

Capítulo 6.

Árboles binarios de búsqueda.

En listas enlazadas de n componentes las operaciones generales de inserción, descarte y búsqueda son $O(n)$.

Como veremos, en árboles binarios de búsqueda (bst por binary search trees) con n nodos estas operaciones serán $O(\log_2 n)$ en promedio.

Un árbol presenta vínculos jerárquicos entre sus componentes y permite representar variadas situaciones de interés. Por ejemplo: la estructura de un directorio, la conectividad entre vértices de un grafo, representación de expresiones, el almacenamiento de las palabras reservadas, diccionarios, etc.

6.1. Definiciones.

Un árbol es una colección de cero o más **nodos** vinculados con una relación de jerarquía.

Un árbol con cero nodos se denomina árbol **vacío**.

Un árbol tiene un nodo especial, o punto de entrada a la estructura, denominado **raíz**.

La raíz puede tener cero o más nodos accesibles desde ella. El conjunto de esos nodos forman **subárboles** de la raíz, y son nodos **descendientes** de la raíz. La raíz es el **ancestro** de sus descendientes.

El nodo raíz no tiene ancestros.

Un **subárbol** es un nodo con todos sus descendientes.

Un nodo sin descendientes es una **hoja**. Una hoja no tiene nodos **hijos**.

Un **árbol** es un: árbol vacío o un nodo simple o
un nodo que tiene **árboles** descendientes.

La definición es **recursiva**. Se define un árbol en términos de árboles.

Una **trayectoria** del nodo n_i al nodo n_k , es una secuencia de nodos desde n_i hasta n_k , tal que n_i es el padre de n_{i+1} . Existe un solo **enlace** o vínculo entre un **padre** y sus hijos.

Largo de una trayectoria es el número de enlaces en la trayectoria. Una trayectoria de k nodos tiene largo $k-1$.

Alto de un nodo: largo de la trayectoria más larga de ese nodo a una hoja.

Profundidad de un nodo: es el largo de la trayectoria de la raíz a ese nodo.

Ejemplos de definiciones.

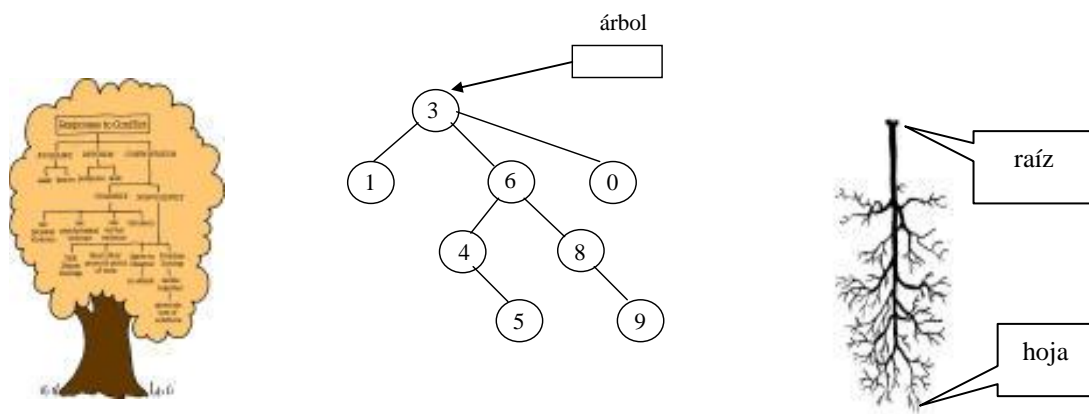


Figura 6.1. Árboles.

El nodo con valor 3 es la raíz. Los nodos con valores: 1, 5 y 9 son hojas. Los nodos con valores: 4, 6 y 8 son **nodos internos**. El nodo con valor 6 es hijo de 3 y padre de 8. El nodo con valor 4 es ancestro del nodo con valor 5. {8, 9} y {4, 5} son subárboles. El nodo con valor 1 es subárbol de la raíz. Los nodos con valores: 1, 6 y 0 son **hermanos**, por tener el mismo padre. El conjunto de nodos con valores: {3, 6, 4, 5} es una trayectoria, de largo 3. Alto del nodo con valor 6 es 2. La profundidad del nodo con valor 5 es 3.

Todos los nodos que están a igual profundidad están en el mismo **nivel**.
La profundidad del árbol es la profundidad de la hoja más profunda.

Se dice que un árbol es una estructura **ordenada**, ya que los hijos de un nodo se consideran ordenados de izquierda a derecha. También es una estructura **orientada**, ya que hay un camino único desde un nodo hacia sus descendientes.

6.2. Árbol binario.

Cada nodo puede tener: un hijo izquierdo, o un hijo derecho, o ambos o sin hijos. A lo más cada nodo puede tener dos hijos.

Un árbol binario está formado por un nodo raíz y un subárbol izquierdo *I* y un subárbol derecho *D*. Donde *I* y *D* son árboles binarios. Los subárboles se suelen representar gráficamente como triángulos.

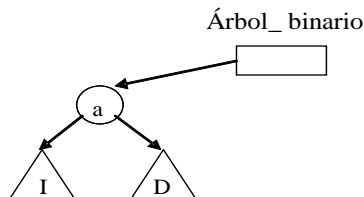


Figura 6.2. Árbol binario.

Definición de tipos de datos.

Usualmente el árbol se trata como conjunto dinámico, mediante la creación de sus nodos bajo demanda. Es decir, un nodo se crea con malloc, y contiene punteros a otros nodos de la estructura.

En caso de un árbol binario, debe disponerse al menos de dos punteros. Se ilustra un ejemplo con una clave entera, no se muestra espacio para la información periférica que puede estar asociada al nodo. En la implementación de algunas operaciones conviene disponer de un puntero al padre del nodo, que tampoco se declara en el molde del nodo.

```
typedef struct moldenode
{
    int clave;
    struct moldenode *left;
    struct moldenode *right;
} nodo, *pnodo;
```

6.3. Árbol binario de búsqueda.

Para cada nodo de un árbol binario de búsqueda debe cumplirse la propiedad:

*Las claves de los nodos del subárbol izquierdo deben ser menores que la clave de la raíz.
Las claves de los nodos del subárbol derecho deben ser mayores que la clave de la raíz.*

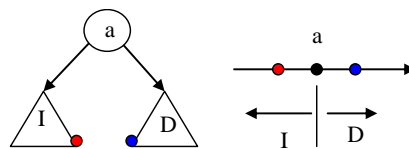


Figura 6.3. Árbol binario de búsqueda.

Esta definición no acepta elementos con claves duplicadas.

