 'G' tiene nivel 2, y el nodo 'N', nivel 3.
* **Altura**: la altura de un árbol se define como el nivel del nodo de mayor nivel. Como cada nodo de un árbol puede considerarse a su vez como la raíz de un árbol, también podemos hablar de altura de ramas. El árbol del ejemplo tiene altura 3, la rama 'B' tiene altura 2, la rama 'G' tiene altura 1, la 'H' cero, etc.

Hay tres formas de recorrer un árbol completo, y las tres se suelen implementar mediante recursividad. En los tres casos se sigue siempre a partir de cada nodo todas las ramas una por una.

Supongamos que tenemos un árbol de orden tres, y queremos recorrerlo por completo.

**Pre-orden**

En este tipo de recorrido, el valor del nodo se procesa antes de recorrer las ramas:

Si seguimos el árbol del ejemplo en pre-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así:

A B E K F C G L M D H I J N O

**INORDEN**

En este tipo de recorrido, el valor del nodo se procesa después de recorrer la primera rama y antes de recorrer la última. Esto tiene más sentido en el caso de árboles binarios, y también cuando existen ORDEN-1 datos, en cuyo caso procesaremos cada dato entre el recorrido de cada dos ramas (este es el caso de los árboles-b):

Si seguimos el árbol del ejemplo en in-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así:

K E B F A L G M C H D I N J O

**Post-orden**

En este tipo de recorrido, el valor del nodo se procesa después de recorrer todas las ramas:

Si seguimos el árbol del ejemplo en post-orden, y el proceso de los datos es sencillamente mostrarlos por pantalla, obtendremos algo así:

K E F B L M G C H I N O J D A

## 2.-Tipos de árboles:

* 1. [**Árboles Binarios**](http://es.wikipedia.org/wiki/%C3%81rbol_binario)**:**

Un árbol binario es un grafo conexo, a cíclico y no dirigido tal que el grado de cada vértice no es mayor a 3. De esta forma sólo existe un camino entre un par de nodos.

Un árbol binario con enraizado es como un grafo que tiene uno de sus vértices, llamado raíz, de grado no mayor a 2. Con la raíz escogida, cada vértice tendrá un único padre, y nunca más de dos hijos. Si rehusamos el requerimiento de la conectividad, permitiendo múltiples componentes conectados en el grafo, llamaremos a esta última estructura un bosque.

* + 1. [**Árbol de búsqueda binario auto-balanceado**](http://es.wikipedia.org/wiki/%C3%81rbol_de_b%C3%BAsqueda_binario_auto-balanceable)**:**

Un árbol binario de búsqueda auto-balanceado o equilibrado es un [árbol binario de búsqueda](http://es.wikipedia.org/wiki/%C3%81rbol_binario_de_b%C3%BAsqueda) que intenta mantener su altura, o el número de niveles de nodos bajo la raíz, tan pequeños como sea posible en todo momento, automáticamente. Esto es importante, ya que muchas operaciones en un árbol de búsqueda binaria tardan un tiempo proporcional a la altura del árbol, y los árboles binarios de búsqueda ordinarios pueden tomar alturas muy grandes en situaciones normales, como cuando las claves son insertadas en orden. Mantener baja la altura se consigue habitualmente realizando transformaciones en el árbol, como la [rotación de árboles](http://es.wikipedia.org/w/index.php?title=Rotaci%C3%B3n_de_%C3%A1rboles&action=edit&redlink=1), en momentos clave.

Estructuras de datos populares que implementan este tipo de árbol:

Operaciones Arboles binarios de búsqueda

-Insertar:

**-**Eliminar

* + 1. [**Árboles AVL**](http://es.wikipedia.org/wiki/%C3%81rbol_AVL)**:**

Los árboles AVL están siempre equilibrados de tal modo que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa. Gracias a esta forma de equilibrio o balanceo, la complejidad de una búsqueda en uno de estos árboles se mantiene siempre en orden de [complejidad](http://es.wikipedia.org/wiki/Complejidad_computacional) [O](http://es.wikipedia.org/wiki/Cota_superior_asint%C3%B3tica)(log n). El factor de equilibrio puede ser almacenado directamente en cada nodo o ser computado a partir de las alturas de los sub-árboles.

Para conseguir esta propiedad de equilibrio, la inserción y el borrado de los nodos se han de realizar de una forma especial. Si al realizar una operación de inserción o borrado se rompe la condición de equilibrio, hay que realizar una serie de [rotaciones de los nodos](http://es.wikipedia.org/w/index.php?title=Rotaci%C3%B3n_de_%C3%A1rbol&action=edit&redlink=1).

* + 1. [**Árboles Rojo-Negro**](http://es.wikipedia.org/wiki/%C3%81rbol_rojo-negro)**:**

Un árbol rojo-negro es un árbol binario de búsqueda en el que cada nodo almacena un bit adicional de información llamado color, el cual puede ser rojo o negro. Sobre este atributo de color se aplican restricciones que resultan en un árbol en el que ningún camino de la raíz a una hoja es más de dos veces más largo que cualquier otro, lo cual significa que el árbol es balanceado.

Cada nodo de un árbol rojo negro contiene la siguiente información: color, clave, hijo izquierdo, hijo derecho y padre.

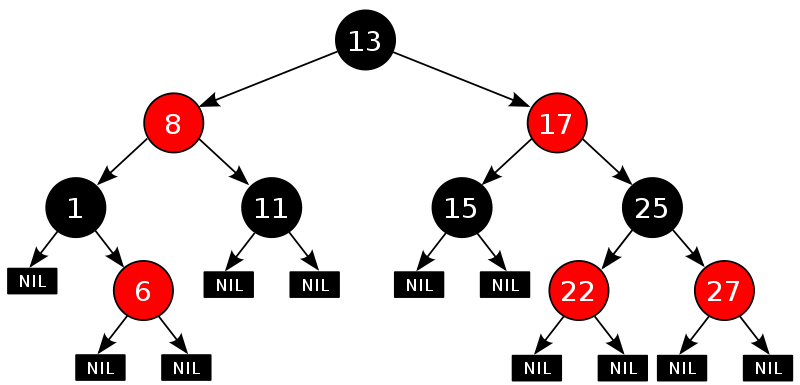
Un árbol rojo-negro es un tipo especial de árbol binario usado en informática para organizar información compuesta por datos comparables (como por ejemplo números).

