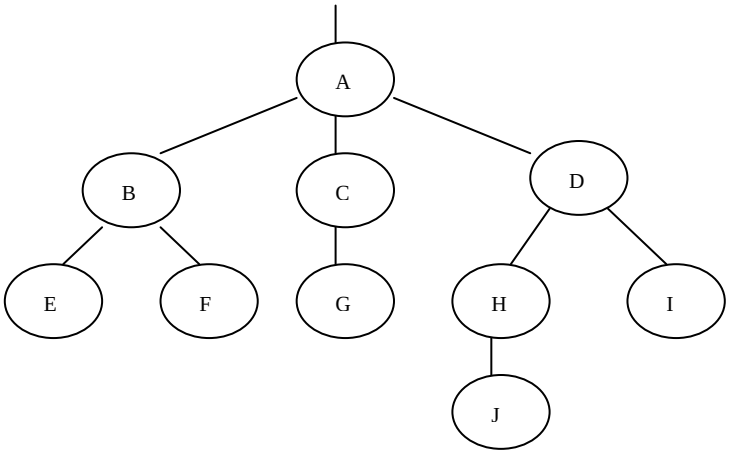
TIPO ABSTRATO DE DATOS ***ÁRBOL***

Un árbol impone una estructura jerárquica sobre un conjunto de elementos, llamados nodos. Organigramas y árboles genealógicos son ejemplos comunes de árboles. En computación se usan árboles para organizar información en sistemas de Bases de Datos, para representar fórmulas matemáticas y estructuras sintácticas de programas fuente en los compiladores, por citar algunos ejemplos.

En nuestra área de conocimiento, los árboles suelen graficarse invertidos, esto es la copa del árbol hacia abajo, en comparación a su disposición en la naturaleza.



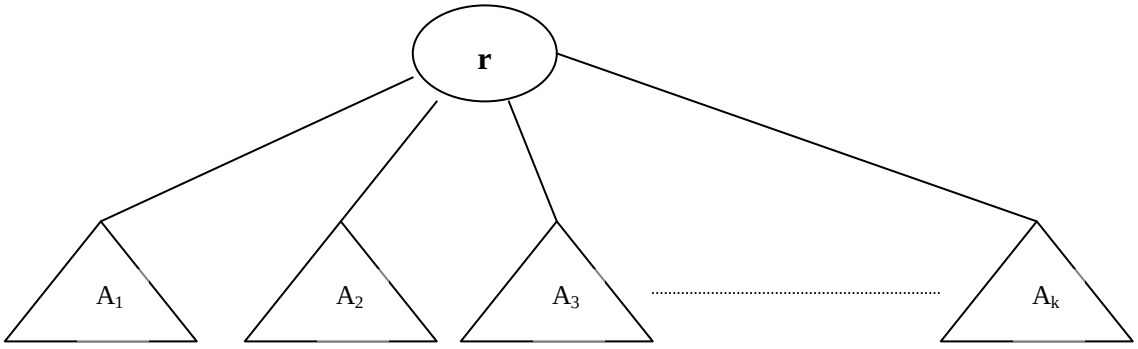
Al nodo superior – el nodo A en el ejemplo- se lo llama raíz del árbol. Weiss hace explícita esta distinción, como lo expresa la siguiente definición[1]:

Un ***Árbol*** es una colección o conjunto de nodos. La colección puede:

1 - Estar vacía

2 –Si no está vacía, consiste de un nodo distinguido **r**, conocido como *raíz*, y cero o más (sub)árboles A1, A2,.....,Ak, cada uno de los cuales tiene su raíz conectada a r por medio de una *arista* (o *rama*) dirigida.

Por esta definición encontramos que un árbol es un conjunto de n nodos y n-1 aristas, que gráficamente representamos:



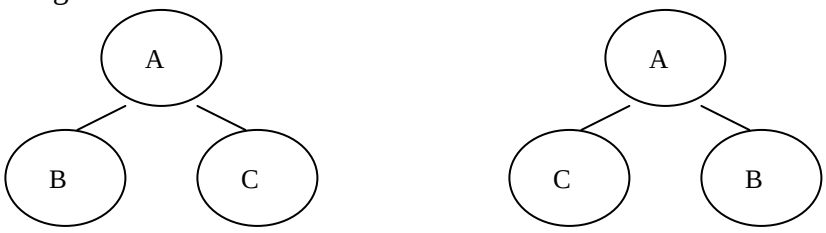
[1] Weiss, Mark Allen, **Estructuras de Datos y Algoritmos,** Addison-Wesley Iberoamericana, pág.93

Es importante resaltar, que estamos ante una estructura de datos que es por naturaleza *recursiva*, ya que se define en término de sí misma.

- La raíz de cada subárbol es un ***hijo*** o ***descendiente directo*** de r, y r es el ***padre*** o ***antecesor directo*** de cada raíz de los subárboles.
  Retomando el gráfico de la página 1, podemos considerar que el nodo C tiene como hijo al nodo G y como padre al nodo A.
- Los nodos sin hijos se llaman ***hojas***. Hojas en el ejemplo son los nodos E, F, G, J, I.
- ***Camino de un nodo n_1 a otro n_k***: secuencia de nodos n_1, n_2, ..., n_k , tal que n_i es el padre de n_{i+1} para $1<=i<k$. En un árbol existe solamente un camino desde la raíz a cada nodo. Por ejemplo el camino entre los nodos A y J es la secuencia: A,D,H,J.
- Si hay un camino entre n_1 y n_2, entonces n_1 es ***antecesor*** de n_2 y n_2 es ***descendiente*** de n_1.
- ***Longitud de camino de un nodo n_i a otro n_k***: Número de aristas que forman el camino, o número de nodos menos 1 que forman la secuencia. Existe un camino de longitud cero desde cada nodo a sí mismo.

Asimismo, según Wirth[2] :

- ***Nivel de un nodo***: si el nodo x está en el nivel i, entonces sus descendientes directos están en el nivel i+1. La raíz de un árbol, se define como localizada en el nivel 1. En el ejemplo, el nivel del nodo C es 2; el nivel del nodo J es 4.
- ***Profundidad o Altura del árbol***: máximo de los niveles de todos los nodos del árbol. La altura del árbol ejemplo es 4.
- ***Nodo Interior***: Nodo no hoja. Son interiores los nodos A, B, C, D, H.
- ***Grado de un nodo***: Número de descendientes directos de un nodo. El grado del nodo A es 3, del nodo D es 2 y del nodo I es 0. En particular las hojas tienen grado 0.
- ***Grado del árbol***: Grado máximo en todos los nodos. El grado del árbol es 3.
- ***Árbol Ordenado***: Árbol en el que las ramas de cada nodo están ordenadas. En la siguiente figura se muestran dos árboles ordenados diferentes.



Si deseamos ignorar explícitamente el orden de los hijos, hablamos entonces de un árbol no ordenado.

Árbol Binario

Un árbol ordenado de grado 2 se denomina ***Árbol Binario***. Este árbol se define como

[2] Wirth, Niklaus, **Algoritmos y Estructura de Datos**, México, Prentice Hall, 1998, 306 páginas . pág 210-211.

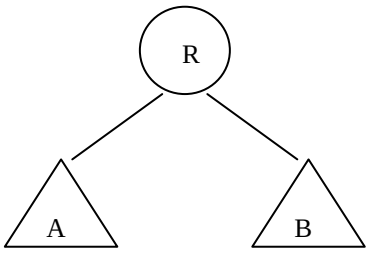
un conjunto finito de elementos (nodos) que es vacío o bien consta de una raíz (nodo) con dos árboles binarios disjuntos llamados *subárbol izquierdo* y *derecho* de la raíz[3].

Del abanico de situaciones en las que se usan árboles binarios, citamos:

- ✓ Árbol genealógico en el que para cada nodo, que representa a una persona, sus descendientes directos representan a su padre y a su madre.
- ✓ Historia de un torneo de tenis, cada nodo caracteriza al jugador ganador de un juego de competidores representados por sus descendientes.
- ✓ Mención especial merece la aplicación de árboles binarios en la construcción de "Códigos de Huffman", usados en *compresión de datos*. La compresión de datos consiste en la transformación de una cadena de caracteres de cierto alfabeto en una cadena que contiene la misma información, pero cuya longitud es mas pequeña que la de la cadena original. Esta tarea requiere el diseño de un *código* que pueda ser utilizado para representar de manera única todo carácter de la cadena de entrada. Es para la construcción de este código, que en este caso es de longitud variable pues no todas las palabras código tienen la misma longitud, que se utiliza de manera eficiente el árbol binario[4].
- ✓ Árboles de expresiones aritméticas.

Recorridos en un árbol binario

Una de las tareas que comúnmente debemos realizar sobre las estructuras de datos, y que denominamos *recorrido*, consiste en procesar todos sus elementos. En el caso particular del árbol, los nodos individuales son visitados siguiendo un orden específico, por lo que pueden considerarse como colocados en una disposición lineal. Existen tres ordenamientos principales que surgen en forma natural de las estructuras de árboles binarios. Estos ordenamientos, al igual que la definición de la estructura de árbol, se expresan adecuadamente en términos recursivos. A partir de un árbol binario, como el de la siguiente figura:



en el que R representa a la raíz y A y B simbolizan a los (sub)árboles izquierdos y derecho de R, podemos distinguir los tres ordenamientos siguientes:

1 – Preorden : Procesar nodo R
Procesar A en Preorden.
Procesar B en Preorden.

[3] Wirth, Niklaus, op.cit. pág 213.
[4] Aho, Alfred,Hopcroft,John y Ullman, Jeffrey. **Estructuras de Datos y Algoritmos.** México.Adisson Wesley Longman de México. 1998. 438 páginas. Pág.95.

2 – En orden o Inorden : Procesar A en Inorden.
Procesar nodo R.
Procesar B en Inorden.

3 – Postorden: Procesar A en Postorden.
Procesar B en Postorden.
Procesar nodo R.

Aplicación de árboles binarios:

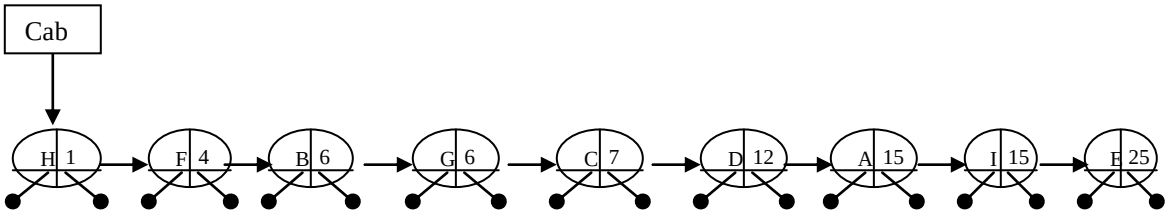
a) Códigos de Huffman

Vamos a generar códigos de longitud variable, que representen de manera única cada carácter. Para poder alcanzar el objetivo tras el cual estos códigos son generados, que es el de comprimir los datos que codifican, las longitudes de los códigos tienen relación con la cantidad de veces que los caracteres ocurren, esto es, a los caracteres que mas ocurren les corresponderán los códigos de menor longitud.
El proceso se realiza a partir de un conjunto de caracteres que pertenecen al alfabeto sobre el que se construye la entrada, y de los que se conoce su frecuencia de ocurrencia :

Caracter	A	B	C	D	E	F	G	H	I
Frecuencia	15	6	7	12	25	4	6	1	15

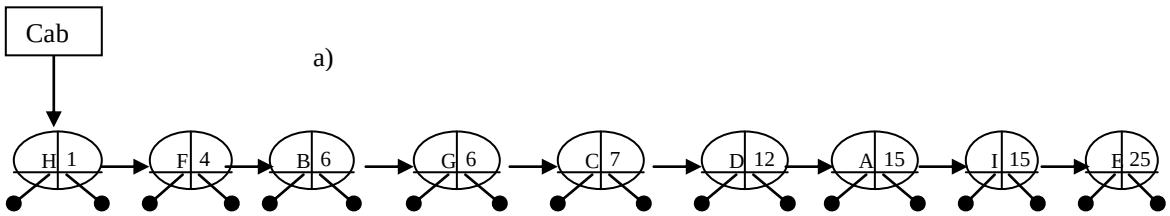
A partir de estos datos se realizan una serie de pasos en los que se trabaja con listas ordenadas por contenido y árboles binarios.

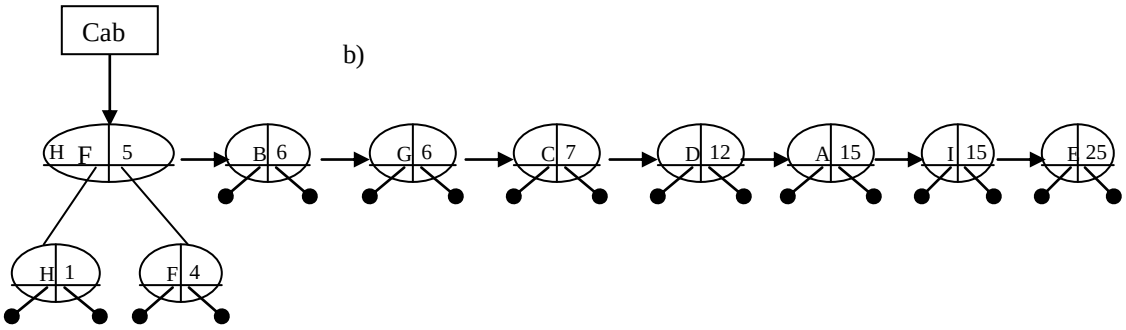
- Paso 1: Generar una lista de caracteres, ordenada en forma creciente por sus frecuencias de ocurrencia.