Se indican en el diagrama de la Figura 6.3, el descendiente del subárbol izquierdo con **mayor** clave y el descendiente del subárbol derecho con **menor** valor de clave; los cuales son el antecesor y sucesor de la raíz.

El siguiente árbol no es binario de búsqueda, ya que el nodo con clave 2, ubicado en el subárbol derecho de la raíz, tiene clave menor que ésta.

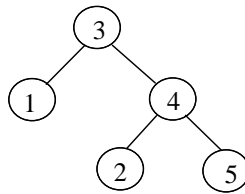


Figura 6.4. No es árbol binario de búsqueda.

Los siguientes son árboles de búsqueda ya que cumplen la propiedad anterior.

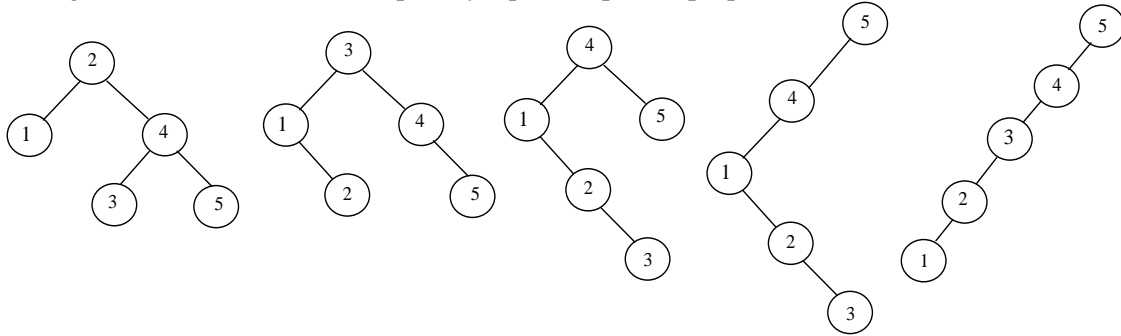


Figura 6.5. Varios árboles binarios de búsqueda con distinta forma.

La generación de estos árboles depende del orden en que se ingresen las claves en los nodos, a partir de un árbol vacío. El de la izquierda se generó insertando las claves en orden de llegada: 2, 1, 4, 3, 5 (o bien: 2, 4, 1, 5, 3). El de más a la derecha, se generó con la llegada en el orden: 5, 4, 3, 2, 1.

Los dos árboles de más a la izquierda, en la Figura 6.5, se denominan **balanceados**, ya que las diferencias en altura de los subárboles izquierdo y derecho, para todos los nodos, difieren a lo más en uno. Los tres a la derecha están desbalanceados. El último tiene la estructura de una lista, y es un árbol degenerado.

6.4. Cálculos de complejidad o altura en árboles.

6.4.1. Árbol completo.

Se denomina árbol completo, a aquél que tiene presentes todas las hojas en el menor nivel. La raíz es de nivel cero, los hijos de la raíz están en nivel 1; y así sucesivamente.

Deduciremos, de manera inductiva la altura de las hojas en función del número de nodos.

El caso más simple de un árbol completo tiene tres nodos, un nivel y altura dos. Hemos modificado levemente la definición de altura, como el número de nodos que deben ser revisados desde la raíz a las hojas, ya que la complejidad de los algoritmos dependerá de esta variable.

Árbol de nivel 1.
 Nodos = $3 = 2^2 - 1$

Altura = 2



Figura 6.6. Árbol completo de nivel 1.

Árbol de nivel 2.

Nodos = $7 = 2^3 - 1$

Altura = 3

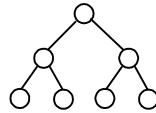


Figura 6.7. Árbol completo de nivel 2.

Árbol de nivel 3.

Nodos = $15 = 2^4 - 1$

Altura = 4

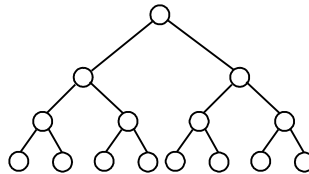


Figura 6.8. Árbol completo de nivel 3.

Se deduce para un árbol de m niveles:

Árbol de nivel m .

Nodos = $n = 2^A - 1$

Altura = $A = m + 1$

Hojas = 2^m

Nodos internos = $n - \text{Hojas}$

De la expresión para el número de nodos, puede despejarse A , se logra:

$$A = \log_2(n+1) = O(\log n)$$

Resultado que es simple de interpretar, ya que cada vez que se sigue una trayectoria por un determinado subárbol, se descarta la mitad de los nodos. Es decir, se rige por la relación de recurrencia: $T(n) = T(n/2) + c$, con solución logarítmica. Esta propiedad le otorga, a la estructura de árbol binario de búsqueda, grandes ventajas para implementar conjuntos dinámicos, en relación a las listas, que permiten elaborar algoritmos de complejidad $O(n)$.

La demostración por inducción matemática completa, del resultado anterior, es sencilla de deducir a partir del caso m -avo.

Puede demostrarse por inducción completa, el siguiente teorema:

Teorema: Un árbol perfectamente balanceado que tiene n nodos internos tiene $(n+1)$ hojas. El que se demostrará en 6.4.4.

También se denominan árboles perfectamente balanceados, en éstos todas las hojas tienen igual profundidad.

6.4.2 Árboles incompletos con un nivel de desbalance.

Se ilustran los tres casos de árboles, de nivel dos, con un nivel de desbalance, para $n=4, 5$ y 6 . Un árbol con 3 nodos es completo en caso de aceptarse sólo un nivel de desbalance. Lo mismo puede decirse de un árbol con 7 nodos y que tenga un nivel de desbalance.

Árboles de nivel 2.

Nodos de 4 a 6. De 2^{3-1} hasta 2^3-2 .

Altura = 3 en peor caso.

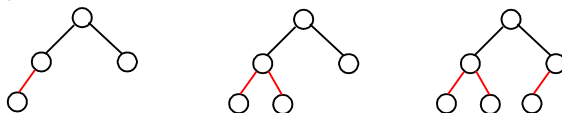


Figura 6.9. Árboles incompletos de nivel 2.

Árboles de nivel 3.

Nodos de 8 a 14. De 2^{4-1} hasta 2^4-2

Altura = 4 en peor caso.

Árboles de m niveles.

Nodos de 2^{A-1} hasta 2^A-2 .

Altura A .

La inecuación:

$2^{A-1} \leq n \leq 2^A - 2$ tiene como solución: $A \leq (1 + \log_2(n))$ para la primera desigualdad y $A \geq \log_2(n+2)$ para la segunda.