En los árboles rojo-negros las hojas no son relevantes y no contienen datos. A la hora de implementarlo en un lenguaje de programación, para ahorrar memoria, un único nodo (nodo-centinela) hace de nodo hoja para todas las ramas. Así, todas las referencias de los nodos internos a las hojas van a parar al nodo centinela.

En los árboles rojo-negros, como en todos los árboles binarios de búsqueda, es posible moverse ordenadamente a través de los elementos de forma eficiente si hay forma de localizar el padre de cualquier nodo. El tiempo de desplazarse desde la raíz hasta una hoja a través de un árbol equilibrado que tiene la mínima altura posible es de O (log n).

Además de los requisitos impuestos a los árboles binarios de búsqueda convencionales, se deben satisfacer las siguientes propiedades para tener un árbol rojo-negro válido:

1. Todo nodo es rojo o negro.
2. La raíz es negra.
3. Todas las hojas son negras (las hojas son los hijos nulos).
4. Los hijos de todo nodo rojo son negros (también llamada "Propiedad del rojo").
5. Cada camino simple desde un nodo a una hoja descendiente contiene el mismo número de nodos negros, ya sea contando siempre los nodos negros nulos, o bien no contándolos nunca (el resultado es equivalente). También es llamada "Propiedad del camino", y al número de nodos negros de cada camino, que es constante para todos los caminos, se le denomina "Altura negra del árbol", y por tanto el camino no puede tener dos rojos seguidos.
6. El camino más largo desde la raíz hasta una hoja no es más largo que 2 veces el camino más corto desde la raíz del árbol a una hoja en dicho árbol. El resultado es que dicho árbol está aproximadamente equilibrado.



Representación de un árbol rojo negro y sus nodos en Java sería:

**Rotaciones:**

Las operaciones de inserción y eliminación de un árbol binario de búsqueda, si se aplican a un árbol rojo negro pueden modificar las propiedades enumeradas anteriormente. Para restaurar estas propiedades es necesario cambiar el color de algunos nodos así como también la estructura de los apuntadores.

La estructura de los apuntadores se cambia mediante rotación, la cual es una operación que preserva las propiedades de un árbol binario de búsqueda. Existen dos tipos de rotaciones: a la izquierda y a la derecha.

**Inserción:**

La inserción de un nodo en un árbol rojo-negro con n elementos puede realizarse en un tiempo O (lon n).

Nótese que una vez insertado y marcado como rojo el nodo z (justo antes de llamar a corregir Inserción) se presenta la siguiente situación:

Las propiedades 1 y 3 indudablemente se cumplen, ya que los hijos del nuevo nodo rojo insertado son NULO y por lo tanto son negros.

La propiedad 5 se cumple debido a que no se ha insertado ningún nodo negro y por lo tanto todas las rutas de la raíz a las hojas siguen teniendo el mismo número de hijos negros.

Las únicas propiedades que podrían ser violadas son la 2, que requiere que la raíz sea negra y la 4, que dice que un nodo rojo no puede tener ningún hijo rojo. Ambas violaciones se deben a que el nuevo nodo es rojo. La propiedad 2 se viola si z es la raíz y la 4 si el padre de z es rojo.

Para estudiar cómo corregir Inserción permite restaurar las propiedades de árbol rojo-negro examinaremos el código en tres partes: inicialización, terminación y mantenimiento.

***Inicialización***

Antes de la primera iteración, comenzamos con un árbol rojo-negro sin violaciones y añadimos un nodo rojo z.

Si hay una violación a la propiedad 2 (raíz negra), entonces la raíz roja tiene que ser el nodo recién añadido z, el cual sería el único nodo interno del árbol. Debido a que tanto el padre como los hijos de z son NULO, el cual es negro, no hay violación de la propiedad 4. Así, esta sería la única violación en el árbol.

Si hay una violación de la propiedad 4, entonces, considerando que los hijos del nodo z son NULO negros y que el árbol no tenía violaciones antes de la inserción de z, la violación tiene que ser porque tanto z como z.padre son rojos. Es imposible que haya alguna otra violación de las propiedades.

***Terminación***

Cuando el ciclo termina, lo hace porque z:padre es negro. Así, no hay violación de la propiedad 4 al terminar el ciclo. La única propiedad que podría fallar es la propiedad 2, la cual es restaurada en la línea 27.

***Mantenimiento***

Hay seis casos a considerar dentro del ciclo while, pero tres de ellos son simétricos; dependiendo de si z.padre es un hijo izquierdo o un hijo derecho del abuelo de z (z.padre.padre), lo cual se determina en la línea 2. Estudiaremos únicamente la primera posibilidad, correspondiente a las líneas 3-14 de corregir Inserción (nótese que el código entre las líneas 15 y 26 es simétrico).

Nótese que z.padre.padre existe, ya que la condición del ciclo es que z.padre sea rojo y por lo tanto z:padre no puede ser la raíz.

El primero de los tres casos a considerar se diferencia de los casos 2 y 3 por el color del tío de z (z.padre.padre.der). Si el tío (y) es rojo entonces se ejecuta el caso 1. De otro modo se transﬁere el control a los casos 2 y 3. En los tres casos el abuelo de z (z.padre.padre) es negro, puesto que el padre z.padre es rojo y la propiedad 4 sólo puede ser violada entre z y z.padre.

**Eliminación**

La eliminación de un nodo en un árbol rojo-negro con n elementos puede realizarse en un tiempo O (log n).