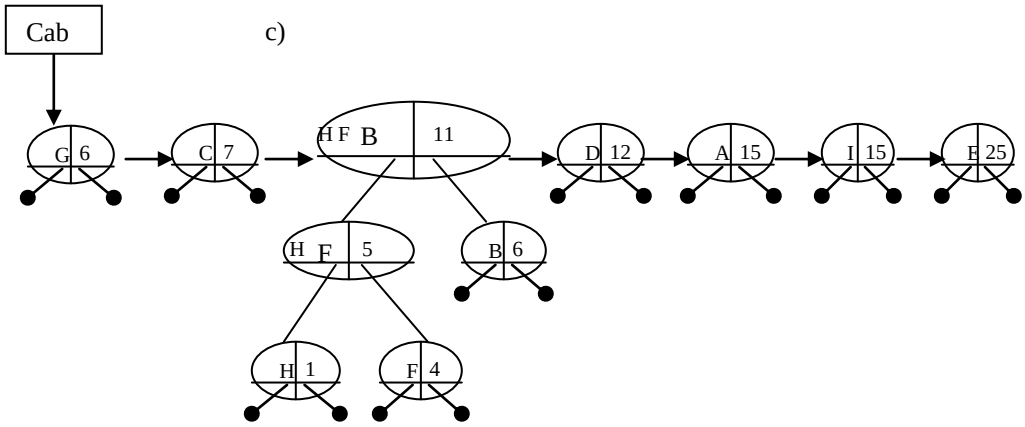
En esta lista, cada celda es un árbol binario, formado por un solo nodo, raíz del árbol.

- Paso 2: Generar árboles binarios a partir de aquellos con frecuencias menores, e insertarlos nuevamente en la lista ordenada. La frecuencia asociada a la raíz de cada nuevo árbol es la *suma* de las frecuencias de sus subárboles.

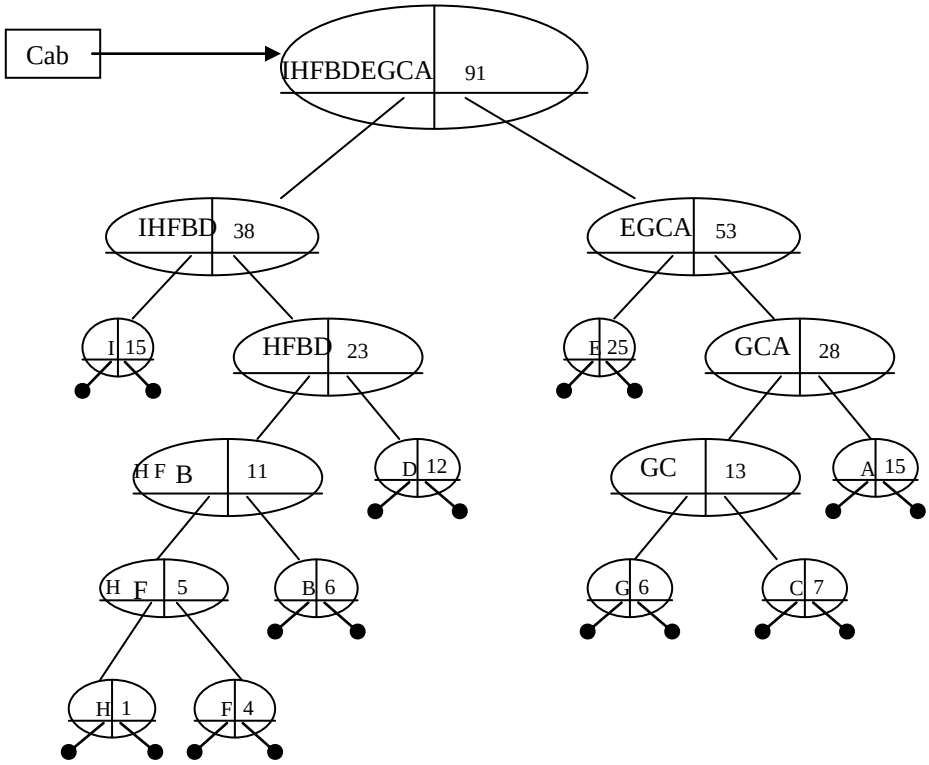




A partir de a), se produce la supresión de las celdas correspondientes a los árboles con menor frecuencia - H y F con frecuencias 1 y 4 respectivamente. Con ellos se genera un nuevo árbol binario, con H como raíz del subárbol izquierdo, y F, raíz del subárbol derecho, que es insertado en la lista. La frecuencia de la raíz de este nuevo árbol, 5, es la suma de las frecuencias de las raíces de sus subárboles.
El siguiente gráfico resulta de suprimir de la lista los árboles con valores HF y B en sus raíces, y frecuencias 5 y 6 respectivamente. El nuevo árbol, con frecuencia 11, es insertado en la lista.

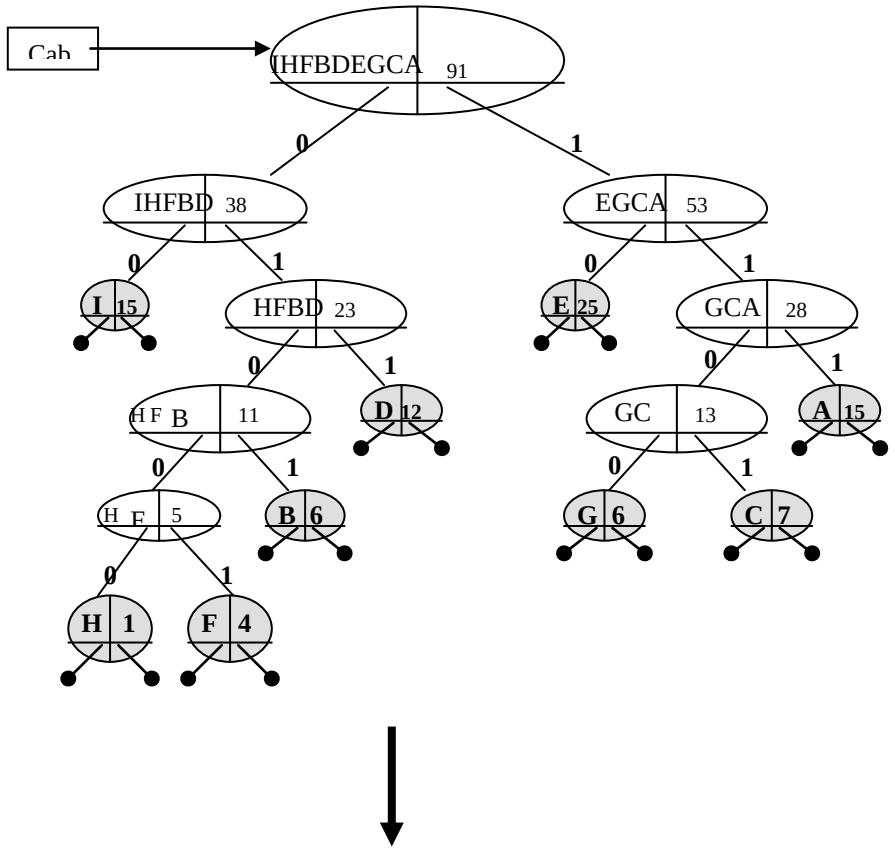


Esta secuencia de supresiones, síntesis e inserciones se realiza hasta el momento en que en la lista queda una sola celda, que corresponde al árbol binario con todos los caracteres del diccionario de entrada en sus hojas.
El árbol resultante en nuestro ejemplo, es el siguiente:



Si bien en cada nodo *no* terminal se muestra una cadena de caracteres que corresponde a todos los caracteres contenidos en sus nodos hojas descendientes, esto es al solo efecto de brindar un apoyo visual al momento del análisis de esta estructura.

- Paso 3: Generar el código de longitud variable correspondiente a cada carácter, hoja del árbol, concatenando ceros y unos, según se atraviese en el árbol una rama izquierda o una rama derecha.

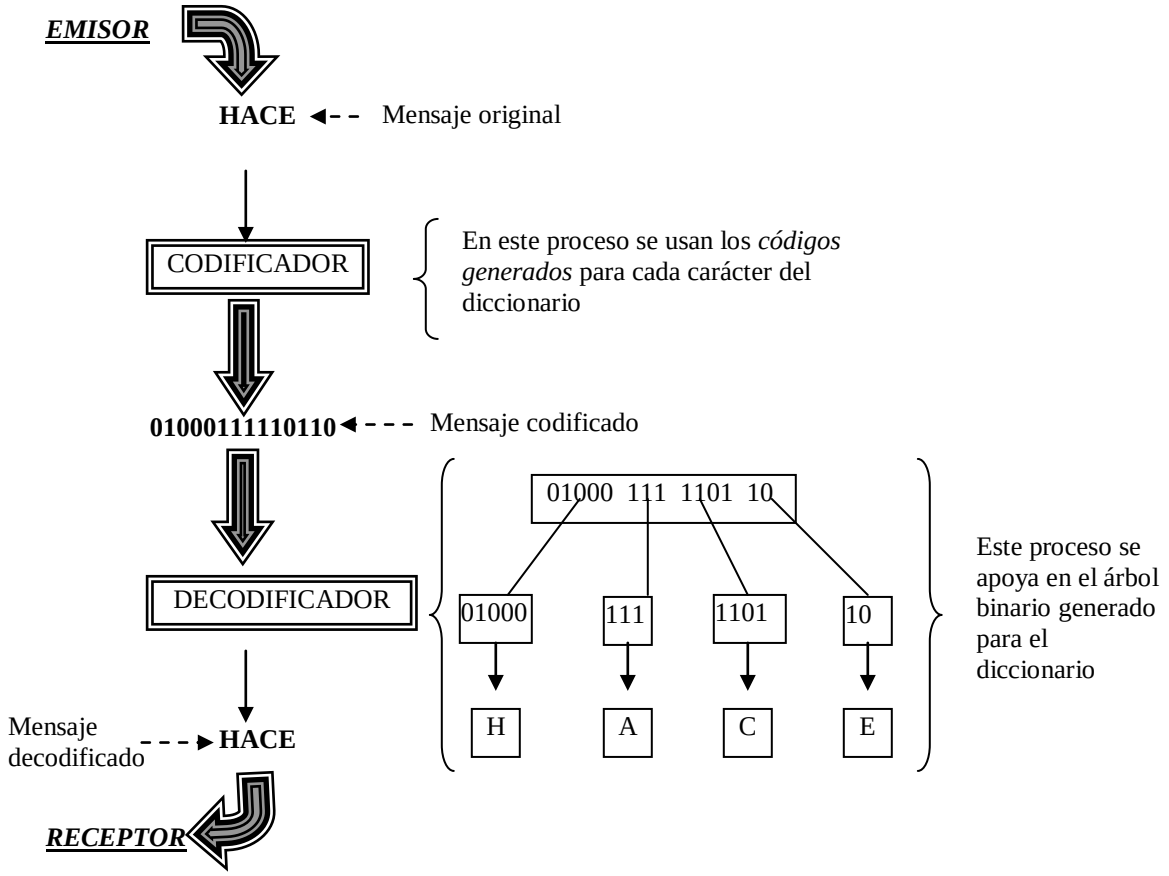


CARACTER	FRECUENCIA	CÓDIGO
H	1	**01000**
F	4	**01001**
B	6	**0101**
G	6	**1100**
C	7	**1101**
D	12	**011**
A	15	**111**
I	15	**00**
E	25	**10**

Hasta el momento hemos usado un árbol binario para generar el código, único, asociado a cada carácter. Esta unicidad códigos-caracteres facilita el proceso de decodificación. Para este proceso se hace uso del árbol binario, recorriéndolo desde la raíz, luego hacia izquierda o hacia derecha según la entrada sea cero o uno, hasta alcanzar una hoja y así descubrir el caracter.

Para interpretar mejor lo expresado, pensemos en un contexto de transferencia de datos, en el que existe un Emisor que genera un mensaje, un proceso que lo Codifica para reducir su longitud -Codificador-, un medio de transferencia del mensaje codificado, un

proceso que decodifica la secuencia de bits recibida para reconstruir el mensaje original -Decodificador-, el que es finalmente entregado al Receptor. Esquemáticamente:



Actividad

A partir de un mensaje original que consta de 91 caracteres, con frecuencias de ocurrencia según la tabla:

Caracter	A	B	C	D	E	F	G	H	I
Frecuencia	15	6	7	12	25	4	6	1	15

I. Calcular el tamaño del mensaje codificado, en bits, si se usa la codificación de Huffman propuesta.
II. Idem a I., pero en bytes.
III. Calcular el tamaño del mensaje codificado, si se usa código ASCII.
IV. Comparar los resultados obtenidos.

b) Árbol de expresión[5].