Se pueden encontrar **constantes** que acoten, por arriba y por abajo a ambas funciones:

$$1 * \log_2 n \leq \log_2(n+2) \leq A \leq 1 + \log_2 n \leq 1.3 * \log_2 n \text{ para } n > 10,079.$$

> **solve** (1+ (ln (n) / ln (2)) < 1.3*ln (n) / ln (2)) ;
 RealRange(Open(10.07936840), ∞)

Entonces, se tiene:

$$A = \mathcal{O}(\log n)$$

Veremos más adelante la estructura denominada heap, la cual puede interpretarse como un árbol perfectamente balanceado excepto que en el mayor nivel no está completo. Si no está completo, se van agregando nodos de izquierda a derecha.

El peor caso para la altura, se tiene con un árbol degenerado en una lista, en el cual la altura resulta $O(n)$.

6.4.3. Árboles contruidos en forma aleatoria.

Para n nodos, con claves: 1, 2, 3, ..., n , se pueden construir $n!$ árboles. Ya que existen $n!$ permutaciones de n elementos. El orden de los elementos de la permutación, es el orden en que se ingresan las claves a partir de un árbol vacío.

Lo que se desea conocer es la altura A_n , definida como la altura promedio de las búsquedas de las n claves y promediadas sobre los $n!$ árboles que se generan a partir de las $n!$ permutaciones que se pueden generar con la n claves diferentes.

Si el orden de llegada de las claves que se insertan al árbol se genera en forma aleatoria, la probabilidad de que la primera clave, que es la raíz, tenga valor i es $1/n$. Esto en caso de que todas las claves sean igualmente probables.

Se ilustra esa situación en el siguiente diagrama.

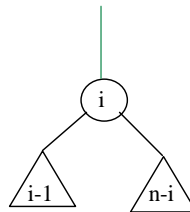


Figura 6.10. Raíz con valor i .

Se considera que el subárbol izquierdo contiene $(i-1)$ nodos; por lo tanto el subárbol derecho contiene $(n-i)$ nodos. Esto debido a la propiedad de árbol de búsqueda, en el subárbol izquierdo debe tenerse los $(i-1)$ valores menores que la clave i de la raíz.

Denominamos A_{n-i} al largo promedio de las trayectorias en el subárbol derecho, y A_{i-1} al largo promedio de las trayectorias del subárbol que contiene $(i-1)$ nodos. Esto asume que el resto de las permutaciones de las $(n-1)$ claves restantes son igualmente probables.

Es decir:

Los $i-1$ nodos del subárbol izquierdo tienen largo de trayectorias promedio igual a $A_{i-1} + 1$.

Los $n-i$ nodos del subárbol derecho tienen largo de trayectorias promedio igual a $A_{n-i} + 1$.

La raíz tiene un largo de trayectoria igual a 1.

El promedio ponderado, de los largos de trayectoria para buscar los n nodos, para el árbol con clave i en la raíz es:

$$A_n(i) = \frac{(i-1)(A_{i-1} + 1) + 1 + (n-i)(A_{n-i} + 1)}{n}$$

Y promediando considerando que el nodo i , en la raíz, puede ser: 1, 2, 3, ..., n

$$A_n = \frac{1}{n} \sum_{i=1}^{i=n} A_n(i)$$

Reemplazando la expresión para $A_n(i)$ se tiene:

$$A_n = \frac{1}{n} \sum_{i=1}^{i=n} \left(\frac{i-1}{n} (A_{i-1} + 1) + 1 \frac{1}{n} + \frac{n-i}{n} (A_{n-i} + 1) \right)$$

efectuando factorizaciones:

$$A_n = \frac{1}{n^2} \sum_{i=1}^{i=n} ((i-1)A_{i-1} + n + (n-i)A_{n-i})$$

Sumando el término que no depende de i , resulta:

$$A_n = 1 + \frac{1}{n^2} \sum_{i=1}^{i=n} ((i-1)A_{i-1} + (n-i)A_{n-i})$$

El último paso considera que n es una constante, ya que el índice de la suma es i .

En la última sumatoria, los dos factores a sumar, dan origen a iguales sumandos. En ambas sumatorias se obtiene: $0 \cdot A_0 + 1 \cdot A_1 + \dots + (n-1) \cdot A_{n-1}$

Resulta entonces:

$$A_n = 1 + \frac{2}{n^2} \sum_{i=1}^{i=n} (i-1)A_{i-1}$$

Efectuando un cambio de variables, se obtiene:

$$A_n = 1 + \frac{2}{n^2} \sum_{i=1}^{i=n-1} iA_i$$

Se tiene una relación de recurrencia de orden $(n-1)$. Se requieren conocer $(n-1)$ términos para calcular el n -avo.

La cual puede ser transformada en una relación de recurrencia de primer orden, por el siguiente procedimiento:

a) Se reemplaza n por $(n-1)$, en la relación anterior, con lo que se obtiene:

$$A_{n-1} = 1 + \frac{2}{(n-1)^2} \sum_{i=1}^{i=n-2} iA_i$$

b) Se extrae el término $(n-1)$ de la sumatoria, de la relación de recurrencia de orden $(n-1)$, con lo que se obtiene:

$$A_n = 1 + \frac{2}{n^2} (n-1)A_{n-1} + \frac{2}{n^2} \sum_{i=1}^{i=n-2} iA_i$$

Si se despeja la sumatoria de la relación obtenida en a) y se reemplaza en la sumatoria de la relación obtenida en b), quedará una relación de recurrencia de primer orden; es decir, A_n en términos de A_{n-1} .

$$A_n = 1 + \frac{2}{n^2} (n-1)A_{n-1} + \frac{2}{n^2} \left(\frac{(A_{n-1}-1)(n-1)^2}{2} \right)$$

Arreglando:

$$A_n = \frac{1}{n^2} ((n^2-1)A_{n-1} + 2n-1)$$

Empleando maple, la solución de esta recurrencia, para $n > 1$ es:

$$A_n = \frac{2(n+1)\Psi(n+1) + 2\gamma - 3n + 2\gamma n}{n}$$

Donde $\Psi(n)$ es la función digama.

La relación también puede resolverse mediante la función armónica.

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

Dando como resultado:

$$A_n = 2 \frac{n+1}{n} H_n - 3$$

El procedimiento no es trivial. Se da el resultado sin desarrollo.