Nótese que si el nodo desaparecido es negro, pueden presentarse tres problemas:

1. Si y era la raíz y un hijo rojo de y se convierte en la nueva raíz, se viola la propiedad 2

2. Si tanto x como y:padre (que también es x:padre) eran rojos, entonces se viola la propiedad 4.

3. La remoción de y hace que cualquier ruta que previamente contenía a y ahora tenga un nodo negro menos. Por lo tanto se viola la propiedad 5. Para evitar lidiar con este problema, asumiremos que la negrura del nodo desaparecido y se le transfiere a su hijo x. El problema ahora es que x no es ni negro ni rojo (violando la propiedad 1), sino que es doble negro o rojo-negro y contribuye 2 o 1, respectivamente, al conteo de nodos negros en las rutas que lo contengan. El atributo color de x seguirá siendo ROJO (si es rojo-negro) o NEGRO (si es negro-negro). En otras palabras, el atributo negro extra de un nodo se establece por el hecho de que x apunte a él y no en el atributo color.

A continuación examinaremos cómo el procedimiento corregirEliminar restaura las propiedades de árbol rojo-negro.

* + 1. [**Árbol AA**](http://es.wikipedia.org/wiki/%C3%81rbol_AA)**:**

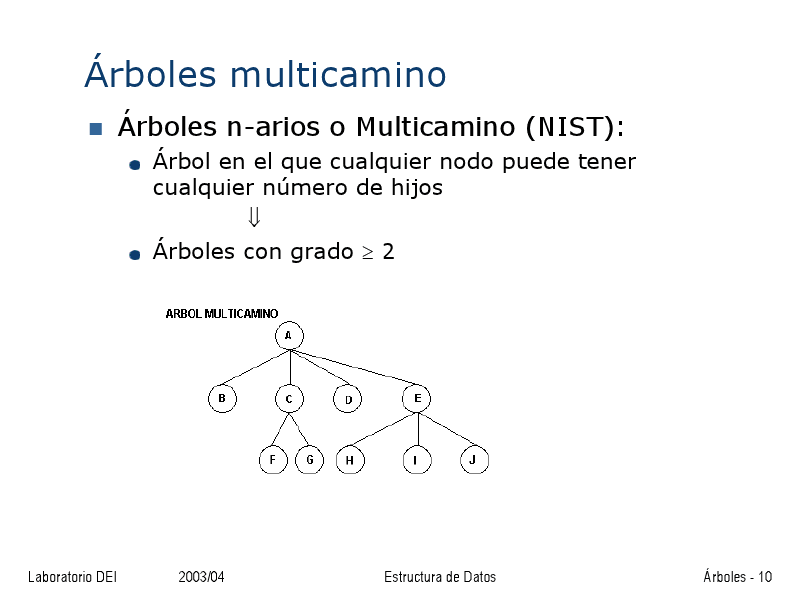
Es un tipo de [árbol binario de búsqueda auto-balanceado](http://es.wikipedia.org/wiki/%C3%81rbol_binario_de_b%C3%BAsqueda_auto-balanceable) utilizado para almacenar y recuperar información ordenada de manera eficiente.

Los árboles AA son una variación del [árbol rojo-negro](http://es.wikipedia.org/wiki/%C3%81rbol_rojo-negro), que a su vez es una mejora del [árbol binario de búsqueda](http://es.wikipedia.org/wiki/%C3%81rbol_binario_de_b%C3%BAsqueda). A diferencia de los árboles rojo-negro, los nodos rojos en un árbol AA sólo pueden añadirse como un hijo derecho. En otras palabras, ningún nodo rojo puede ser un hijo izquierdo. De esta manera se simula un [árbol 2-3](http://es.wikipedia.org/wiki/%C3%81rbol_2-3) en lugar de un [árbol 2-3-4](http://es.wikipedia.org/w/index.php?title=%C3%81rbol_2-3-4&action=edit&redlink=1), lo que simplifica las operaciones de mantenimiento.

* 1. [**Árboles Multi-camino**](http://es.wikipedia.org/wiki/%C3%81rbol_multicamino)**:**

Árbol multicamino

Hasta ahora el análisis se ha limitado a los arboles en que cada nodo tienen como máximo dos descendientes o hijos, es decir, a los arboles binarios. Esto resulta perfectamente apropiado si, por ejemplo, se quieren representar relaciones familiares en las que cada persona se encuentre relacionada con sus padres. Pero si la relación es a la inversa necesitamos una estructura que asocie a cada padre un número arbitrario de hijos. A estas estructuras se les llama arboles multicamino o n-arios. Este ultimo termino viene del ingles n-ary, donde n es el grado máximo de los nodos del árbol, por ejemplo, si n es igual a 2 al árbol se le llama binario, para n igual a 3 se le llama terciario o ternario, para n igual a 4 se le llama cuaternario, y así sucesivamente.



Definición:

Se denomina arboles multi-camino a aquellos arboles de grado mayor que dos. Es un árbol que contiene nodos con más de dos ramas.

**Ventajas e inconvenientes**

La principal ventaja de este tipo de árboles consiste en que existen más nodos en un mismo nivel que en los árboles binarios con lo que se consigue que, si el árbol es de búsqueda, los accesos a los nodos sean más rápidos.

El inconveniente más importante que tienen es la mayor ocupación de memoria, pudiendo ocurrir que en ocasiones la mayoría de los nodos no tengan descendientes o al menos no todos los que podrían tener desaprovechándose por tanto gran cantidad de memoria. Cuando esto ocurre lo más frecuente es transformar el árbol multicamino en su binario de búsqueda equivalente.

**Aplicaciones:**

* Muy utilizados en la construcción y mantenimiento de arboles de búsqueda con:
* Gran cantidad de nodos.
* Guardados en memoria secundaria,
* En los que se realizan con frecuencia inserciones y supresiones.