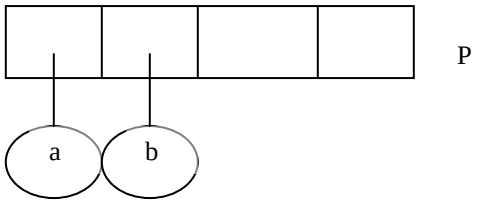
En un árbol de expresión las hojas son *operandos*: constantes o nombres de variables, y los nodos interiores son *operadores*, binarios en este caso.

A partir de una expresión de entrada en posfijo- aquella en la que los operadores suceden a sus operandos-, se genera un árbol de expresión. A tal efecto el método seguido, que se apoya en el uso de una pila de árboles binarios como elementos, es el siguiente:

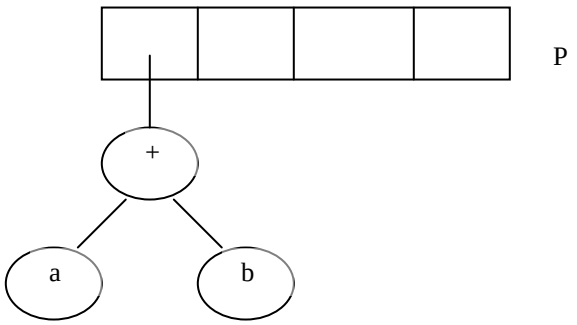
Leer la expresión símbolo por símbolo hasta el final de la misma. Si el símbolo es un operando, crear un árbol de un solo nodo e insertar dicho árbol en la pila. Si el símbolo es un operador, suprimir de la pila los árboles A_1 y A_2 – dos veces el tope-, y construir un nuevo árbol cuya raíz es el operador, y sus hijos izquierdo y derecho los árboles A_1 y A_2 respectivamente. Insertar este nuevo árbol en la pila.

Como ejemplo consideremos la siguiente expresión en postfijo: ab+

Los primeros dos símbolos son operandos, por lo que en los primeros pasos se crean dos árboles, de un nodo cada uno, los que son ingresados en la pila P



El siguiente símbolo de la expresión es un operador, entonces se suprimen los dos árboles de la pila, se genera un nuevo árbol con el operador como raíz, que es insertado en la pila.



Luego de procesada toda la expresión, el árbol resultante queda en la pila. A partir de él, y aplicando los distintos recorridos, puede reconstruirse la expresión en prefijo, en infijo o en postfijo.

Actividad

Diseñe el algoritmo que construya un árbol de expresiones.

[5] Weiss, Mark Allen, op.cit. pág 101.

Árbol Binario de Búsqueda - ABB

Supongamos que necesitamos gestionar los datos de los alumnos de nuestra facultad. La administración de esta información involucra tareas tales como: ingresar nuevos alumnos, eliminar alumnos, modificar datos de alumnos, procesar los datos de todos o de un subconjunto de alumnos según sus claves de identificación, etc.

Antes de abocarnos a resolver las tareas mencionadas, debemos considerar que aspectos de cada alumno, en base a la realidad considerada, deben ser mantenidos. Por ejemplo podemos tener en cuenta: Nombre, DNI, Nº Registro, Fecha de Nacimiento, Domicilio, etc. De las características seleccionadas, destacamos al Nº Registro, ya que a partir de él, es posible individualizar a cada alumno. En tal caso, formalmente, Nº Registro es una función biyectiva que asigna, a cada alumno, un único valor que lo identifica entre todos los alumnos de esta facultad :

NºReg: Alumnos ⟶ { x / x es Número Natural y Menor <= x <= Mayor}

Menor y Mayor son los límites del rango de números de registros válidos.

Puesto que frecuentemente se realizan las tareas mencionadas, sobre datos con características similares a los mencionados, construiremos el TAD ABB - Árbol Binario de Búsqueda-. En este árbol, las operaciones Insertar, Suprimir y Buscar se realizan en *h* accesos como máximo, siendo *h* la altura del árbol binario.

T.A.D. Árbol Binario de Búsqueda

a) Especificación

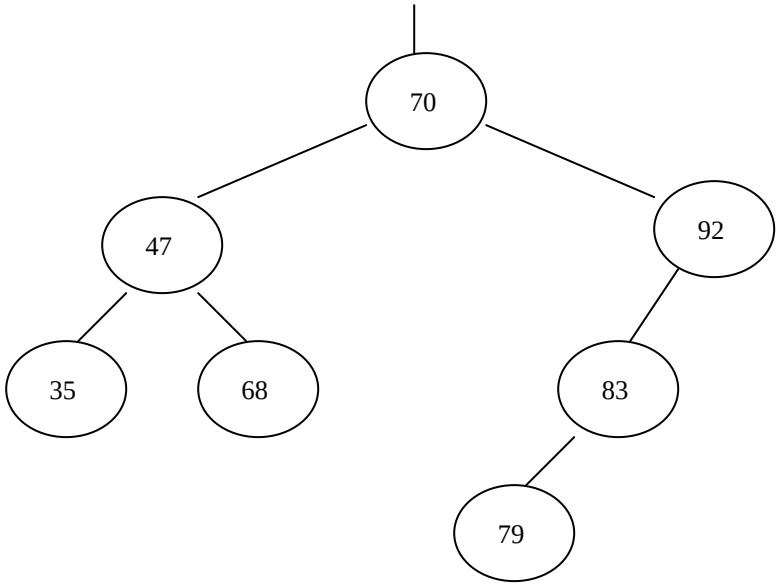
Una aplicación importante de los árboles binarios es su uso en la búsqueda y recuperación de información. Para ello, consideramos que cada nodo tiene un valor de clave, es decir, cada nodo contiene un campo cuyo valor permite identificar en forma única a ese elemento.

A partir de lo expresado por Weiss[6] y por Aho[7], un **Árbol Binario de Búsqueda** es un Árbol Binario en el que:

- Cada nodo contiene un campo clave (lo identifica en forma única dentro del Árbol).
- Los nodos del árbol están ordenados de tal forma que para cualquier nodo x del árbol, si z es un nodo cualquiera del subárbol izquierdo de x, entonces la clave[z] < clave[x] y si z es un nodo cualquiera del subárbol derecho de x, entonces clave[x] < clave[z]. Esta condición es conocida como *Propiedad del Árbol Binario de Búsqueda*.

Presentamos a continuación un ejemplo de árbol binario de búsqueda, con valores de claves pertenecientes al conjunto de los números naturales.

[6] Weiss, Mark Allen, op.cit. pág 105
[7] Aho, Alfred,Hopcroft,John y Ullman, Jeffrey.op. cit. Pág.157.



Los árboles binarios de búsqueda son estructuras de datos para conjuntos de elementos, que permiten realizar operaciones sobre dichos conjuntos de manera mas eficiente que en el caso de listas ordenadas por contenido. Como en ese caso, los elementos del conjunto presentan un orden lineal, establecido sobre el campo clave.

Las operaciones básicas para este TAD son:

- **Insertar** en el (sub)árbol A el elemento X

  Si (sub)árbol vacío
  entonces X nueva hoja- raíz de (sub)árbol.
  Si Clave [X] < Clave [R] (R:nodo raíz de (sub)árbol)
  entonces insertar ((sub)árbol izquierdo(R) , X)
  Si Clave[X] >Clave [R] (R: nodo raíz de (sub)árbol)
  entonces insertar ((sub)árbol derecho(R), X)

- **Suprimir** del (sub)árbol A el elemento X

  Si (sub)árbol vacío
  entonces elemento inexistente.
  Si Clave [X] = Clave [R] (R: nodo raíz de (sub)árbol) entonces
  Si Grado(R) = 0
  entonces eliminar nodo R
  Si Grado(R) = 1
  entonces Padre(R) ⟵ Hijo(R)
  Si Grado(R) = 2
  entonces R ⟵ Máximo ((sub)árbol-izquierdo (R))
  Eliminar (Máximo((sub)árbol-izquierdo (R)))

Máximo refiere al mayor valor de clave de un conjunto.

- **Hijo** en el árbol A, el nodo con clave X del nodo con clave Z
- **Padre** en el árbol A, el nodo con clave X del nodo con clave Z
- **Nivel en el árbol** A del nodo con clave X
- **Hoja** en el árbol A, el nodo con clave X
- **Grado** en el árbol A del nodo con clave X
- **Altura** del árbol A
- **Camino** en el árbol A, desde el nodo con clave X al nodo con clave Z
- Recorrer en **InOrden** el árbol A
- Recorrer en **PreOrden** el árbol A
- Recorrer en **PostOrden** el árbol A
- **Buscar** el nodo con clave X en el árbol A

Actividad

Analice la especificación de la operación Suprimir. Proponga una alternativa a la situación en que el grado del nodo a eliminar es igual a 2.

Sean A: Árbol; X,Z : claves

NOMBRE	*ENCABEZADO*	*FUNCION*	*ENTRADA*	*SALIDA*
Crear	Crear (A)	Inicializa el árbol A	A	A=( )
Insertar	Insertar(A, X)	Ingresa el elemento X en el árbol A, manteniéndolo como árbol binario de búsqueda.	A, X	A con X como hoja, si X $\notin$ A; Error en caso contrario
Suprimir	Suprimir(A, X)	Elimina el elemento X del árbol A, manteniéndolo como árbol binario de búsqueda.	A, X	A sin el nodo que contenía a X, si X $\in$ A; Error en caso contrario
Hijo	Hijo(A, X, Z)	Evalúa si X es hijo de Z	A, X, Z	Verdadero si X es hijo de Z, Falso en caso contrario.

Padre	Padre(A, X, Z)	Evalúa si X es padre de Z	A, X, Z	Verdadero si X es padre de Z, Falso en caso contrario.
Nivel	Nivel(A, X)	Calcula el nivel de X	A, X	Reporta el nivel de X.
Hoja	Hoja(A, X)	Evalúa si X es nodo hoja	A, X	Verdadero si X es nodo Hoja, Falso en caso contrario.
Altura	Altura(A)	Evalúa la altura de A	A	Reporta la altura de A.
Camino	Camino(A, X, Z)	Recupera el camino de X a Z	A, X, Z	Reporta el camino de X a Z.
InOrden	InOrden(A)	Procesa A en Inorden	A	Está sujeta al proceso que se realice sobre los elementos de A
PreOrden	PreOrden(A)	Procesa A en Preorden	A	Está sujeta al proceso que se realice sobre los elementos de A
PostOrden	PostOrden(A)	Procesa A en Postorden	A	Está sujeta al proceso que se realice sobre los elementos de A
Buscar	Buscar(X,A)	Localiza el nodo con clave X en el árbol A	A,X	Reporta los datos asociados con X, si $X \in A$; Error en caso contrario

Actividad

Analice la trascendencia de la operación Buscar respecto a las demás operaciones especificadas.

b) Representación

La estructura de un árbol varía, ya que crece y se contrae dinámicamente. Por ello, y recordando el análisis realizado al momento de construir el TAD Lista, adoptamos inicialmente la representación encadenada del objeto de datos, ya sea por medio del uso de variables generadas dinámicamente o de cursores. La celda correspondiente a cada nodo del árbol deberá tener, además de los campos correspondientes a los datos, otros dos campos, destinados a almacenar las `direcciones` de las raíces de sus subárboles izquierdo y derecho.

Actividad

Construya en C++, el objeto de datos del TAD ABB:

a) Por medio de variables dinámicas
b) Por medio de cursores.

c) Construcción de operaciones abstractas

Sabemos que un objeto es *recursivo*, si en parte está formado por sí mismo o se define en términos de sí mismo. El poder de la recursión radica en la posibilidad de definir un conjunto potencialmente infinito de objetos mediante una proposición finita. Los algoritmos recursivos son adecuados cuando el problema a resolver, la función por calcular o la estructura de datos por procesar ya están definidos en términos recursivos[8]. En este último caso se encuentra la estructura de datos Árbol.

A pesar de lo expresado, Wirth manifiesta también que para ciertas definiciones recursivas, el algoritmo recursivo no es la mejor manera de resolver el problema -ejemplo de ello es el cálculo de los número de Fibonacci. Debemos entonces aprender que no debería utilizarse la recursión cuando la iteración ofrece una solución obvia. El hecho de que los procedimientos recursivos se realicen en máquinas esencialmente no recursivas demuestra que, en la práctica, todo programa recursivo puede transformarse en uno esencialmente iterativo[9].

Como la solución iterativa en muchos casos oscurece la esencia de un problema, nosotros recurriremos a algoritmos recursivos en lugar de iterativos, aun reconociendo que en ciertas ocasiones, éstos pueden representar una solución mas eficiente.

En la especificación de las distintas operaciones de este TAD, subyace la necesidad de descubrir si una clave particular está contenida o no en algún nodo del árbol. Por ejemplo las operaciones Insertar y Suprimir evalúan primero si el elemento a ingresar o eliminar se encuentra ya en el árbol, por lo que una búsqueda del elemento está inmersa. Este escenario ha llevado a distintos autores, con los que coincidimos, a hablar de algoritmos de búsqueda e inserción en lugar de inserción solamente.

Actividad

1- Construya en C++, por medio de algoritmos recursivos, las operaciones Buscar, Insertar e InOrden.
2- ¿Cual es el resultado de la operación InOrden si se considera que el proceso de cada nodo es la presentación de su contenido por monitor?