Para verificar que es solución puede reemplazarse ésta en la ecuación de recurrencia, con lo que se obtiene:

$$2 \frac{n+1}{n} H_n - 3 = \frac{1}{n^2} ((n^2-1)(2 \frac{n}{n-1} H_{n-1} - 3) + 2n-1)$$

La cual es una identidad, ya que se obtiene, luego de un trabajo algebraico:

$$H_n - H_{n-1} = \frac{1}{n}$$

Retomando el cálculo, la función armónica puede ser aproximada por la función gama (constante de Euler, $\gamma=0,577\dots$), mediante:

$$H_n = \gamma + \ln(n) + \frac{1}{12n^2} + \dots$$

Si n es grande:

$$H_n \approx \ln(n)$$

$$A_n \approx 2H_n - 3$$

Lo cual permite obtener, finalmente:

$$A_n \approx 2 \ln(n) = \Theta(\log n)$$

Resultado que garantiza que **en promedio**, el largo promedio de cualquier trayectoria en un árbol generado aleatoriamente es de complejidad logarítmica.

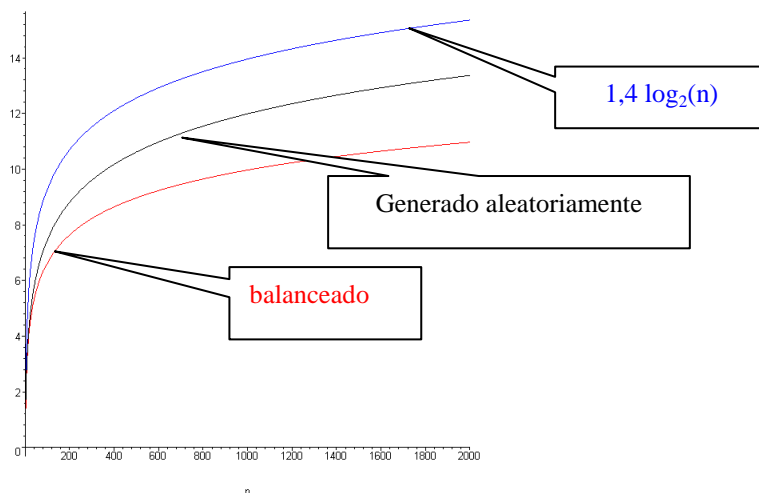


Figura 6.11. Altura de árbol generado aleatoriamente.

La gráfica muestra que para árboles de tipo 1000 nodos, deben recorrerse cerca de nueve nodos desde la raíz hasta las hojas (peor caso), si está balanceado. El largo promedio de los recorridos es 12 en un árbol generado aleatoriamente, y 1000 en peor caso.

Otra forma de expresar el resultado, es considerando el alargue de la trayectoria en términos del largo para un árbol perfectamente balanceado. Éste era $A = \log_2(n+1)$, entonces para n grande:

$$\frac{A_n}{A} \approx \frac{2 \ln(n)}{\log_2(n)} \approx 2 \ln(2) \approx 1,386$$

El resultado establece que el promedio de alargue es de un 39%. Pero esta cota es para el caso promedio, además si n es relativamente grande el alargue es menor. Se muestra una gráfica del alargue, en función de n . Para árboles con menos de 2000 nodos el alargue es menor que un 22%.

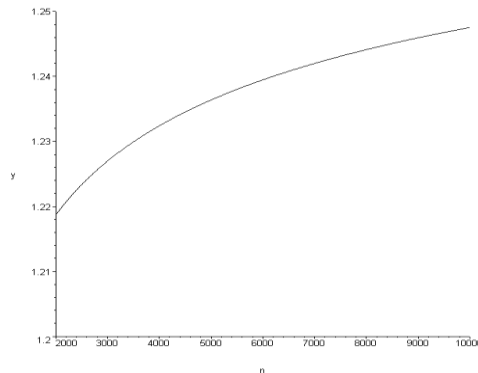


Figura 6.12. Alargue de altura de árbol generado aleatoriamente.

Si dada la naturaleza de la aplicación se desea tener un **peor caso** con complejidad logarítmica, debemos garantizar que el árbol se mantendrá *lo más balanceado posible*. Existen una serie de algoritmos que logran este objetivo: Árboles: coloreados, 2-3, AVL.

6.4.4. Número de comparaciones promedio en un árbol binario de búsqueda.

6.4.4.1. Árbol binario externo

Se consideran las hojas como nodos externos.

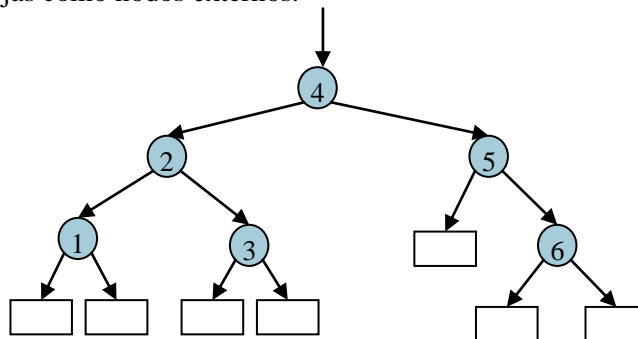


Figura 6.13. Nodos internos y externos.

Teorema

Para un árbol binario de búsqueda de n nodos internos el correspondiente árbol binario externo tiene: $(2n + 1)$ nodos. Hay $(n+1)$ nodos externos u hojas.

Demostración.

Por inducción completa, cuyo método resumimos a continuación:

Se tiene un conjunto y una propiedad que puede expresarse mediante una fórmula $P(n)$, se verifica:

1° Que la propiedad se cumpla para el primer elemento o los primeros elementos del conjunto.

2° Que dado $P(n)$ se cumpla que: $P(n) \Rightarrow P(n+1)$.

La conclusión es: Todos los elementos del conjunto cumplen esa propiedad. Esta forma de razonamiento también se denomina por recurrencia.

Sea n_i el número de nodos internos, y n_e el número de nodos externos.

Para $n_i = 1$ se tiene: $n_e = 2$ y se cumple que: $n_e = n_i + 1$

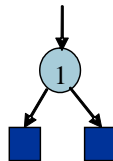


Figura 6.14. $P(1): n_e = n_i + 1$

Para $n_i = 2$ se tiene: $n_e = 3$ y se cumple que: $n_e = n_i + 1$

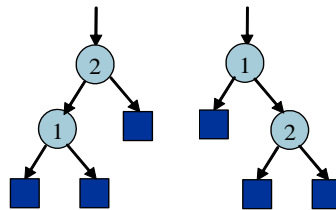


Figura 6.15. $P(2): n_e = n_i + 1$

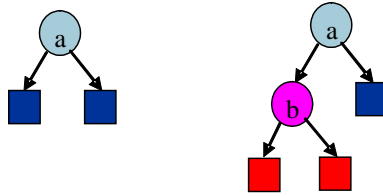
En este caso se pueden formar dos árboles. Si a partir de un árbol vacío, llegan las claves en orden 2,1 ó 1, 2, se tienen los diagramas anteriores.