**Idea básica:**

* Un árbol se subdivide en sub-arboles
* Cada subárbol se representan como unidades a las que se accede simultáneamente y reciben el nombre de páginas.
* Cada acceso a página requiere un único acceso a memoria secundaria.
* Se aprovecha las características de direccionamiento mediante paginación de la memoria secundaria y se consigue un ahorro en el número de accesos a la misma.

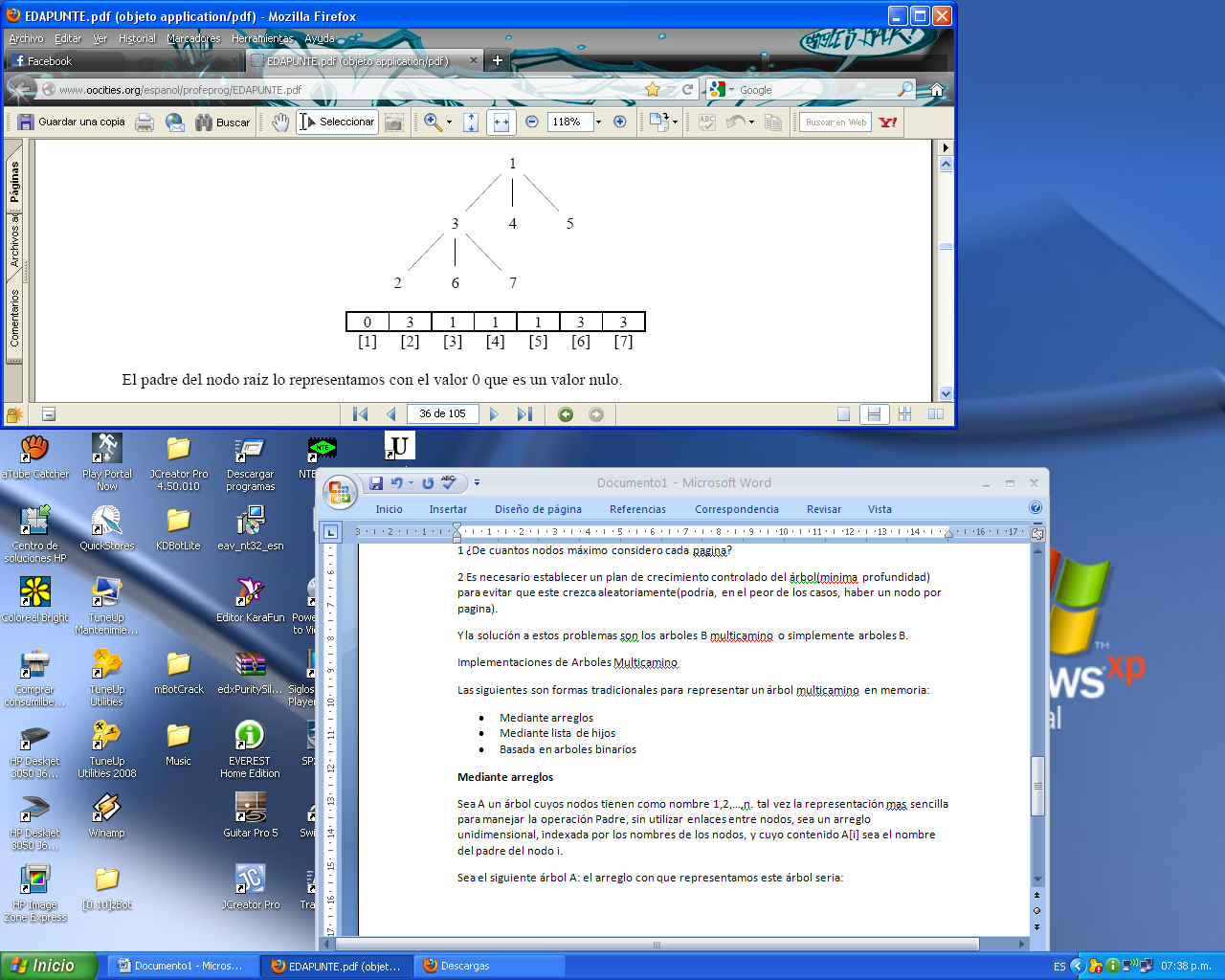
**Implementaciones de Arboles Multi-camino**

Las siguientes son formas tradicionales para representar un árbol multicamino en memoria:

* Mediante arreglos
* Mediante lista de hijos
* Basada en arboles binarios
* Mediante arreglos

Sea A un árbol cuyos nodos tienen como nombre 1,2,…, n. tal vez la representación más sencilla para manejar la operación Padre, sin utilizar enlaces entre nodos, sea un arreglo unidimensional, indexada por los nombres de los nodos, y cuyo contenido A[i] sea el nombre del padre del nodo i.

Sea el siguiente árbol A: el arreglo con que representamos este árbol seria:



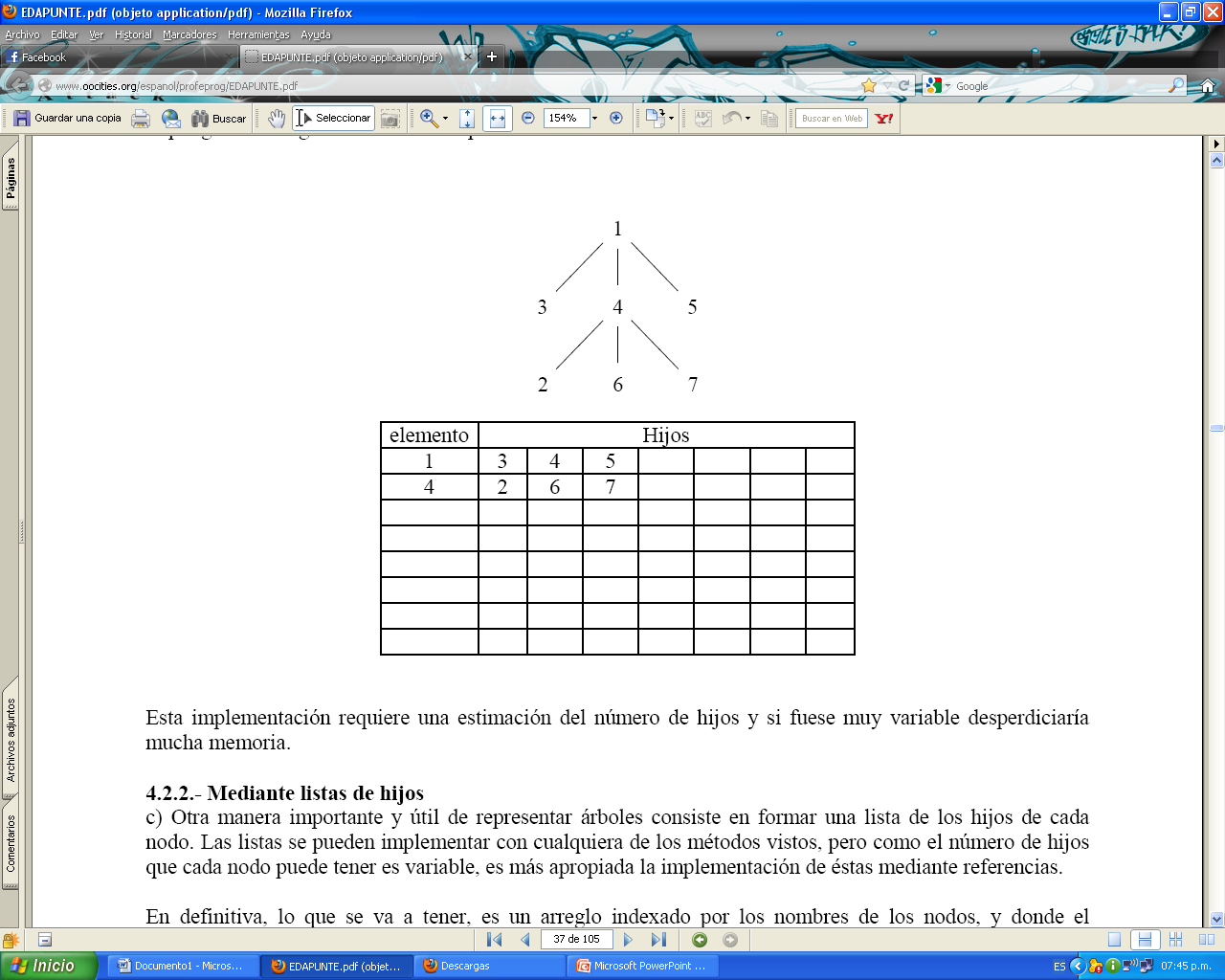
**El padre del nodo raíz lo presentamos con el valor 0 que es un valor nulo.**

Esta representación se puede realizar porque en un árbol cada nodo tiene un padre único, por lo tanto esta implementación permite hallar el padre de un nodo en un tiempo constante. Esto da lugar a que se pueda obtener un camino ascendente en el árbol, es decir de un nodo a su padre, de este a su padre y así sucesivamente, en un tiempo proporcional al número de nodos del camino.

Esta representación no facilita las operaciones que requieren información de los hijos. Dado un nodo n, resulta difícil determinar sus hijos o su altura. Además, esta representación no especifica el orden de los hijos de un nodo. Se puede imponer un orden artificial, por ejemplo, numerando los hijos de cada nodo después de numerar al padre, y numerando los hijos en orden ascendente, de izquierda a derecha.

**Otra realización** seria aquella en la que cada nodo se compone de la raíz y un arreglo de arboles con tantas componentes como el máximo de hijos de los nodos del árbol.

Supongamos el siguiente árbol: Su representación seria:



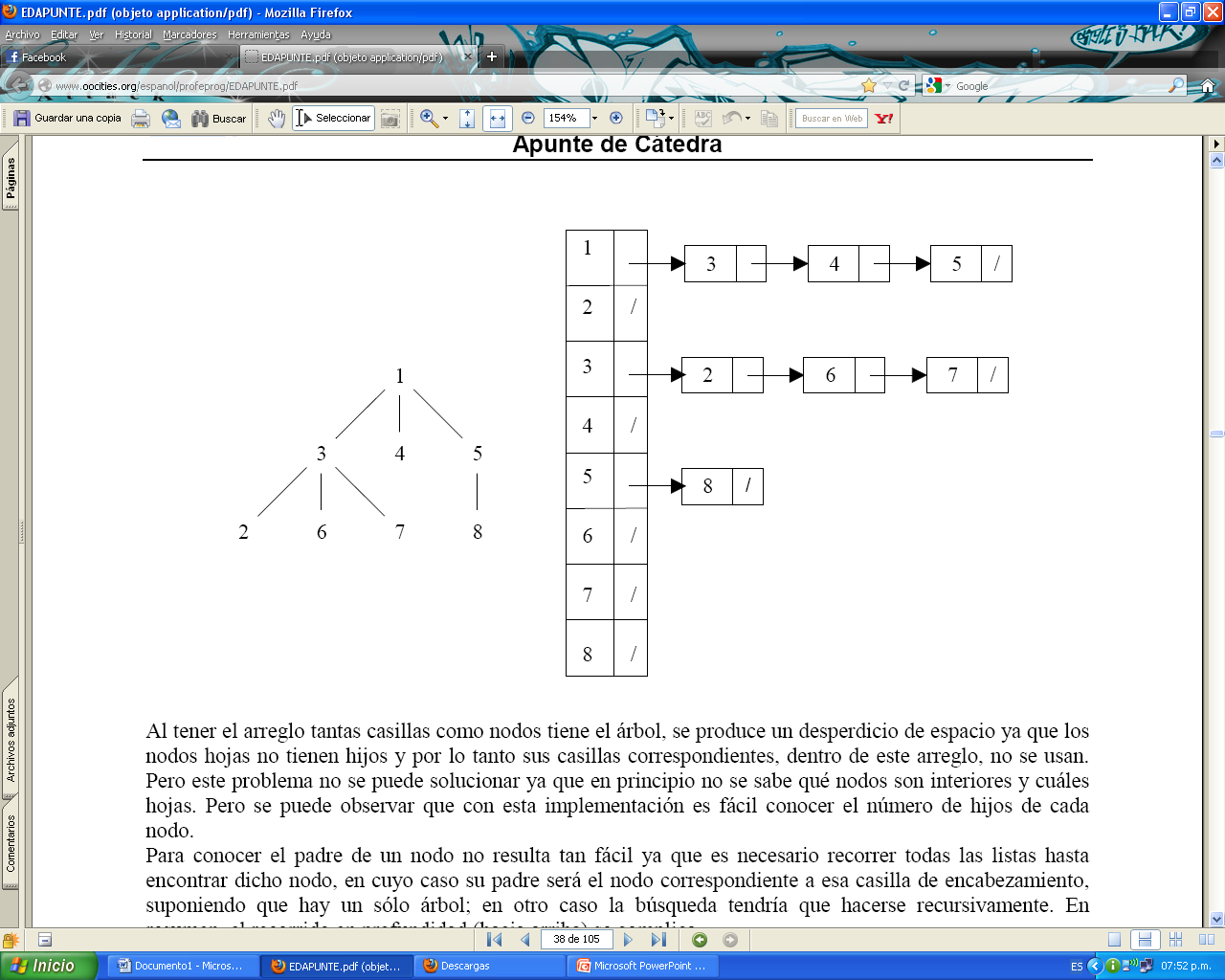
Esta implementación requiere una estimación del número de hijos y si fuese muy variable desperdiciaría mucha memoria.