Una aplicación de los árboles binarios de búsqueda es la construcción de *índices de referencias cruzadas*. Estos índices, que se encuentran generalmente en las últimas páginas de los libros de textos, constan de palabras ordenadas alfabéticamente, con el/los números de página (de línea en nuestro ejemplo) en las cuales dicha palabra aparece en el texto.

[8] Wirth, Niklaus, op.cit. pág 146
[9] Wirth, Niklaus, op.cit. pág 148-150.

Supongamos el siguiente texto en el que los números de la izquierda representan números de línea:

1 . `el editor de lenguaje C
2 . presenta un entorno
3 . similar al entorno
4. de lenguaje Pascal`

El índice de referencias cruzadas para el texto propuesto es el siguiente:

al	3
C	1
de	1, 4
editor	1
entorno	2, 3
el	1
lenguaje	1, 4
Pascal	4
presenta	2
similar	3
un	2

Actividad

Para generar un índice de referencias cruzadas apoyado en un ABB :

a) Adapte el TAD ABB para esta situación.
b) Diseñe el algoritmo que realiza la tarea propuesta.

Árbol Balanceado-Árbol AVL

Como hemos expresado, en distintas operaciones sobre nodos particulares del árbol -Insertar o Suprimir por ejemplo-, está en cierto modo inmersa la operación que verifica si un elemento con un cierto valor de clave está o no en el árbol. De esto se desprende la importancia de considerar la cantidad de comparaciones que se realizan para localizar un nodo con clave determinada, o establecer que dicho nodo no se encuentra en el árbol, pues este aspecto influye de manera directa en el `costo` de ejecución del algoritmo que resuelve las operaciones mencionadas.

En general no sabemos como crecerá el ABB, ni que forma tomará. Si las claves fuesen ingresadas en orden estrictamente ascendente o descendente, el árbol resultante tendrá la apariencia de una lista. En este caso, el esfuerzo promedio de búsqueda es n/2 comparaciones. A la situación presentada podríamos considerarla como la mas desfavorable entre las que pueden presentarse, pues con ***n*** claves tenemos ***n!*** posibles secuencias de ingreso distintas, generándose entonces ***n!*** árboles binarios de búsqueda diferentes.

Analicemos los siguientes conceptos:

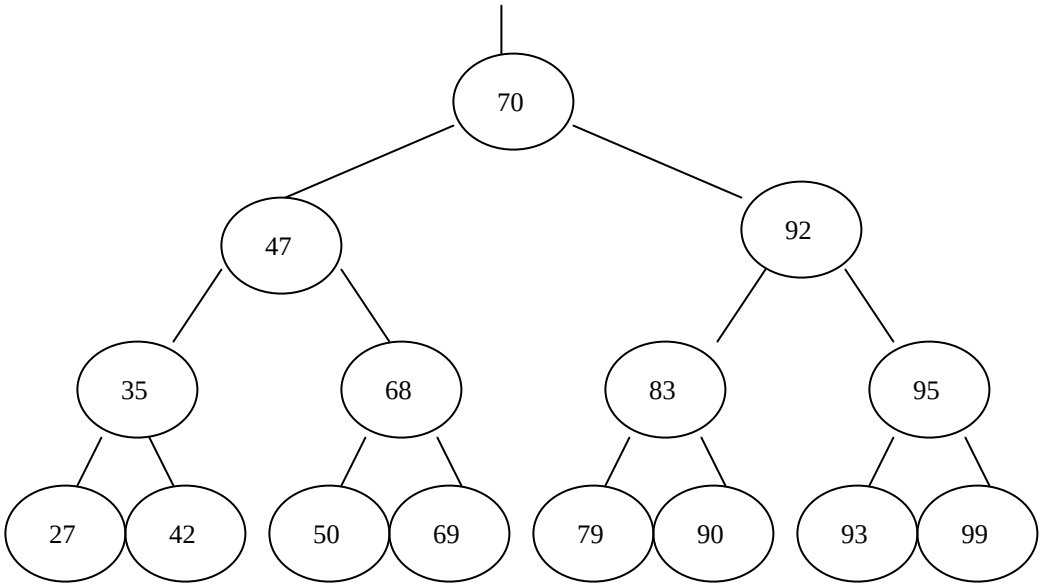
- *Árbol binario relleno*: Es aquel árbol en el que todo nodo o bien es una hoja, o tiene dos hijos.
- *Árbol binario completo*: Es un árbol binario relleno en el que todas las hojas están en el mismo nivel.

¿Cuántas comparaciones se realizan en una búsqueda, en el peor de los casos, en un ABB Completo, considerando al peor de los casos como aquel en el que la clave a buscar se encuentra en un nodo hoja o dicha clave no está en el árbol?

N : Cantidad de Nodos del ABB Completo
C : Cantidad de Comparaciones que se realizan, en el peor de los casos, en el ABB de N nodos.

N	1	3	7	15
C	1	2	3	4

En el siguiente árbol la operación Buscar el nodo con clave 79, realiza 4 comparaciones.



Generalizando, de este análisis surge que:

$\mathbf{N = 2^C - 1} \Rightarrow N + 1 = 2^C \Rightarrow \log_2 (N+1) = C (\log_2 2) \Rightarrow \mathbf{\log_2 (N+1) = C}$

Siendo **C** la cantidad de comparaciones que se realizan, en el peor de los casos, en un ABB completo.

La reflexión previa puede llevarnos a pensar que lo mas conveniente sería generar ABB completos o bien árboles perfectamente equilibrados.

- Un árbol es ***perfectamente equilibrado***, si para cada nodo los números de nodos en sus subárboles izquierdo y derecho difieren cuanto más en uno[10].

Según expresa Wirth, el procedimiento que mantiene el equilibrio perfecto en un árbol es bastante complicado y poco rentable. Una alternativa a esta situación se presenta en la formulación de definiciones menos estrictas de equilibrio, que llevan al desarrollo de procedimientos mas sencillos de reorganización de los árboles.

- Un árbol está ***balanceado*** si y solo si en cada nodo las alturas de sus dos subárboles difieren a lo máximo en uno[11].

Estos árboles reciben el nombre de arboles AVL por ser Adelson- Velski y Landis quienes propusieron esta definición de equilibrio.

Actividad

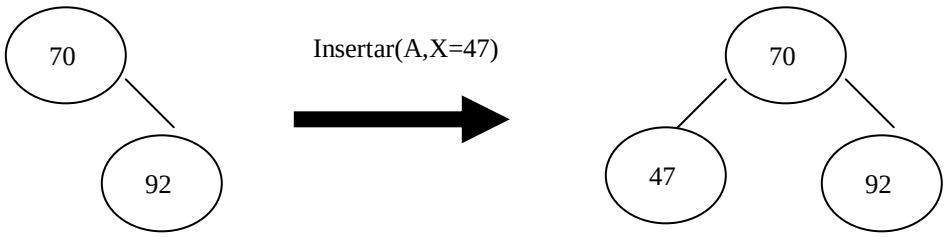
¿Son todos los árboles perfectamente equilibrados, árboles balanceados? ¿Y la condición reciproca?

Inserción en un Árbol Balanceado

Supongamos que se inserta un nuevo nodo en un árbol balanceado, con raíz ***r***, y subárboles izquierdo y derecho *I* y *D* respectivamente. ¿Qué situaciones se presentan si el nuevos elemento, llamémoslo ***X***, se inserta en el subárbol izquierdo?

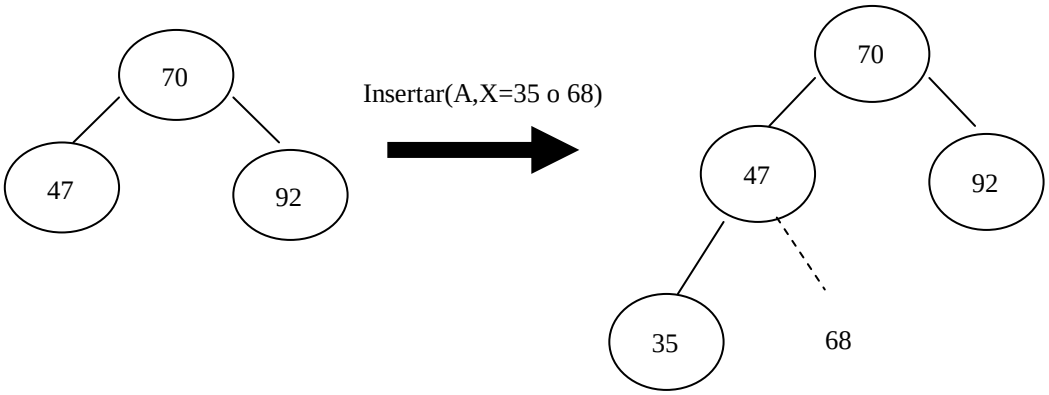
Para responder a esta pregunta, debemos distinguir tres casos:

a) Si altura(I(r)) < altura(D(r)) (previo a la inserción) y *X* se inserta en I(r), el balanceo mejora, pues las alturas de los subárboles se igualan. Por ejemplo:

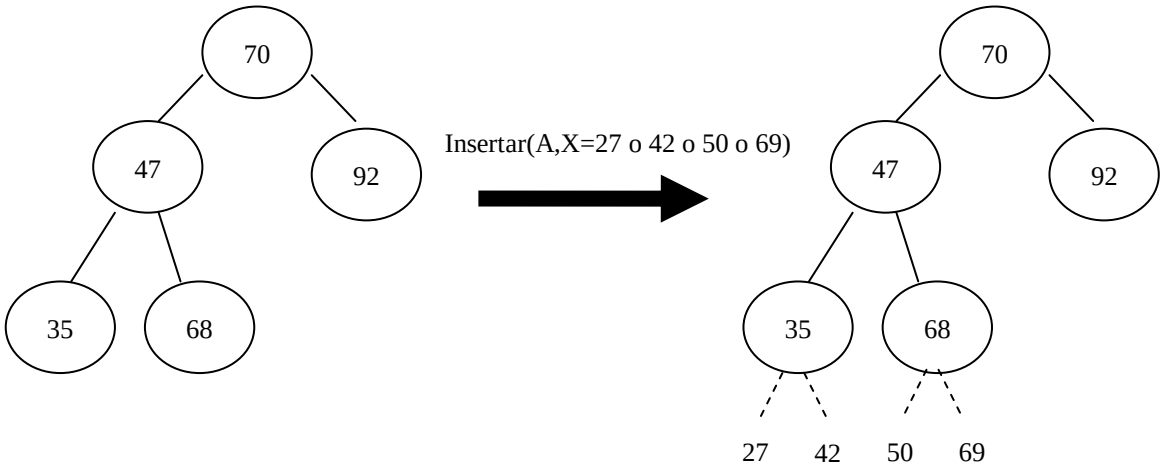


b) Si altura(I(r)) = altura(D(r)) (previo a la inserción) y *X* se inserta en I(r), aumenta la altura del subárbol izquierdo, pero el árbol sigue balanceado.

[10] Wirth, Niklaus. Op. Cit. Pág 215.
[11] Wirth, Niklaus. Op. Cit. Pág 232.



c) Si altura(I(r)) > altura(D(r)) (previo a la inserción) y *X* se inserta en I(r), el árbol se desbalancea, ya que la diferencia de las alturas de los subárboles izquierdo y derecho de r sería mayor que 1. Esta situación se presenta al insertar en el árbol, por ejemplo, las claves 27 o 42 o 50 o 69.



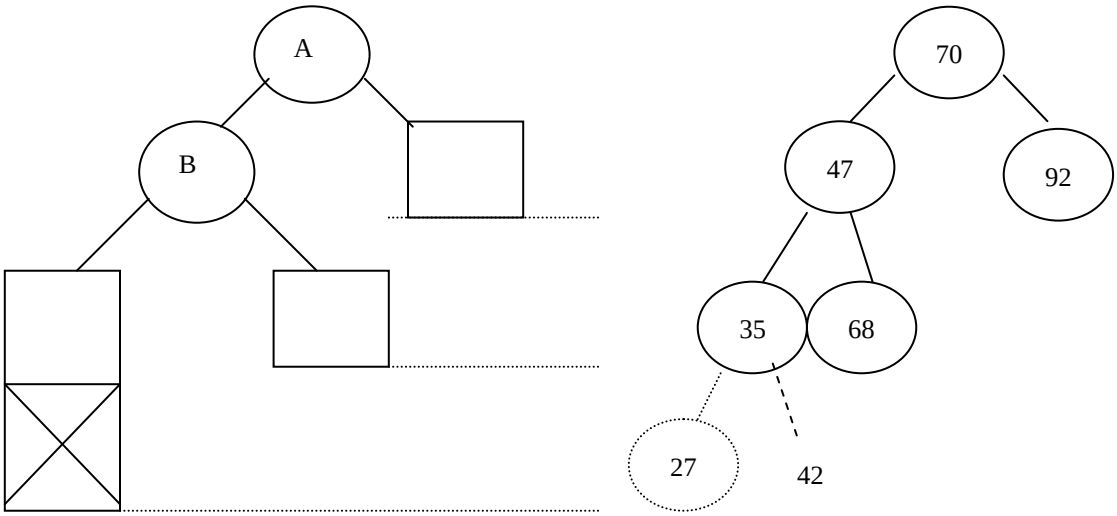
Actividad

Explicite, para cada uno de los casos presentados, el balanceo de cada nodo del árbol antes y después de realizada la inserción del nuevo elemento.