Supongamos que para $n_i = n$ se tiene que: $n_e = n+1$ entonces:

Para $n_i = n+1$ podemos agregar el nodo interno de dos formas: Una reemplazando a un nodo externo, o bien intercalándolo entre dos nodos internos.

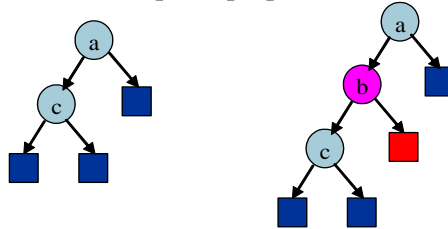
En el primer caso, disminuye en uno el número de nodos externos, pero luego se agregan dos nodos externos, se tendrá:

$n_e' = (n+1) - 1 + 2 = (n+1) + 1 = n_i + 1$ y se cumple la propiedad.

Figura 6.16. Primer caso de $P(n+1)$: $n_e = n_i + 1$

En el segundo caso:

$n_e'' = (n+1) + 1 = n_i + 1$ también se cumple la propiedad.

Figura 6.17. Segundo caso de $P(n)$: $n_e = n_i + 1$

6.4.4.2. Largos de trayectorias interna y externa.

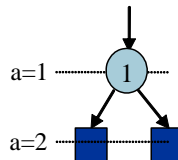
Teorema.

Sea el **largo de la trayectoria interna** $I(n)$, la suma de las alturas de todos los nodos internos; y sea el **largo de la trayectoria externa** $E(n)$, la suma de las alturas de todos los nodos externos.

Entonces: $E(n) = I(n) + (2n + 1)$

Demostración por inducción.

La propiedad $P(n)$, se cumple para el caso base, con $n = 1$.

Figura 6.18. $P(1)$: $E(n) = I(n) + (2n + 1)$

$$I(1) = 1, E(1) = 2 + 2 = 4$$

$$E(1) = I(1) + (2 \cdot 1 + 1) = 4$$

Para $n=2$, en ambos diagramas:

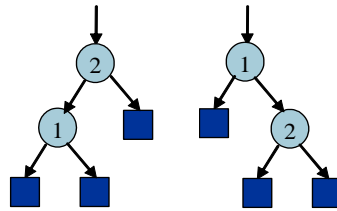


Figura 6.19. $P(2)$: $E(n) = I(n) + (2n + 1)$

$$I(2) = 1 + 2 = 3$$

$$E(2) = 3 + 3 + 2 = 8$$

$$E(2) = I(2) + (2 \cdot 2 + 1) = 8 \quad \text{Entonces } P(2) \text{ se cumple.}$$

Nótese que $I(n)$ es el número de comparaciones de claves que deben realizarse para encontrar todos los nodos. También se tiene que $E(n) - (n+1)$ es el número de comparaciones que deben realizarse para encontrar las hojas o nodos externos; se cuentan sólo las comparaciones de claves.

La hipótesis inductiva, está basada en asumir $P(n)$ verdadero y demostrar que:

$$P(n) \Rightarrow P(n+1)$$

$$\text{Se cumple } E(n) = I(n) + (2n+1)$$

Se reemplaza cualquiera de los nodos externos, digamos uno ubicado a altura d de la raíz, por un nodo interno y sus dos hijos externos:

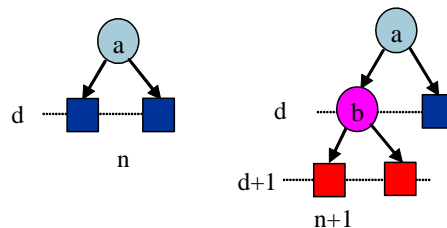


Figura 6.20. Primer caso de $P(n)$: $E(n) = I(n) + (2n + 1)$

Los nuevos largos, para el árbol con $(n+1)$ nodos internos quedan:

$$I(n+1) = I(n) + d \quad \text{ya que el nodo } b \text{ tiene altura } d \text{ respecto de la raíz.}$$

$$E(n+1)' = E(n) - d + 2(d+1) \quad \text{ya que pierde un nodo con altura } d, \text{ pero adquiere dos a distancia } (d+1).$$

Entonces reemplazando $P(n)$ en el largo externo queda:

$$E(n+1)' = I(n) + (2n+1) - d + 2(d+1) = I(n) + d + 2n + 1 + 2$$

Y empleando la relación para $I(n+1)$, resulta:

$$E(n+1)' = I(n+1) + 2(n+1) + 1 \quad \text{que demuestra que } P(n+1) \text{ se cumple.}$$

Falta analizar la situación en la que se intercala un nodo a altura d de la raíz, entre dos nodos internos.

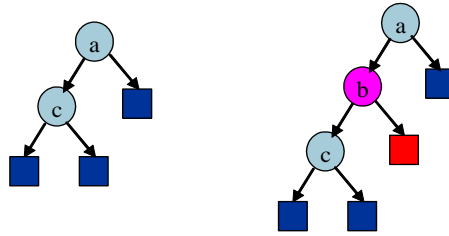


Figura 6.21. Segundo caso de $P(n)$: $E(n) = I(n) + (2n + 1)$

Los nuevos largos en función de la situación para n nodos internos, se pueden plantear:

$I(n+1) = I(n) + d + k$. Con k igual al número de nodos internos bajo el insertado.

$E(n+1)'' = E(n) + (d+1) + j$. Con j igual al número de nodos externos bajo el insertado.

Se tiene que $j = k+1$, ya que para un árbol de k nodos internos se tienen $(k+1)$ nodos externos.

Reemplazando j y el valor de $E(n)$ mediante la propiedad $P(n)$ se tiene:

$$E(n+1)'' = (I(n) + 2n+1) + (d+1) + k+1 = I(n) + d + k + 2n+3.$$

Reemplazando el valor de $I(n+1)$ se obtiene:

$$E(n+1)'' = I(n+1) + 2n+3 = I(n+1) + 2(n+1) + 1 \text{ que muestra que } P(n+1) \text{ se cumple.}$$

6.4.4.3. Búsquedas exitosas y no exitosas.

Teorema

- $S(n) \leq 1.39 \log_2(n)$ $\Rightarrow S(n) = \Theta(\log_2 n)$
- $U(n) \leq 1.39 \log_2(n+1)$ $\Rightarrow U(n) = \Theta(\log_2 n)$

Sea $S(n)$ el número de comparaciones en una búsqueda exitosa en un árbol binario de búsqueda de n nodos internos, construido aleatoriamente.