* **Mediante listas de hijos**

Otra manera importante y útil de representar arboles consiste en formar una lista de los hijos de cada nodo. Las listas se pueden implementar con cualquiera de los métodos vistos, pero como el número de hijos que cada nodo puede tener es variable, es más apropiada la implementación de estas mediante referencias.

En definitiva, lo que se va a tener, es un arreglo indexado por los nombres de los nodo, y donde el contenido de cada casilla va a ser la etiqueta del nodo y una lista enlazada de arboles hijos de izquierda a derecha.

Supongamos el Árbol A: Su representación mediante listas de hijos es la siguiente:



Al tener el arreglo tantas casillas como nodos tiene el árbol, se produce un desperdicio de espacio ya que los nodos hojas no tienen hijos y por lo tanto sus casillas correspondientes, dentro de este arreglo, no se usan. Pero este problema no se puede solucionar ya que en principio no se sabe que nodos son interiores y cuales hojas. Pero se puede observar que con esta implementación es fácil conocer el número de hijos de cada nodo.

Para conocer el padre de un nodo no resulta tan fácil ya que es necesario recorrer todas las listas hasta encontrar dicho nodo, en cuyo caso su padre será el nodo correspondiente a esa casilla de encabezamiento, suponiendo que hay un solo árbol; en otro caso la búsqueda tendría que hacerse recursivamente. En resumen, el recorrido en profundidad (hacia arriba) se complica.

**Otra alternativa,** es utilizar una lista de listas. Hacerlo totalmente dinámico. En vez de representar los nodos de origen en un vector, se crea un nodo de una lista por cada uno, siempre que tenga hijos. De esta manera, no se desperdicia memoria en aquellos casos donde un nodo no posee hijos.

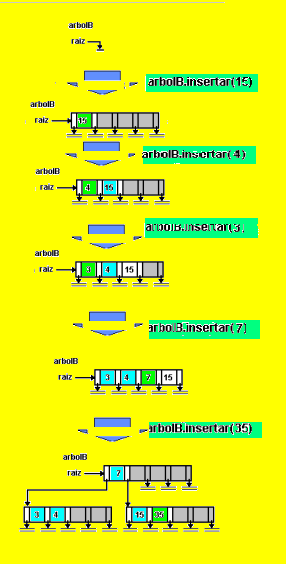
Un tipo especial de árboles multi-camino utilizado para solucionar el problema de la ocupación de memoria son los [árboles B](http://es.wikipedia.org/wiki/B-%C3%81rbol):

* + 1. [**Árboles B**](http://es.wikipedia.org/wiki/%C3%81rbol-B)**(Arboles de búsqueda multi-camino auto-balanceados):**

La idea es que los nodos internos deben tener un número variable de nodos hijo dentro de un rango predefinido. Cuando se inserta o se elimina un dato de la estructura, la cantidad de nodos hijo varía dentro de un nodo. Para que siga manteniéndose el número de nodos dentro del rango predefinido, los nodos internos se juntan o se parten.

**Algoritmos**

**Búsqueda**

La búsqueda es similar a la de los árboles binarios. Se empieza en la raíz, y se recorre el árbol hacia abajo, escogiendo el sub-nodo de acuerdo a la posición relativa del valor buscado respecto a los valores de cada nodo. Típicamente se utiliza la búsqueda binaria para determinar esta posición relativa.

Procedimiento

ejemplo2 inserción en árbol B.

1. Situarse en el nodo raíz.
2. (\*) Comprobar si contiene la clave a buscar.
   1. Encontrada fin de procedimiento.
   2. No encontrada:
      1. Si es hoja no existe la clave.
      2. En otro caso el nodo actual es el hijo que corresponde:
         1. La clave a buscar k < k1: hijo izquierdo.
         2. La clave a buscar k > ki y k < ki+1 hijo iésimo.
         3. Volver a paso 2(\*).

**Inserción**

Todas las inserciones se hacen en los nodos hoja.