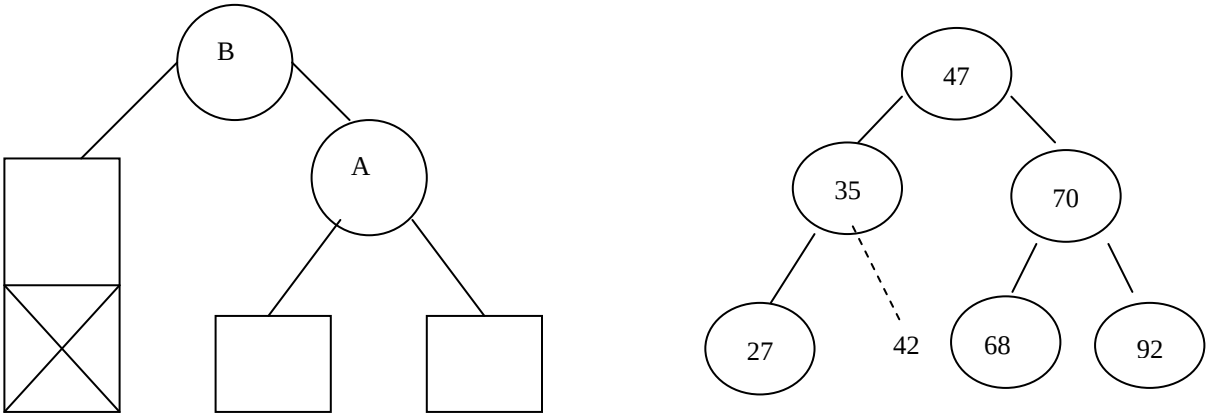
Wirth expresa que un cuidadoso análisis de la situación c) revela que hay dos constelaciones esencialmente diferentes – Caso 1 y Caso 2 presentados a continuación- que requieren tratamiento individual. Las restantes pueden derivarse de esas dos por medio de consideraciones simétricas[12].

[12] Wirth, Niklaus.op. cit. Pág.233.

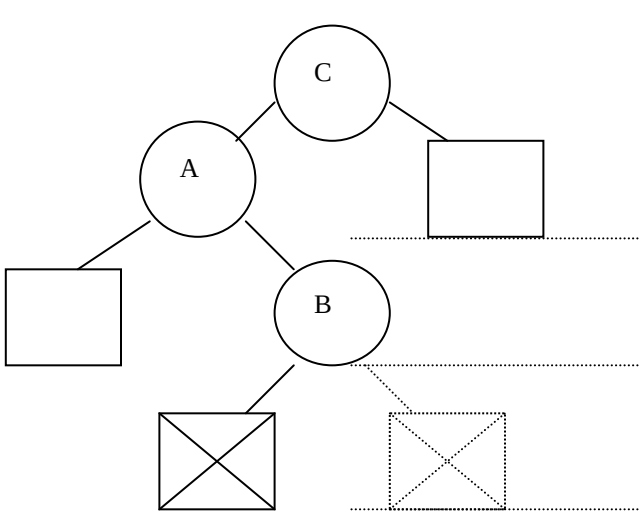
Caso 1: Si en el subárbol izquierdo crece la rama izquierda (o en el subárbol derecho crece la rama derecha) . Esta situación se presenta al insertar en el árbol ejemplo la clave 27 o la clave 42.

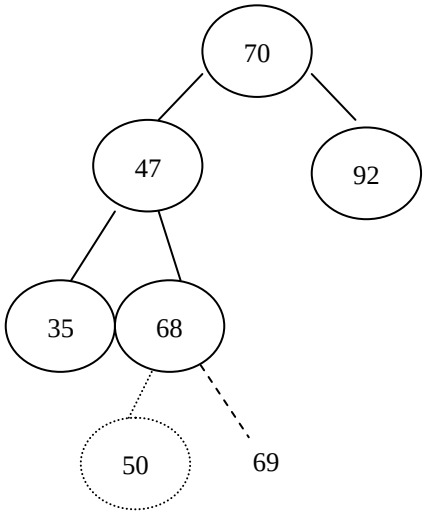


El rebalanceo requerido se realiza a partir de la secuencia de pasos que se engloban en lo que Wirth denomina *rotación simple.*

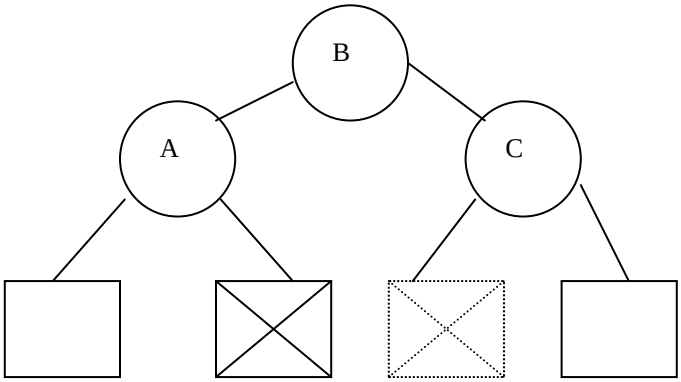


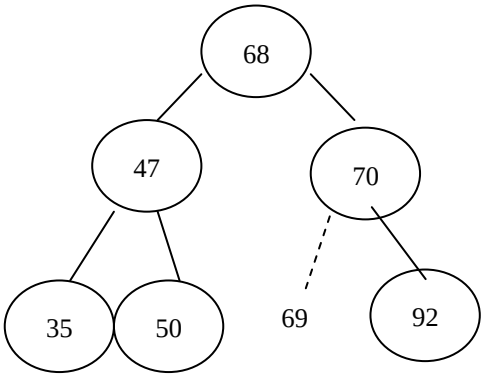
Caso 2: Si en el subárbol izquierdo crece la rama derecha, (o en el subárbol derecho crece la rama izquierda)- . En el ejemplo, esto ocurre al insertar las claves 50 o 69.





El rebalanceo en este caso se realiza a partir de una *rotación doble*.





Como controlar entonces el crecimiento del Arbol Balanceado?

Al insertar cada nuevo elemento, retroceder a lo largo de *su* trayectoria de búsqueda comparando en cada nodo las alturas de sus subárboles izquierdo y derecho para detectar si se produjo o no desbalanceo.

Si este análisis se realiza a partir de un árbol balanceado en el que se adopte la definicion del objeto de datos del arbol binario de búsqueda - registro que contiende la informacion

(siendo uno de sus campos el campo clave), y las direcciones de sus subárboles –, el control de las alturas de los subárboles de cada nodo, llevaría aparejado un alto costo.

En el algoritmo propuesto por Wirth[13], la definicion del objeto de datos correspondiente a cada nodo –r-, incorpora un nuevo campo –campo Balanceo- que contiene la información sobre la diferencia de las alturas de los subárboles del nodo.

Campo Balanceo (r) = altura(D(r)) – altura(I(r))

La incorporación del estado de balanceo de cada nodo, reduce el gasto en tiempo de ejecución –overhead- que se generaría por los sucesivos cálculos de alturas. El algoritmo decide si hace falta rebalanceo o no en cada nodo de la trayectoria de búsqueda, a partir del valor del campo balanceo y de la informacion de que ha aumentado la altura de alguno de sus subárboles. Para determinar si corresponde rotacion simple o doble, el algoritmo inspecciona el campo balanceo correpondiente al hijo, raíz del subárbol que arrojó el desbalanceo.

Actividad

1- Al momento de ejecutar el algoritmo de inserción propuesto para arbol balanceado, explicite el valor del campo balanceo del nodo raiz del subárbol que arroja desbalanceo, para cada caso de rebalanceo.
2- Analice cual sería el `costo` de ejecución de un algoritmo de inserción en árbol balanceado, que adopte la definicion del objeto de dato del ABB.

Árbol B

Hasta este momento hemos dirigido nuestra atención a árboles binarios. Este tipo de árboles no es útil para representar por ejemplo, una relación de descendencia en la que una persona puede tener más de dos hijos. Estos árboles, de grado mayor que dos, se denominan *Árboles Multicamino*.

En general las operaciones vinculadas a su manipulación, podrían ser las ya especificadas; lo que restaría es revisar cual sería la representación mas adecuada del objeto de datos, al momento de construir este TAD. Podríamos por ejemplo preasignar espacio para el número máximo de descendientes dentro de cada nodo pero esto generaría un gasto innecesario de memoria si el factor de ocupación efectiva es pequeño. Por esta razón, Wirth[14] propone utilizar una definición de nodo con dos enlaces: uno al primer hermano y otro al primer hijo. Si bien esta representación puede parecerse a la de un árbol binario, la semántica de cada enlace, y su posterior administración varían de las ya tratadas.

Un área de aplicación de los árboles multicamino es la relacionada con la construcción y mantención de árboles de búsqueda a gran escala, que se almacenan en memoria secundaria.

Si tuviésemos que almacenar datos de 1 millon de elementos y elegimos para ello un árbol balanceado, se requerirán :

[13] Wirth, Niklaus. Op. Cit. Pág 237.
[14] Wirth, Niklaus. Op. Cit. Pág 256.

$$\text{Log}_2\ 10^6 \quad = 20$$

comparaciones, en el peor de los casos, para localizar un elemento particular. Si además cada comparación requiere un acceso a disco, estaríamos hablando de 20 accesos a disco!

Con la intención de reducir la cantidad de accesos involucrados cuando se administra gran cantidad de información, incorporamos un tipo particular de árbol multicamino.

En este árbol cada vez que se accede a un elemento, se recupera un conjunto de ítems sin costo adicional, pues el árbol es organizado en subárboles y todos los nodos de cada subárbol son accedidos en forma simultánea.

A cada uno de estos subárboles se los denomina página, y el acceso a una página involucra un acceso a disco.

Si en cada página colocamos por ejemplo 100 nodos y, si se mantiene un crecimiento controlado del árbol, la cantidad de accesos en el peor de los casos sería:

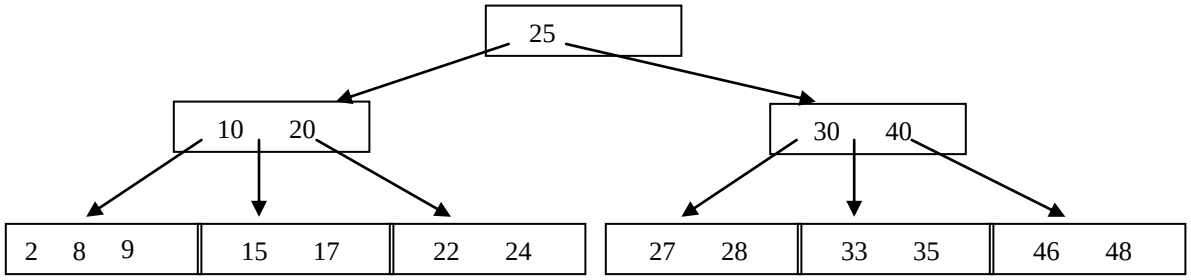
$$\text{Log}_{100}\ 10^6 = 3 \text{ accesos}$$

R.Bayer y E..M..McCreight postularon, según expresa Wirth, un criterio muy razonable al respecto: cada página (salvo una) contiene entre *n* y *2n* nodos para determinada constante n. De ahí que, en un árbol con N elementos y un tamaño máximo de página de 2n nodos por página, en el peor caso requiere $\log_n N$ accesos a páginas. También es importante el factor de utilización de la memoria, que es por lo menos del 50%, ya que las páginas siempre están al menos llenas a la mitad.

Las estructuras de datos subyacentes reciben el nombre de ***Árboles B*** y tienen las siguientes características; se dice que n es el *orden* del árbol B[15]

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

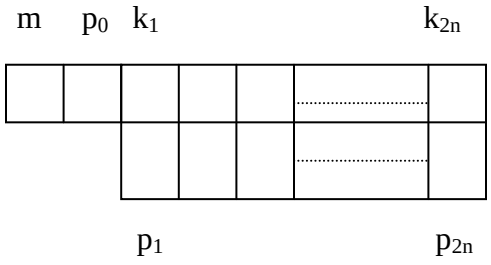
El siguiente ejemplo muestra un árbol B de orden 2 con 3 niveles. Todas las páginas pueden contener 2, 3 o 4 elementos, excepto la raiz que puede contener 1 elemento (menos de n)



[15] Wirth, Niklaus. Op.cit. pág 258.

Para la elección del orden n del árbol B, deben tenerse en cuenta distintos aspectos; por ejemplo el *n* debería ser tal que permita el acceso y recuperación de una página completa, colaborando de esta manera a que las comparaciones involucradas en una búsqueda dentro de esa página sean todas realizadas en memoria principal, esto es, no requieran nuevos accesos a memoria secundaria.

El algoritmo de búsqueda e inserción de una clave propuesto por Wirth, estructura cada página de la siguiente manera:



donde:

m : cantidad de claves en la página; n<=m<=2n excepto para la raíz

k_i : clave; 1<=i<=m

p_i : enlace a páginas, 0<=i<=m, tal que

p_0 : dirección de la página que contiene claves menores que k_1

p_i : dirección de la página que contiene claves mayores que k_i y menores que k_{i+1}

p_m : dirección de la página que contiene claves mayores que k_m

¿Cómo se realiza la búsqueda de una clave *x*?