Para un árbol dado el número esperado de comparaciones para encontrar todas las claves es:

$$S(n) = I(n)/n$$

$$S(1) = 1$$

$$S(2) = 3/2$$

$$I(1) = 1, I(2) = 3$$

Sea $U(n)$ el número de comparaciones en una búsqueda no exitosa en un árbol binario de búsqueda de n nodos internos, construido aleatoriamente.

Para un árbol dado, $U(n)$ es el número esperado de comparaciones para encontrar todas las hojas es:

$$U(n) = \frac{E(n) - (n+1)}{n+1}$$

Una búsqueda no exitosa termina en uno de los $(n+1)$ nodos externos.

$$U(1) = 2$$

$$U(2) = 5/3$$

$$E(1) = 4, E(2) = 8$$

$$U(0) = 0 \text{ ya que } E(0) = 1 \text{ y } n=0$$

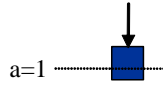


Figura 6.22. Evaluación de $U(0)$.

Relación entre $S(n)$ y $U(n)$.

Puede expresarse el número promedio de comparaciones en una búsqueda exitosa S , en términos del número de comparaciones no exitosas U , eliminando las variables E e I , empleando la relación entre éstas, según:

$$S(n) = \frac{I(n)}{n} = \frac{E(n) - (2n+1)}{n} = \frac{(n+1)U(n) + (n+1) - (2n+1)}{n}$$

Resulta:

$$S(n) = \left(1 + \frac{1}{n}\right)U(n) - 1$$

La inserción de un nuevo valor de clave en un árbol de búsqueda implica un recorrido descendente, desde la raíz hasta una hoja, es decir una búsqueda no exitosa. Esto debido a que no se aceptan claves repetidas.

Entonces: Se requiere una comparación adicional para encontrar una clave en una búsqueda exitosa que lo que se requiere para insertar la clave en una búsqueda no exitosa que precede a su inserción.

Si la búsqueda encuentra al nodo en la raíz se requieren $U(0) + 1$ comparaciones, ya que fue el primero que se insertó.

Si el buscado fue el k -ésimo insertado, se requieren $U(k-1) + 1$ comparaciones para encontrarlo.

Como el nodo buscado puede haber sido insertado en un árbol vacío o en un árbol que ya tuviera 1, 2, .. o $(n-1)$ nodos se tiene, el siguiente promedio para las comparaciones:

$$S(n) = \frac{(U(0) + 1) + (U(1) + 1) + \dots + (U(n-1) + 1)}{n}$$

$$S(n) = \frac{1}{n} \sum_{k=1}^{k=n} (U(k-1) + 1)$$

$$S(n) = 1 + \frac{1}{n} \sum_{k=1}^{k=n} U(k-1) = 1 + \frac{1}{n} \sum_{k=0}^{k=n-1} U(k)$$

Relación de recurrencia para $U(n)$.

Igualando las dos expresiones para $S(n)$, se obtiene:

$$(1 + \frac{1}{n})U(n) - 1 = 1 + \frac{1}{n} \sum_{k=0}^{n-1} U(k)$$

Despejando:

$$(n+1)U(n) = 2n + \sum_{k=0}^{n-1} U(k)$$

Si en la anterior se reemplaza, n por $n-1$, se obtiene:

$$nU(n-1) = 2(n-1) + \sum_{k=0}^{n-2} U(k)$$

Restando la segunda a la primera, se obtiene:

$$(n+1)U(n) - nU(n-1) = 2 + U(n-1)$$

Despejando $U(n)$, se logra, la relación de recurrencia de primer orden, con $U(0) = 0$:

$$U(n) = U(n-1) + \frac{2}{n+1}$$

Que puede resolverse, mediante el siguiente método:

Reemplazando n por $n-1, n-2, \dots, 2, 1$ se tienen:

$$\begin{aligned} U(n-1) &= U(n-2) + \frac{2}{n} \\ U(n-2) &= U(n-3) + \frac{2}{n-1} \\ &\dots \\ U(1) &= U(0) + 1 \end{aligned}$$

Las que reemplazadas en la expresión para $U(n)$ permiten expresar:

$$U(n) = U(0) + \frac{2}{2} + \frac{2}{3} + \dots + \frac{2}{n-2} + \frac{2}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

Sumando y restando uno y factorizando por dos, se tiene:

$$U(n) = U(0) + 2(-1 + 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-1} + \frac{1}{n} + \frac{1}{n+1})$$

Reemplazando mediante $H(n)$, la función armónica, cuya suma es conocida:

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n-1} + \frac{1}{n}$$

$$U(n) = U(0) + 2(H(n+1) - 1)$$

Finalmente, con $U(0)=0$:

$$U(n) = 2(H(n+1) - 1)$$

La que puede aproximarse por:

$$U(n) \approx 2 \ln(n+1)$$

Relación de recurrencia para $S(n)$.

Reemplazando $U(n)$ en la expresión para $S(n)$:

$$S(n) = (1 + \frac{1}{n})2(H(n+1) - 1) - 1$$

$$S(n) = (1 + \frac{1}{n})2(H(n) + \frac{1}{n+1} - 1) - 1$$

Resultando:

$$S(n) = 2(1 + \frac{1}{n})H(n) - 3$$

Aproximando:

$$S(n) \approx 2H(n)$$

$$S(n) \approx 2 \ln(n) = 2 \ln(2) \log_2(n) = 1,39 \log_2(n) = \Theta(\log_2(n))$$

Finalmente:

$$S(n) = \Theta(\log_2(n))$$

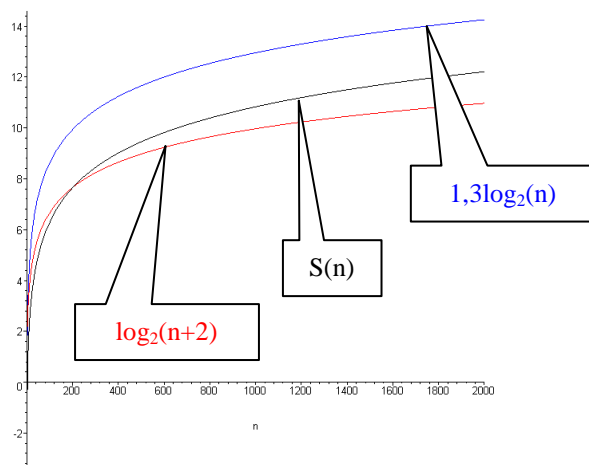


Figura 6.23. $S(n)$ es $\Theta(\log_2(n))$.