1. Realizando una búsqueda en el árbol, se halla el nodo hoja en el cual debería ubicarse el nuevo elemento.
2. Si el nodo hoja tiene menos elementos que el máximo número de elementos legales, entonces hay lugar para uno más. Inserte el nuevo elemento en el nodo, respetando el orden de los elementos.
3. De otra forma, el nodo debe ser dividido en dos nodos. La división se realiza de la siguiente manera:
   1. Se escoge el valor medio entre los elementos del nodo y el nuevo elemento.
   2. Los valores menores que el valor medio se colocan en el nuevo nodo izquierdo, y los valores mayores que el valor medio se colocan en el nuevo nodo derecho; el valor medio actúa como valor separador.
   3. El valor separador se debe colocar en el nodo padre, lo que puede provocar que el padre sea dividido en dos, y así sucesivamente.

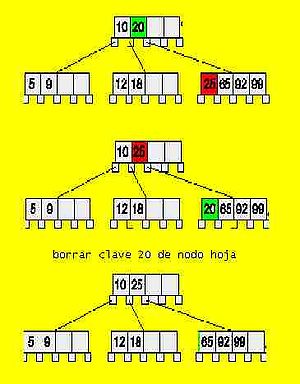
Si las divisiones de nodos suben hasta la raíz, se crea una nueva raíz con un único elemento como valor separador, y dos hijos. Es por esto por lo que la cota inferior del tamaño de los nodos no se aplica a la raíz. El máximo número de elementos por nodo es *U*-1. Así que debe ser posible dividir el número máximo de elementos *U*-1 en dos nodos legales. Si este número fuera impar, entonces *U*=2*L*, y cada uno de los nuevos nodos tendrían (*U*-2)/2 = *L*-1 elementos, y por lo tanto serían nodos legales. Si *U*-1 fuera par, *U*=2*L*-1, así que habría 2*L*-2 elementos en el nodo. La mitad de este número es *L*-1, que es el número mínimo de elementos permitidos por nodo.

Un algoritmo mejorado admite una sola pasada por el árbol desde la raíz, hasta el nodo donde la inserción tenga lugar, dividiendo todos los nodos que estén llenos encontrados a su paso. Esto evita el costo de volver a cargar en memoria los nodos padres (lo que puede llegar a ser caro si los nodos se encuentran en una memoria secundaria). Sin embargo, para usar este algoritmo mejorado, debemos ser capaces de enviar un elemento al nodo padre y dividir el resto *U*-2 elementos en 2 nodos legales, sin añadir un nuevo elemento. Esto requiere que *U*=2*L* en lugar de *U*=*L*-1, lo que explica por qué algunos libros de texto imponen este requisito en la definición de árboles-B.

**Eliminación**

La eliminación de un elemento es directa si no se requiere corrección para garantizar sus propiedades. Hay dos estrategias populares para eliminar un nodo de un árbol B.

* localizar y eliminar el elemento, y luego corregir, o
* hacer una única pasada de arriba a abajo por el árbol, pero cada vez que se visita un nodo, reestructurar el árbol para que cuando se encuentre el elemento a ser borrado, pueda eliminarse sin necesidad de continuar reestructurando

Se pueden dar dos problemas al eliminar elementos: primero, el elemento puede ser un separador de un nodo interno. Segundo, puede suceder que al borrar el elemento, el número de elementos del nodo quede debajo de la cota mínima. Estos problemas se tratan a continuación en orden.

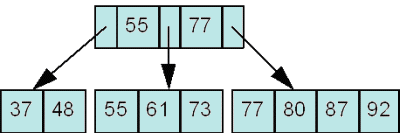
#### Eliminación en un nodo hoja

* Busque el valor a eliminar.
* Si el valor se encuentra en un nodo hoja, se elimina directamente la clave, posiblemente dejándolo con muy pocos elementos; por lo que se requerirán cambios adicionales en el árbol.
  + 1. [**Árbol-B+**](http://es.wikipedia.org/wiki/%C3%81rbol-B%2B)**:**

Los árboles –B+ se han convertido en la técnica más utilizada para la organización de archivos indizados. La principal característica de estos árboles es que toda la información se encuentra en las hojas, mientras que los nodos raíz e interiores almacenan claves que se utilizan como índices.

Es de notar que los arboles-B+ ocupan un poco mas de espacio que los arboles-B, y esto ocurre al existir duplicidad en algunas claves. Sin embargo, esto es aceptable si el archivo se modifica frecuentemente, puesto que se evita la operación de reorganización del árbol que es tan costosa en los arboles-B.

Formalmente se define un árbol-B+ de la siguiente manera:

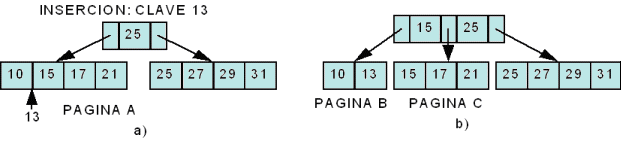
1. Cada página, excepto la raíz, contiene entre d y 2d elementos.
2. Cada página, excepto la raíz, tiene entre d + 1 y 2d + 1 descendientes. Se utiliza m para expresar el número de elementos por página.
3. La página raíz tiene al menos dos descendientes.
4. Las paginas hojas están todas al mismo nivel.
5. Todas las claves se encuentran en las páginas hojas.
6. Las claves de las páginas raíz e interiores se utilizan como índices.

**Búsqueda De Arboles-B+**

La operación de búsqueda en árboles-B+ es similar a la operación de búsqueda en árboles-B. El proceso es simple, sin embargo puede suceder que al buscar una determinada clave la misma se encuentra en una página raíz o interior, en dicho caso no debe detenerse el proceso, sino que debe continuarse la búsqueda con la página apuntada por la rama derecha de dicha clave.

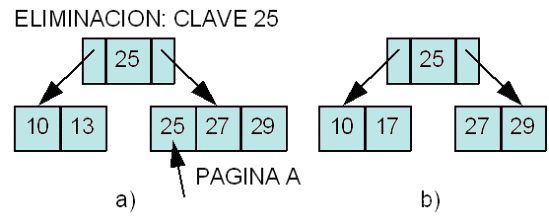
Por ejemplo, al buscar la clave 55 en el árbol –b+ de la figura anterior se advierte que esta se encuentra en la página raíz. En este caso, se debe continuar el proceso de búsqueda en la página apuntada por la rama derecha de dicha clave.