Dentro de una página particular, las claves se encuentran en orden ascendente, por lo que es posible aplicar una búsqueda binaria o secuencial. Si la búsqueda de *x* dentro de la página corriente fracasa - asumiendo que en esa página se encuentran las claves k_1, k_2, ...,k_m-, la búsqueda de ese elemento puede continuar según se especifica en las siguientes situaciones:

1) $k_i < x < k_{i+1}$ (1 <= i < m) entonces la búsqueda continua por la página apuntada por p_i
2) $k_m < x$; entonces la búsqueda continua por la pagina apuntada por p_m
3) $x < k_1$; entonces la búsqueda continua por la pagina apuntada por p_0

¿Cuál es el proceso de inserción de una clave dada?

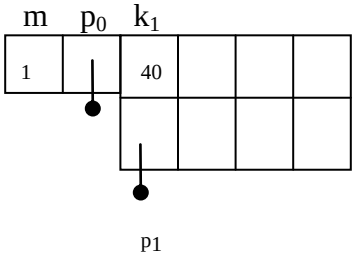
El algoritmo propuesto es de búsqueda e inserción. Cuando determina que la clave no está y la página hoja contiene m < 2n elementos, entonces la nueva clave es ingresada en esa página. Si esa página está llena, las 2n + 1 claves se distribuyen en dos páginas, siendo la clave del medio de la secuencia `subida` a la página padre. Si la página padre tiene

lugar para la nueva clave (m<2n), entonces esta clave es ingresada; caso contrario, el desdoblamiento de páginas (split) se repite, pudiendo provocar inclusive el aumento de la profundidad del árbol.

Actividad

1 -A partir de la ejecución del algoritmo de inserción en un árbol B de orden 2, grafique el estado del árbol luego de ingresar cada una de las siguientes claves:

x = 40



x = 35

x = 70

x = 22

x = 80

2-¿Cuál es el grado de un árbol B de orden n?

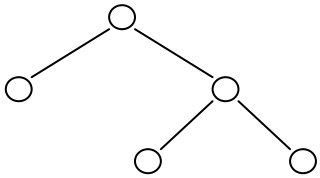
3-Explicite el o los criterios que, a partir del TAD Lista ordenada por contenido, han conducido el camino hasta Árboles B.

Montículo Binario

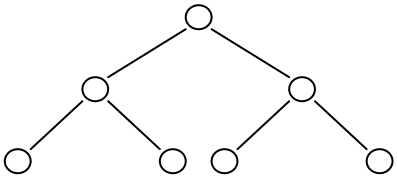
Un Montículo Binario es un tipo de árbol binario, que a diferencia de los vistos anteriormente, es almacenado secuencialmente; esto es, prescinde de enlaces. En otras palabras, el montículo binario es un tipo de árbol binario para el cual se adopta la representación secuencial del objeto de datos, al momento de construir el TAD.
Ya hemos incorporado los árboles binarios completos, a partir de ellos ahora presentamos a los árboles binarios semicompletos.

- ***Árbol Binario Semicompleto***[16]: Un Árbol Binario Semicompleto de ***n*** nodos, se forma a partir de un árbol binario completo de ***n+q*** nodos, quitando las ***q*** hojas extremo derechas del árbol binario completo.

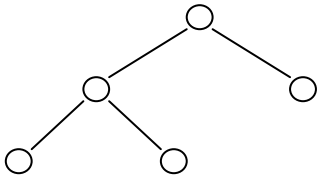
[16] Heileman, Gregory. **Estructuras de Datos, Algoritmos y Programación Orientada a Objetos.** Mc Graw Hill. España. 1998. Pag. 285.



a) Árbol Binario Relleno



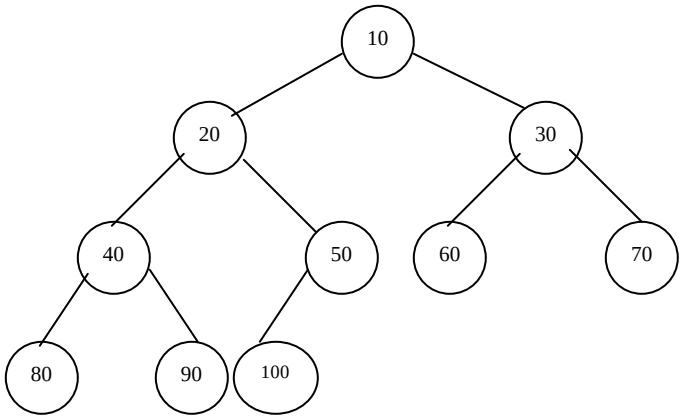
b) Árbol Binario Completo



c) Árbol Binario Semicompleto

El árbol binario semicompleto de 5 nodos- fig. c) se forma a partir de un árbol binario completo de 5+2 nodos- fig.b), al que se le han quitado las 2 hojas de extremo derechas.

- ***Montículo Binario[17]:*** Un *Montículo Binario* es un árbol binario semicompleto en el que el valor de clave almacenado en cualquier nodo es menor o igual que el valor de clave de sus hijos.



Montículos Binarios y Colas de Prioridad

Sabemos que en un sistema en el que múltiples procesos son ejecutados concurrentemente por un solo procesador, se sigue alguna política de panificación. Un política alternativa consiste en disparar los procesos respetando el orden en el que requieren del procesador, lo

[17] Heileman, Gregory. Op. Cit. Pag165

que conlleva asociada la idea de una Cola de procesos (Lista FIFO) que esperan el servicio de un servidor ,que en este caso es el procesador.
Ahora bien, si se está ante una situación en la que los procesos deben ser ejecutados según su importancia, en relación a los restantes, ya no es adecuado el uso de una Lista FIFO, sino de una Cola de Prioridad. Esa importancia a la que hacemos referencia puede ser cuantificada por medio de un valor al que denominamos prioridad.
A veces resulta útil pensar que las claves, en las que se apoya el concepto de montículo binario, en este contexto representan a prioridades. Como las claves/prioridades obedecen a una relación de orden total, entonces pueden identificarse los elementos con prioridades mayores o menores. Esto permite que en un sistema que tiene un procesador y múltiples procesos, estos puedan ser ejecutados según sus prioridades. Esta situación puede ser reproducida en un ambiente de impresión, en el que el recurso compartido, servidor, es una impresora, y los que requieren servicio son distintos trabajos a los que se les asigna una prioridad según su extensión. En este caso, a las tareas se les podrían asignar prioridades adoptando el siguiente criterio, menor extensión-mayor prioridad, ya que los trabajos mas cortos liberan mas rápido al recurso impresora.
En una cola de prioridad el primer elemento, que es el que tiene la máxima prioridad, es el próximo elemento a ser eliminado. Un nuevo elemento que llega a la cola, es ubicado según su prioridad.
Los valores –claves- que representan prioridades deben interpretarse de la siguiente manera: a menor valor-mayor prioridad, por lo que la máxima prioridad se encuentra en la raíz del árbol.
Estas ideas pueden ser mantenidas a través del uso de una lista ordenada por campo clave, o como lo hacemos en este documento, por medio de un montículo binario. Por ello, vamos a reconocer como operaciones abstractas relevantes a: aquella que ingresa un nuevo elemento en la cola de prioridad -Insertar, y la que elimina el elemento de máxima prioridad -Eliminar-Mínimo.

TAD Montículo Binario

a)Especificación de las operaciones abstractas

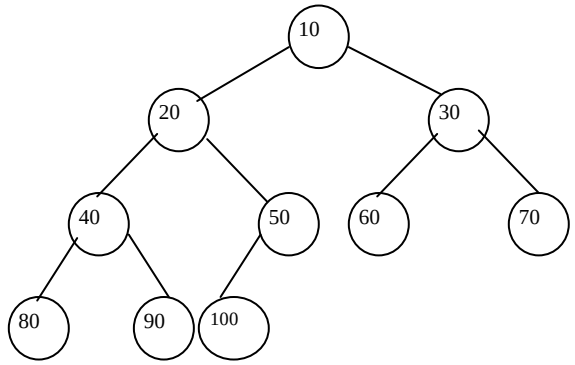
Sea M un Montículo Binario y X una Clave

NOMBRE	*ENCABEZADO*	*FUNCIÓN*	*ENTRADA*	*SALIDA*
Insertar	Insertar(M, X)	Ingresa el elemento X al montículo M	M, X	M con el nuevo elemento
Eliminar_Mínimo	Eliminar_Mínimo (M, X)	Suprime del montículo M el elemento de máxima prioridad- mínimo valor de clave	M	M y X: elemento de máxima prioridad

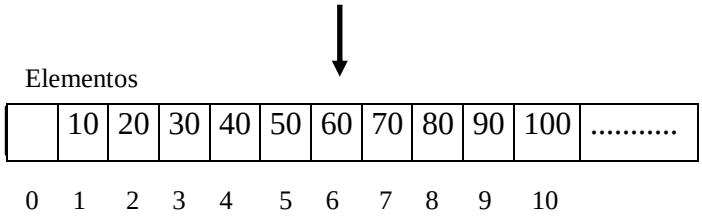
b)Representación

La particularidad del TAD Montículo Binario radica en que el objeto de datos es representado secuencialmente, en esta oportunidad por medio de un arreglo. En este arreglo – Elementos, cada componente soporta un nodo del montículo.

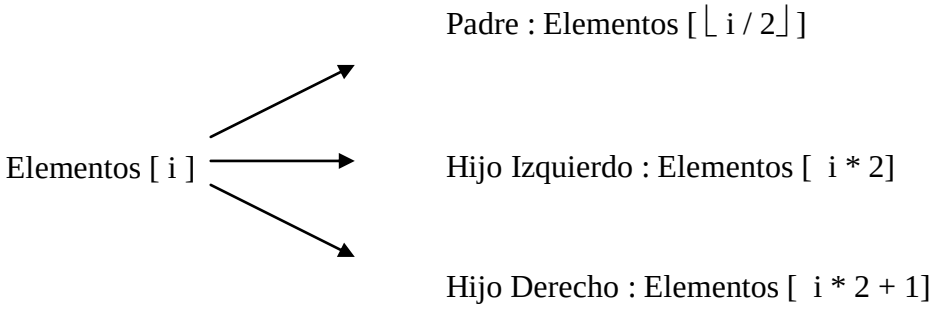
Montículo Binario desde una perspectiva conceptual



Montículo Binario en su almacenamiento



En esta representación los nodos son almacenados a partir de la componente referenciada por medio del subíndice **1.** Esto hace posible que desde cualquier nodo se pueda acceder a su padre, o a sus hijos izquierdo o derecho, a partir de las siguientes transformaciónes:



Así, el nodo que se encuentra en la posición 2 -20-, tiene a su padre en la componente de la posición 1 –10-, y a sus hijos izquierdo y derecho en las componentes de las posiciones 4 y 5 respectivamente – 40 y 50. Es claro que el nodo almacenado en la componente 1 del arreglo – raíz del montículo, no tiene padre. En la componente 0 debería almacenarse la cantidad de nodos del montículo, esto es *n.*

d) Construcción de las operaciones abstractas

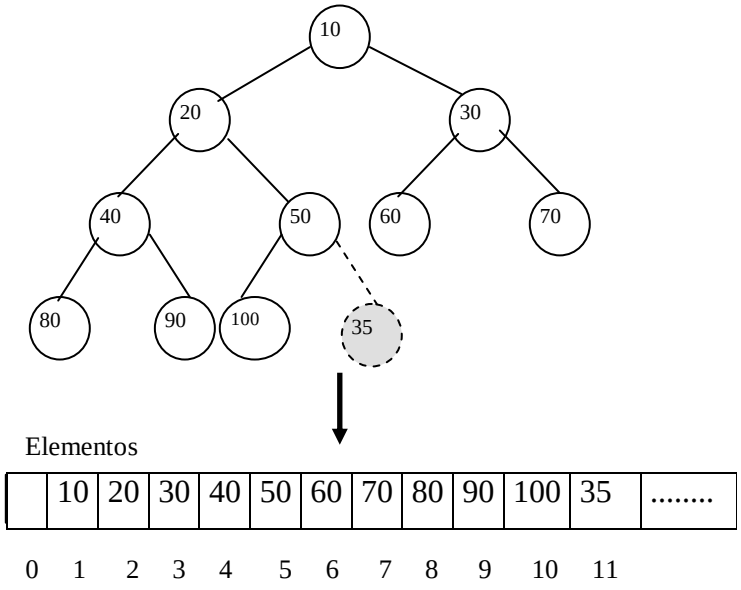
Las operaciónes *Insertar* y *Eliminar_Mínimo*, deben garantizar que en el Montículo Binario se mantengan en todo momento las siguientes dos propiedades:

- *Propiedad de Estructura*, que tiene que ver con que el objeto de datos sea un árbol binario semicompleto.
- *Propiedad de Orden*, que es la que controla la relación entre los valores de las claves de cada nodo respecto a los valores de claves de sus hijos, esto es, el valor de clave almacenado en cualquier nodo debe ser menor o igual que el valor de clave de sus hijos.