Teorema

Después de insertar n claves en un árbol binario de búsqueda T , inicialmente vacío, la altura promedio de T es $\Theta(\log_2 n)$.

6.5. Recorridos en árboles.

Se denominan recorridos a la forma en que son visitados todos los nodos de un árbol. Existen tres modos de recorrido, con las siguientes definiciones recursivas.

6.5.1. En orden:

- Se visita el subárbol izquierdo en orden;
- Se visita la raíz;
- Se visita el subárbol derecho en orden;

6.5.2. Pre orden:

- Se visita la raíz;
- Se visita el subárbol izquierdo en preorden;
- Se visita el subárbol derecho en preorden;

6.5.3. Post orden:

- Se visita el subárbol izquierdo en postorden;
- Se visita el subárbol derecho en postorden;
- Se visita la raíz;

6.5.4. Ejemplo de recorridos.

Recorrer el árbol formado con el siguiente conjunto de claves: { $n_0, n_1, n_2, n_3, n_4, n_5$ }, y cuyo diagrama se ilustra en la figura 6.24.

En orden:

{ n_1, n_3, n_4 }, n_0 , { n_2, n_5 }
 $n_3, n_1, n_4, n_0, \{n_2, n_5\}$
 $n_3, n_1, n_4, n_0, n_2, n_5$

Pre orden

$n_0, \{n_1, n_3, n_4\}, \{n_2, n_5\}$
 $n_0, n_1, n_3, n_4, \{n_2, n_5\}$
 $n_0, n_1, n_3, n_4, n_2, n_5$

Post orden

{ n_1, n_3, n_4 }, { n_2, n_5 }, n_0
 $n_3, n_4, n_1, \{n_2, n_5\}, n_0$
 $n_3, n_4, n_1, n_5, n_2, n_0$

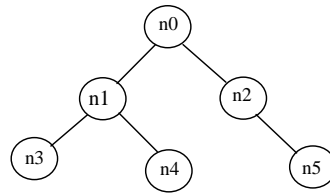


Figura 6.24. Árbol con claves {n0, n1, n2, n3, n4, n5}.

6.5.5. Árboles de expresiones.

Un árbol puede emplearse para representar la vinculación entre operadores y operandos.

Notación in situ, corresponde a recorrido en orden: $(a * b) / (c + d)$

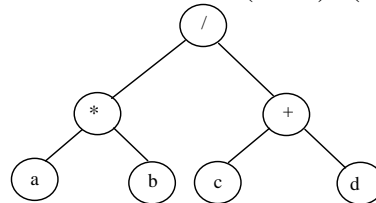


Figura 6.25. Árbol que representa a: $(a * b) / (c + d)$

Notación polaca inversa, corresponde a recorrido en post orden: $a b * c d + /$

Tarea: Encontrar expresión in situ para la polaca inversa: $a b c / + d e f * - *$

6.5.6. Árboles de derivación.

Los compiladores emplean árboles de derivación para verificar la construcción de sentencias sintácticamente correctas.

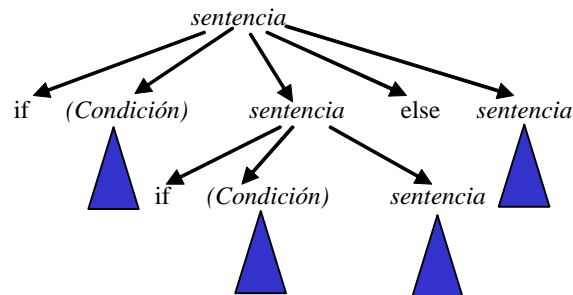


Figura 6.26. Árbol de derivación.

6.6. Operaciones en árboles binarios.

Al estudiar las operaciones más frecuentes en árboles y al analizar sus complejidades se podrá notar el poder de esta estructura de datos para implementar los conceptos de buscar y

seleccionar. Los árboles son necesarios para dar soporte a esas operaciones básicas de conjuntos dinámicos.

6.6.1. Operaciones básicas

6.6.1.1. Crear árbol vacío.

La variable **arbol** debe ser un puntero en zona estática o en el stack. A través de este puntero se tiene acceso a las componentes.

```
pnode arbol=NULL;
```

6.6.1.2. Crea nodo inicializado con un valor de clave.

```
pnode CreaNodo(int valor)
{
    pnode pi=NULL;

    if ( (pi= (pnode) malloc(sizeof(nodo))) ==NULL) exit(1);
    else
    {
        pi->clave=valor; pi->left=NULL; pi->right=NULL;
    }
    return(pi);
}
```

6.6.1.3. Ejemplo de uso.

```
arbol = CreaNodo(5); //si el árbol estaba vacío, crea raíz del árbol con clave igual a 5.
```

6.6.2. Operaciones de recorrido

6.6.2.1. Mostrar en orden

Diferentes árboles de búsqueda que almacenen las mismas claves son mostrados en el mismo orden al efectuar este recorrido.

Un árbol de búsqueda preserva el ordenamiento de sus componentes independiente de su forma.

Una técnica de importancia para efectuar diseños recursivos consiste:

- En conceptualizar lo que realiza la función y **asumir que ésta realiza su objetivo.**
- En establecer las **condiciones de término** de las reinvocaciones.

En el diseño de mostrar en orden, es simple establecer que el término se logra cuando no se encuentran los hijos de las hojas.

```

void RecorraEnOrden(pnodo p)
{
    if (p!= NULL) //si llegó a las hojas o es un árbol vacío.
    {
        RecorraEnOrden(p->left); //primero recorre el subárbol izquierdo.
        printf ("%d \n", p->clave); //terminado lo anterior, imprime el nodo apuntado por p
        RecorraEnOrden(p->right);
    }
}

```

La complejidad de un recorrido que debe visitar n nodos puede intuirse que será $\Theta(n)$.

Si se tiene un árbol de n nodos, y si se asume arbitrariamente que el subárbol izquierdo tiene k nodos, se puede plantear que la complejidad temporal del recorrido es:

$$T(n) = T(k) + \Theta(1) + T(n-k-1)$$

Considerando de costo constante la impresión, y la evaluación del condicional.