**Inserción en árboles B+**

El proceso de inserción en árboles-B+ es relativamente simple, similar al proceso de inserción en árboles-B. La dificultad se presenta cuando desea insertarse una clave en una página que se encuentra llena (m = 2d). En este caso, la página afectada se divide en 2, distribuyéndose las m + 1 claves de la siguiente forma: las d primeras claves en la página de la izquierda y las d + 1 restantes claves en la página derecha. Una copia de la clave del medio sube a la página antecesora.

**Eliminación En Arboles-B+**

La operación de Eliminación en árboles-B+ es más simple que la operación de borrado en árboles-B. Esto ocurre porque las claves a eliminar siempre se encuentran en las páginas hojas. En general deben distinguirse los siguientes casos:

1. Si al eliminar una clave, m queda mayor o igual a d entonces termina la operación de borrado. Las claves de las paginas raíz o internas no se modifican por más que sean una copia de la clave eliminada en las hojas
2. Si al eliminar una clave, m queda menor a d entonces debe realizarse una redistribución de claves, tanto en el índice como en las páginas hojas.
   * 1. [**Árbol-B\***](http://es.wikipedia.org/wiki/%C3%81rbol-B*)**:**

Utilizado en los sistemas de ficheros [HFS](http://es.wikipedia.org/wiki/HFS) y [Reiser4](http://es.wikipedia.org/wiki/Reiser4), que requiere que los nodos no raíz estén por lo menos a 2/3 de ocupación en lugar de 1/2. Para mantener esto los nodos, en lugar de generar inmediatamente un nodo cuando se llenan, comparten sus claves con el nodo adyacente. Cuando ambos están llenos, entonces los dos nodos se transforman en tres. También requiere que la clave más a la izquierda no sea usada nunca.

## 3.-Operaciones de árboles:

Las operaciones básicas en árboles son:

* Insertar elementos.
* Buscar un elemento.
* Moverse a través del árbol.
* Borrar un elemento.
* Recorrer el árbol completo.

## 4.-Algoritmos de operación árboles:

* **ALGORITMO DE PRIM:**

Fue diseñado en 1930 por el matemático [Vojtech Jarnik](http://es.wikipedia.org/wiki/Vojtech_Jarnik) y luego de manera independiente por el científico computacional [Robert C. Prim](http://es.wikipedia.org/wiki/Robert_C._Prim) en 1957 y redescubierto por [Dijkstra](http://es.wikipedia.org/wiki/Dijkstra) en 1959.

Es un algoritmo de la teoría de los grafos para encontrar un árbol de expansión mínimo en un grafo conexo no dirigido en el cual las aristas están etiquetadas.

El algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible.

Si el grafo no es conexo, entonces el algoritmo encontrará el árbol de expansión mínimo para uno de los componentes conexos que forman dicho grafo no conexo.

Se considera el coste del árbol la suma de los pesos de las aristas que lo componen. La construcción del árbol sigue los siguientes pasos.

Algoritmo:

1. Empezar en un vértice arbitrario v. El árbol consta inicialmente sólo del nodo v.

2. Del resto de vértices, buscar el que esté más próximo a v, es decir, con la arista (v, w) o (w, v) de coste mínimo. Añadir w y la arista al árbol.

3. Buscar el vértice más próximo a cualquiera de estos dos. Añadir ese vértice y la arista al árbol de expansión.

4. Repetir sucesivamente hasta añadir todos los vértices.

SEUDO CÓDIGO

Código en C++:

* **ALGORITMO DE KRUSKAL:**

El algoritmo de Kruskal fue propuesto por Joseph Kruskal en 1956 que, al igual que el algoritmo de Prim, tiene como objetivo encontrar el árbol mínimo es sus aristas partiendo de un grafo conexo y ponderado.

Este algoritmo consiste en: primero crear un árbol por cada nodo del grafo y luego se busca cuál de las aristas tiene menor ponderación, para que los arboles involucrados se conecten y formen un árbol, luego se vuelve a buscar la arista de ponderación de menor ponderación sin contar los ya elegidos y cuidando que no se formen circuitos en los árboles, es decir que se conecte un árbol con sí mismo. Esto se repite hasta que el árbol conecte todos los nodos.

Se sabe que el grafo a optimizar se puede implementar con una matriz bidimensional de adyacencias, cuidando por supuesto que sea conexo y que la adyacencia sea simétrica (es decir que el peso de nodo A a nodo B sea el mismo que de nodo B a nodo A). Una vez que se tiene la matriz de adyacencia de forma correcta se procede a aplicar el algoritmo; el árbol resultante será también una matriz de adyacencia.

**Pseudocódigo:**

Código en C++

# Conclusiones

Se ha definido en el presente trabajo el diseño de redes:

* Físicas.
* Telefónicas.
* Eléctricas.
* Hidráulicas.
* Televisión por cable, computadores.
* Carreteras.
* Como la solución de problemas NP (es el conjunto de los problemas de decisión que pueden ser resueltos por una máquina no determinista en tiempo polinómico).
* Aplicaciones indirectas: plegamiento de proteínas, reconocimiento de células cancerosas, etc.

# Bibliografía

* Wikipedia [Arbol\_(informatica)](http://es.wikipedia.org/wiki/%C3%81rbol_(programaci%C3%B3n)).
* Wikipedia Estructura de datos.
* Libro matemáticas discretas para la ciencia de la computación Hugo David Calderón Vilca.
* [www.lawebdelprogramador.com](http://www.lawebdelprogramador.com)
* [c.conclase.net](chrome://newtabhttp/c.conclase.net/edd/index.php?cap=000#inicio)