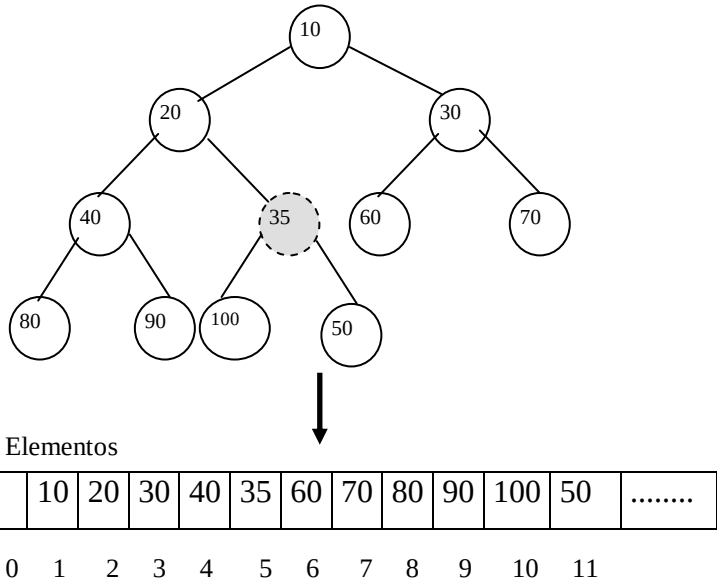
Insertar(M,X)

Al ingresar el elemento X con valor 35, en el montículo M, se siguen los siguientes pasos:

a) El nuevo nodo debe ser añadido como hoja extremo derecha del último nivel del árbol, para mantener la propiedad de estructura.
En el espacio de almacenamiento, el nuevo elemento es cargado detrás de la última componente ocupada del arreglo.



b) Ahora el nodo con valor de clave 35 debe ser colocado en la posición correcta, intercambiándolo con el padre mientras su valor sea menor. Con estas acciones se mantiene la propiedad de orden.



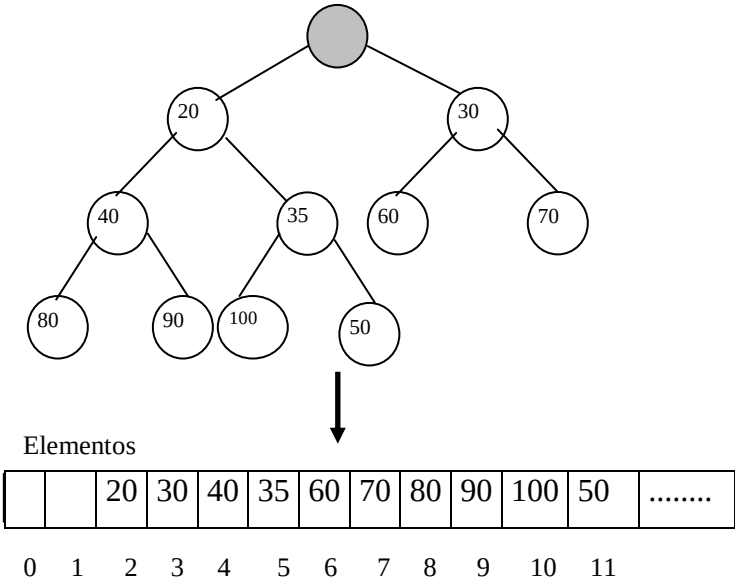
Para colocar al nuevo elemento en la posición de almacenamiento correcta, se lo compara con el padre y se lo intercambia mientras sea menor.
El padre del Elemento[11] –35- se encuentra en la posición Elemento[$\lfloor 11 / 2 \rfloor$] -50. Como 35 es menor, entonces se lo intercambia. Nuevamente se compara con el padre, pero como 35 no es menor que 20, entonces Elementos[5] es la posición que le corresponde en almacenamiento.

Eliminar_Mínimo (M, X)

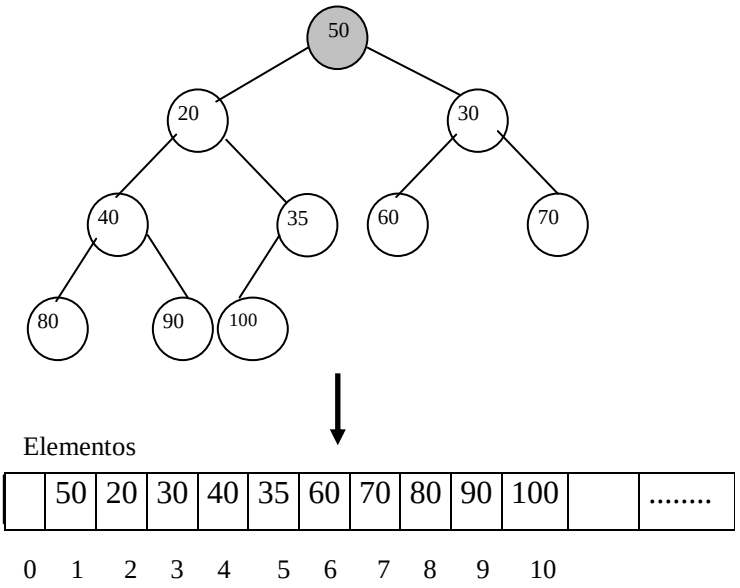
Esta operación extrae del montículo binario el elemento con menor valor de clave, que es quien tiene la mayor prioridad. Este elemento es el que se encuentra en Elementos[1], esto es, en la componente con subíndice 1. Los tres pasos a seguir son:

a) Se recupera el elemento de menor valor de clave:

   X ← Elementos [1] esto es X recibe el valor 10

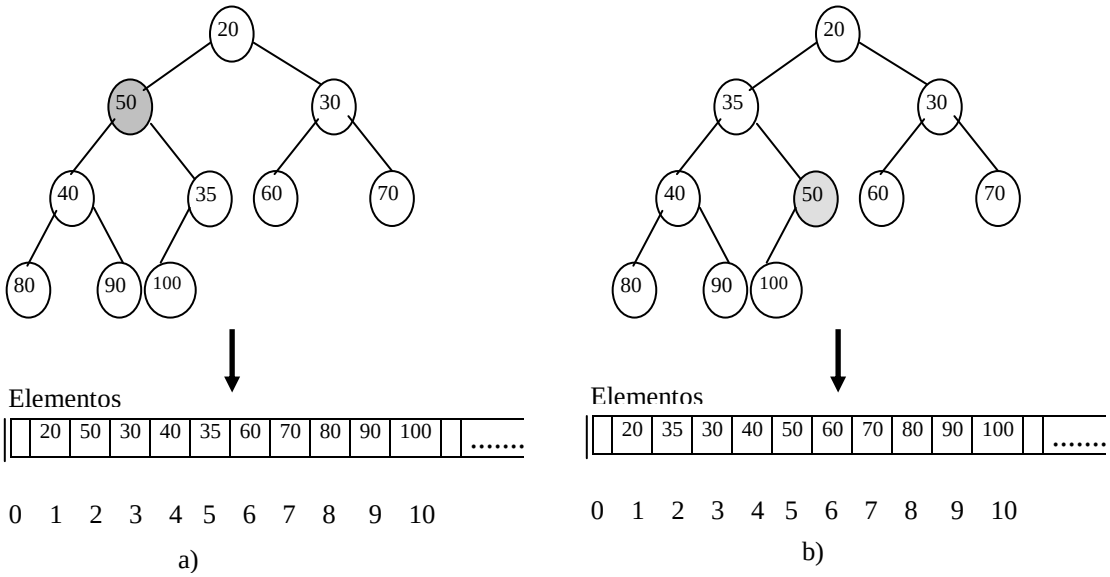


b) El nodo raíz es reemplazado por la ultima hoja de la derecha, para que se respete la propiedad de estructura. Esto es, en Elementos[1] se carga el último componente del arreglo.



c) El elemento que se encuentra en la raíz debe ser comparado con sus hijos. Si el elemento es mayor que sus hijos, se lo intercambia con el menor de ellos. Esto es, Elementos[1] que tiene el valor 50, es comparado con sus hijos izquierdo y derecho, que se encuentran en las componentes 2 y 3. Los valores de estas componentes, 20 y

30, son menores que 50, por lo que se lo intercambia con 20 –fig a). Nuevamente se lo compara con sus hijos, que se encuentran en las componentes 4 y 5, intercambiándolo con el valor 35 que es el menor –fig b). Como el valor 50 es menor que 100, entonces 50 ha alcanzado su ubicación en almacenamiento en la componente 5.
En este paso se realizan los intercambios necesarios para que en el montículo se mantenga la propiedad de orden.



Actividad
Investigue porque es adecuado manipular una Cola de Prioridad por medio de un Montículo Binario , en lugar de una Lista ordenada por Contenido.

Código en C++ correspondiente al algoritmo de **Árbol Balanceado** propuesto por N.Wirth

```
#include<stdio.h>
#include <conio.h>
#include<malloc.h>

struct nodo
          {
           int key,con;
           int bal;
           struct nodo *izq,*der;
          };
typedef struct nodo *punt;

void crear(punt &p)
{
  p=NULL;
}

void insertar(punt &p,int x,int h)
{
 punt p1,p2;
 if(p==NULL)/*insertar*/
  {
   p=(nodo*)malloc(sizeof(nodo));
   h=1;
   p->key=x;
   p->con=1;
   p->izq=NULL;
   p->der=NULL;
   p->bal=0;
  }
  else
   if(p->key>x)
       {
        insertar(p->izq,x,h);
        if(h)/*La rama izquierda crecio*/
        switch(p->bal)
        {
         case 1:{
                p->bal=0;
                h=0;
                break;
                }
         case 0:{
                p->bal=-1;
                break;
```

```
        }
 case-1:{/*Rebalancear*/
        p1=p->izq;
        if(p1->bal==-1)/*Rotacion simple lado izquierdo*/
          {
          p->izq=p1->der;
          p1->der=p;
          p->bal=0;
          p=p1;
          }
        else/*Rotacion doble lado derecho*/
          {
          p2=p1->der;
          p1->der=p2->izq;
          p2->izq=p1;
          p->izq=p2->der;
          p2->der=p;
          if(p2->bal==-1)
            p->bal=1;
          else
            p->bal=0;
          if(p2->bal==1)
            p1->bal=-1;
          else
            p1->bal=0;
          p=p2;
          }
         p->bal=0;
         h=0;
        }/*fin Case*/
    }/*fin switch*/
 }/*fin if*/

else
 if(p->key<x)
  {
    insertar(p->der,x,h);
    if(h)/*La rama derecha crecio*/
     switch(p->bal)
      {
       case -1:{
                p->bal=0;
                h=0;
                break;
               }
       case 0:{
                        p->bal=1;
```

```
                break;
               }
          case 1:{/*Rebalancear*/
               p1=p->der;
               if(p1->bal==1)/*Rotacion simple lado derecho*/
                {
                 p->der=p1->izq;
                 p1->izq=p;
                 p->bal=0;
                 p=p1;
                 }
               else/*Rotacion doble lado izq*/
                {
                 p2=p1->izq;
                 p1->izq=p2->der;
                 p2->der=p1;
                 p->der=p2->izq;
                 p2->izq=p;
                 if(p2->bal==1)
                   p->bal=-1;
                 else
                   p->bal=0;
                 if(p2->bal==-1)
                   p1->bal=1;
                 else
                   p1->bal=0;
                 p=p2;
                 }
               p->bal=0;
               h=0;
                }/*fin case*/
              }/*fin switch*/
              }/*fin if*/
        else p->con++;
}


void balani(punt &p,int &h)
{punt p1,p2;
int b1,b2;
switch(p->bal)
{case -1:{p->bal=0;
         break;}
  case  0:{p->bal=1;
         h=1;
         break;}
  case  1:{/*rebalanceo*/
```

```
            p1=p->der;
            b1=p1->bal;
            if(b1>=0)/*rotacion simple*/
            {
             p->der=p1->izq;
             p1->izq=p;
             if(b1==0)
             {
              p->bal=1;
              p1->bal=-1;
              h=0;
              }
              else
              {
              p->bal=0;
              p1->bal=0;
              }
              p=p1;
             }/*fin rotacion simple*/
              else{/*rotacion doble*/
                  p2=p1->izq;
                  b2=p2->bal;
                  p1->izq=p2->der;
                  p2->der=p1;
                  p->der=p2->izq;
                  p2->izq=p;
                  if(b2==1)
                   p->bal=-1;
                  else
                   p->bal=0;
                  if(b2==-1)
                   p1->bal=1;
                  else
                   p1->bal=0;
                  p=p2;
                  p2->bal=0;
                  }/*fin rotacion doble*/
                }/*fin case*/
               }/*fin switch*/
}/*fin balani*/

void baland(punt &p,int &h)
{
 punt p1,p2;
 int b1,b2;
 switch(p->bal)
 {
```

```
  case  1:{
        p->bal=0;
        break;
        }
  case  0:{
        p->bal=-1;
        h=0;
        break;
        }
  case  -1:{/*rebalanceo*/
        p1=p->izq;
        b1=p1->bal;
        if(b1<=0)/*rotacion simple*/
        {
         p->izq=p1->der;
         p1->der=p;
         if(b1==0)
         {
          p->bal=-1;
          p1->bal=1;
          h=0;
         }
         else
          {
         p->bal=0;
         p1->bal=0;
          }
         p=p1;
        }/*fin rotacion simple*/
        else{/*rotacion doble*/
            p2=p1->der;
            b2=p2->bal;
            p1->der=p2->izq;
            p2->izq=p1;
            p->izq=p2->der;
            p2->der=p;
            if (b2==-1)
            //if(b2==1)
             p->bal=1;
            else
             p->bal=0;
            if (b2==1)
//              if(b2==-1)        //****
             p1->bal=-1;
            else
             p1->bal=0;
            p=p2;
```

```
                p2->bal=0;
               }/*fin rotacion doble*/
             }/*fin case*/
           }/*fin switch*/
}/*fin baland*/

void sup(punt &r,int &h, punt &c)
  {
   if((r->der)!=NULL)
   {
    sup(r->der,h,c);
    if (h)
     baland(r,h);
    }
   else{
        c=(nodo*)malloc(sizeof(nodo));
        c->key=r->key;
        c->con=r->con;
        c=r;
        r=r->izq;
        h=1;
        }
  }/*fin sup*/


void suprimir(punt &p,int x,int h)
{
  punt q;
  if(p==NULL)
   printf("\nLa llave no esta en el arbol");
  else
    if(p->key>x)
       {
        suprimir(p->izq,x,h);
        if(h)
          balani(p,h);
        }
        else
           if(p->key<x)
             {
              suprimir(p->der,x,h);
              if(h)
                baland(p,h);
              }
              else/*eliminar p->*/
              {
               q=p;
```

```
            if(q->der==NULL)
            {
             p=q->izq;
             h=1;
            }
             else
               if(q->izq==NULL)
                 {
                  p=q->der;
                  h=1;
                 }
                else
                 {
                  punt c;
                  sup(q->izq,h,c);
                     //agrego
                  p->key=c->key;
                  p->con=c->con;
                     //fin
                  if(h)
                    balani(p,h);
                 }

            printf("%d",p->key);
            ;
            getchar();
           }
        }

void mostrar(punt p)
{  static int i=0;
 if(p!=NULL)
  {
   printf("\n %d   ",p->key);
   printf("\n izquierda  ");
   mostrar(p->izq);
   printf("\n derecha  ");
   mostrar(p->der);

  }
}

void main ()
{
 punt p;
 int op,ele,letra;
crear(p);
```

```
do
{
  clrscr();

   printf("\n\n Menú \n");
   printf("\n 1_ Insertar\n");
   printf("\n 2_ Suprimir\n");
   printf("\n\ 3_ Mostrar\n");
   printf("\n 4_ Salir\n");
   printf("\n \n");

 printf("Ingrese opción  ");
 scanf("%d",&op);
 getchar();
 switch (op)
 {
    case 1:
     {
      printf("Ingrese elemento a insertar, el ingreso finaliza con 0  ");
      scanf("%d",&ele);
      getchar();
      while (ele!=0)
       {
        insertar(p,ele,letra);
        printf("Ingrese elemento a insertar, el ingreso finaliza con 0  ");
        scanf("%d",&ele);
        getchar();
       }
       break;
     }
   case 2:
     {
      printf("Ingrese elemento a suprimir, el ingreso finaliza con 0  ");
      scanf("%d",&ele);
      getchar();
      while (ele!=0)
       {
        letra=0;
        suprimir(p,ele,letra);
        printf("Ingrese elemento a suprimir, el ingreso finaliza con 0  ");
        scanf("%d",&ele);
        getchar();
        }
        break;
      }
  case 3:
     {
```

```
      mostrar(p);
      getchar();
      break;
     };
 }
 }
 while (op!=4);
}
```

Código en C++ correspondiente al algoritmo de **Árbol B** propuesto por N. Wirth

```
/* Este algoritmo permite implementar un árbol B de orden n=2
Probar el algoritmo con por ejemplo los siguientes datos:

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

*/
#include<conio.h>
#include<stdio.h>
#include<alloc.h>
#define n 2
struct item{int kyy;
          int count;
          struct page *p;
          };
struct page{int m;
          struct page *p0;
          item e[2*n];  //n orden del árbol
          };

typedef struct page *puntero;

 // Función que permite insertar claves en el árbol B

void insertar(int x,puntero &a,int &h,item &v)
{int l,i,r;
puntero b;
item u;
if(a==NULL)   //carga el registro que contendrá la clave
{h=1;
  v.kyy=x;
  v.count=1;
  v.p=NULL;
  }
  else  // Busca donde insertar una nueva clave
  {l=1;
   r=(a->m);
    while (l<r)
    {i=(l+r)/2;
    if((a->e[i].kyy<=x))
      l=i+1;
    else
      r=i;
    }
   r--;
   if((r>=0)&&(a->e[r].kyy==x))   //repetidos
```

```
  {
  a->e[r].count++;
  h=0;
  }
else{if ((r==0)&&(a->e[r].kyy>x)) // los mas chicos
     insertar(x,a->p0,h,u);
    else
     insertar(x,a->e[r].p,h,u); // los mayores que...
   if (h)
    if(a->m<2*n)
      {h=0;
       a->m++;
       for(int i=a->m-1;i>=(r+1);i--) // corrimiento para insertar
          a->e[i]=a->e[i-1];
       if (a->e[r].kyy>x)// se ubica a el nuevo elemento
        a->e[r]=u;
       else
        a->e[r+1]=u;
       }
      else{//no hay lugar y se debe pedir una nueva página
           b=new(page);
           if(r<=n)
            {
            for(int i=0;i<n;i++)
             b->e[i]=a->e[i+n];
             if(r==n)
             {
              v=a->e[n];
              b->e[0]=u;
             }
             else{
                  if (a->e[n-1].kyy < x)
                   v=u;
                  else
                  {
                   v=a->e[n-1];
                   if (a->e[0].kyy < x)
                     a->e[n-1]=u;
                   else
                    {
                    for(int i=n-1;i>=(r+1);i--)
                      a->e[i]=a->e[i-1];
                    a->e[r]=u; //+1
                    }
                  }
                 }
            }
```

```
            else{
              r=r-n;
              v=a->e[n];
              for(int i=0;i<r;i++)
                b->e[i]=a->e[i+n+1];
              b->e[r]=u;
              for(int i=r+1;i<=n;i++)
                b->e[i]=a->e[i+n];
               }
          a->m=n; // se especifican los valores de m para las páginas
          b->m=n;
          b->p0=v.p;// y los punteros correspondientes
          v.p=b;
          }
      }
      }
   }// fin inserción

 // Función que permite organizar las claves cuando las páginas no cumplen con
//alguna de las características del árbol B.

void vacio(puntero &c,puntero &a,int &s,int &h)
{puntero b;
  int i,k,mb,mc;
  /*a pagina subocupada c pagina antecesora*/
  mc=c->m;
  if(s>mc)
  {/*b pagina a la derecha de a*/
   s++;
   b=c->e[s].p;
   mb=b->m;
   k=(mb-n+1)/2;/*numero de item de la pag ady b*/
   a->e[n]=c->e[s];
   a->e[n].p=b->p0;
    if (k>0)
    {/*mover k items de b a a */
     for(i=0;i<k-1;i++)
     a->e[i+n]=b->e[i];
     c->e[s]=b->e[k];
     c->e[s].p=b;
     b->p0=b->e[k].p;
     mb=mb-k;
     for(i=0;i<mb;i++)
     b->e[i]=b->e[i+k];
     b->m=mb;
     a->m=n-1+k;
     h=0;
```

```
  }
  else{/*unir pag a y b*/
      for(i=0;i<n;i++)
      a->e[i+n]=b->e[i];
      b=NULL;
      for(i=s;i<mc-1;i++)
      {
        c->e[i]=c->e[i+1];
        c->e[i].p=c->e[i+1].p;
      }
      a->m=2*n;
      c->m=mc-1;
      h=0;
      }
   }
else{/*b pagina a la izquierda de a*/
    if (s==0)
      b=c->p0;
    else
      b=c->e[s-1].p;
    mb=b->m;
    k=(mb-n)/2;
    if (k>0)
    {/*mover k items de la pagina b a la a */
     for(i=n-1;i>=0;i--)
       a->e[i+k]=a->e[i];
     a->e[k]=c->e[s];
     a->e[k].p=a->p0;
     mb=mb-k;
     for(i=k-1;i>=0;i++)
       a->e[i]=b->e[i+mb];
     a->p0=b->e[mb].p;
     c->e[s]=b->e[mb];
     c->e[s].p=a;
     b->m=mb-1;
     a->m=n-1+k;
     h=0;
     }

      else{/*unir pagina a co b */
         b->e[mb]=c->e[s];
         b->e[mb].p=a->p0;
         if (c->e[s].kyy==0)
           {

            for(i=0;i<a->m;i++)
              b->e[i+mb]=a->e[i];
```

```
            b->m=b->m+a->m;
            }
          else{
            for(i=0;i<a->m;i++)
              b->e[i+mb+1]=a->e[i];
            mb=(b->m)+(a->m)+1;}
          if (mb<=2*n)
          { for(i=s ;i<c->m;i++)
              c->e[i]=c->e[i+1];
           c->m=mc-1;
           }
          else  {
            i=(mb/2);
            c->e[c->m-1]=b->e[i];
            b->m=mb-i-1;
            for (int j=i+1;j<b->m;j++)
              a->e[j-(i+1)]=b->e[j];

            a->m=mb-1;
            c->e[c->m-1].p=b;
            c->p0=a;}
          h=c->m <=n;
          }
    }
    }


 // fin vacío


void sup(puntero &p,puntero &a,int &h,int &r)
{puntero q;
 int i;
q=p->e[p->m-1].p;
if(q!=NULL)

   {  sup(q,p,h,r);
   if (h==1)
    vacio(p,q,p->m-1,h);
  }
else

  {
   a->e[r-1]=p->e[0];
   for (i=1;i<p->m;i++)
    p->e[i-1]=p->e[i];
   a->e[r-1].p=p;
```

```
   p->m--;
   if (p->m<n)
     h=1;
   }
}

 // Función que permite suprimir claves del árbol B.

void suprimir(int x,puntero &a,int &h)
{
 int l,i=0,r;
 puntero q;
 if(a==NULL)
 {
  printf("\nNo esta el elemento   ");
  h=0;
  }
 else
 {
  l=1;
  r=a->m;/*Busqueda Binaria en el array*/
  while (l<r)
  {
   i=(l+r)/2;
   if (a->e[i].kyy<=x)
    l=i+1;
   else
    r=i;
   }

  if((r==1) &&(a->e[r-1].kyy>x))
     q=a->p0;
  else
   if((r==1) &&(a->e[r-1].kyy<x))
     q=a->e[r-1].p;
   else
     q=a->e[r-1].p;
  i=r-1;
  if((r<=a->m)&&(a->e[i].kyy==x))/*se encontro en e[i]*/
   {
     if (q==NULL)/*pagina terminal*/
      {
       for (int j=i;j<a->m;j++)
         a->e[j]=a->e[j+1];
       a->m--;
       if (a->m < n)
         h=1;
```

```
      }
     else
     {
     if (a->m==1)
     {
       a->e[i].kyy=0;
       h=1;
       }
      else
        sup(q,a,h,r);
      if(h==1)
         vacio(a,q,r-1,h);
         printf("%d",q->m);
     }
     }
  else
  {
   suprimir(x,q,h);
   if(h==1)
      vacio(a,q,r-1,h);
   }
}
}//fin suprimir

// Función que permite mostrar el Arbol B

void mostrar(puntero p,int niv)
{int i;
if(p!=NULL)
{
 for(i=0;i<niv;i++)
   printf("\n  ");
 for(i=0;i<p->m;i++)
   printf("%d  ",p->e[i].kyy);
 getchar();
 mostrar(p->p0,niv+1);
 for(i=0;i<p->m;i++)
   mostrar(p->e[i].p,niv+1);
 }
 }// fin mostrar

//*** PRINCIPAL ***

void main (void)
{
 puntero raiz,q;
 item v;
```

```
int op,x,h;

raiz=NULL;

do
{
  clrscr();
  printf("\n\n Menú \n");
  printf("\n 1_ Insertar\n");
  printf("\n 2_ Suprimir\n");
  printf("\n 3_ Mostrar\n");
  printf("\n 4_ Salir\n");
  printf("\n \n");

printf("Ingrese opción  ");

scanf("%d",&op);
switch(op)
{
case 1:{
        printf("\n Ingrese clave a insertar(Finaliza con -1) ");
        scanf("%d",&x);
        while(x >=0)
        {
         insertar(x,raiz,h,v);
         if (h)
           {
            q=raiz;
            raiz=new(page);
            raiz->m=1;
            raiz->p0=q;
            raiz->e[0]=v;
            }
          printf("\n Ingrese clave a insertar (Finaliza con -1) ");
          scanf("%d",&x);
        }
        break;
        }
case 2:{
         printf("ingrese clave a suprimir (Finaliza con -1) ");
         scanf("%d",&x);
         while(x >=0)
           {
             suprimir(x,raiz,h);
             if (h)
               {
                 if (raiz->m == 0)
```

```
                {
                 q=raiz;
                 raiz=q->p0;
                 printf("%d %d",raiz->m,q->p0->m);
                 getchar();
                 }
               }
              mostrar(raiz,0);
              getchar();
             printf("ingrese clave a suprimir (Finaliza con -1) ");
             scanf("%d",&x);
            }
           break;
         }
 case 3:{
        mostrar(raiz,0);
        getchar();
         break;
        }
  }
}
while(op!=4);
}// fin principal
```