Para simplificar el cálculo podemos asumir un árbol balanceado.

$$T(n) = T(n/2) + \Theta(1) + T(n/2 - 1)$$

Y para grandes valores de n , podemos simplificar aún más:

$$T(n) = 2 * T(n/2) \text{ que tiene por solución: } T(n) = n = \Theta(n)$$

Otro cálculo es considerar el peor caso para el subárbol derecho:

$$T(n) = T(1) + \Theta(1) + T(n-2)$$

La que se puede estudiar como

$$T(n) = T(n-2) + 2 \text{ con } T(1)=1, T(2)=1 \text{ que tiene por solución}$$

$T(n) = n - (1/2)(1 + (-1)^n)$. El segundo término toma valor cero para n par, y menos uno para n impar. Puede despreciarse para grandes valores de n , resultando: $T(n) = \Theta(n)$

Si se desea mostrar el nivel, de cada nodo, basta una pequeña modificación, agregando un argumento nivel; cada vez que se desciende un nivel se incrementa éste en **uno**. Lo cual es un ejemplo de las posibilidades que tienen los algoritmos recursivos.

```

void inorder(pnodo t, int nivel)
{
    if (t != NULL) {
        inorder(t->left, nivel+1);
        printf ("%d %d \n", t->clave, nivel);
        inorder(t->right, nivel +1);
    }
}

```

Ejemplo de uso:

`inorder(arbol, 0);` //Imprime considerando la raíz de nivel cero.

6.6.2.2. Mostrar en post-orden

```
void prtpostorder(pnodo p)
{
    if (p!= NULL)
    {
        prtpostorder(p->left);
        prtpostorder(p->right);
        printf ("%d \n", p->clave);
    }
}
```

6.6.2.3. Mostrar en pre-orden

```
void prtpreorder(pnodo p)
{
    if (p!= NULL)
    {
        printf ("%d \n", p->clave);
        prtpreorder(p->left);
        prtpreorder(p->right);
    }
}
```

6.6.3. Operaciones de consulta.

Se suele pasar como argumento la raíz del árbol.

6.6.3.1. Seleccionar el nodo con valor mínimo de clave.

Considerando la propiedad de orden del árbol de búsqueda, debe descenderse a partir de la raíz por el subárbol izquierdo hasta encontrar un nodo con hijo izquierdo nulo, el cual contiene el valor mínimo de clave. Debe considerarse que el árbol puede estar vacío.

La implementación iterativa de esta operación es sencilla de implementar. Se retorna puntero al nodo con valor mínimo de clave, y NULL si el árbol está vacío.

```
pnodo BuscarMinimoIterativo(pnodo t) {
    while ( t != NULL){
        if ( t->left == NULL ) return (t); //apunta al mínimo.
        else t=t->left; //desciende
    }
    return (t); /* NULL si árbol vacío*/
}
```

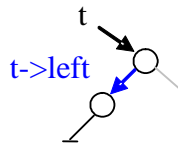


Figura 6.27. Variables en BuscarMinimoIterativo.

```

/* Algoritmo recursivo. Descender siempre por la izquierda */
pnodo BuscaMinimo(pnodo t) {
    if (t == NULL) return(NULL); //si árbol vacío retorna NULL
    else
        // Si no es vacío
        if (t->left == NULL) return(t); // Si no tiene hijo izquierdo: lo encontró.
        else return( BuscaMinimo (t->left) ); //busca en subárbol izquierdo.
}

```

Otra forma de concebir la función, es plantear primero las condiciones de término, y luego seguir *tratando de hacer lo que la función realiza, pero acercándose a la solución*.

```

pnodo BuscaMinimo(pnodo t)
{
    if (t == NULL) return(NULL); //si árbol vacío retorna NULL
    if (t->left == NULL) return(t); // Si no tiene hijo izquierdo: lo encontró.
    return( BuscaMinimo (t->left) ); //busca en subárbol izquierdo.
}

```

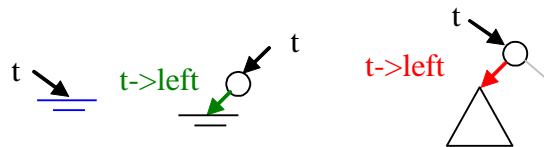


Figura 6.28. Condiciones en BuscaMinimo

La complejidad del algoritmo es $O(a)$, donde a es la altura del árbol. Ésta en promedio es $O(\log_2(n))$, y en peor caso $O(n)$.

6.6.3.2. Seleccionar el nodo con valor máximo de clave.

Algoritmo: Descender a partir de la raíz por el subárbol derecho hasta encontrar un nodo con hijo derecho nulo, el cual contiene el valor máximo de clave. Debe considerarse que el árbol puede estar vacío.


```

pnodo BuscarMaximoIterativo(pnodo t) {
    while ( t != NULL)
    {
        if ( t->right == NULL ) return (t); //apunta al máximo.
        else t=t->right; //desciende
    }
    return (t); /* NULL Si árbol vacío*/
}

/* Recursivo */
pnodo BuscaMaximo(pnodo t)
{
    if (t == NULL) return(NULL); //si árbol vacío retorna NULL
    if (t->right == NULL) return(t); // Si no tiene hijo derecho: lo encontró.
    return( BuscaMaximo(t->right) ); //sigue buscando en subárbol derecho.
}

```

Si en las funciones obtenidas en 6.6.3.2. se cambian left por right y viceversa se obtienen las funciones anteriores. Esta es una importante propiedad de los árboles binarios de búsqueda.

6.6.3.3. Nodo descendiente del subárbol derecho con menor valor de clave.

Se ilustra un ejemplo de árbol binario de búsqueda, en el cual si t apunta a la raíz, se tendrá que el nodo con clave 6 es el menor descendiente del subárbol derecho.

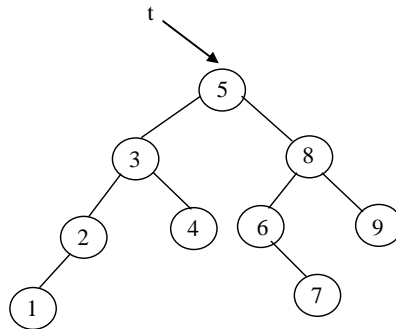


Figura 6.29. Menor descendiente subárbol derecho.

Diseño recursivo.

Se emplea la función BuscaMínimo, que es recursiva.

```

pnodo MenorDescendienteSD(pnodo t)
{
    if (t == NULL) return(NULL); //si árbol vacío retorna NULL
    if (t->right == NULL) return(NULL); // Si no tiene hijo derecho no hay sucesor.
    return( BuscaMinimo(t->right) ); //sigue buscando en subárbol derecho.
}

```

Para el diseño iterativo, cuando existe subárbol derecho, deben estudiarse dos casos, los cuales se ilustran en la Figura 6.30.

El caso D1, un nodo sin hijo izquierdo, indica que se encontró el mínimo.

El caso D2, debe descenderse por el subárbol derecho de t , por la izquierda, mientras se tengan hijos por la izquierda.

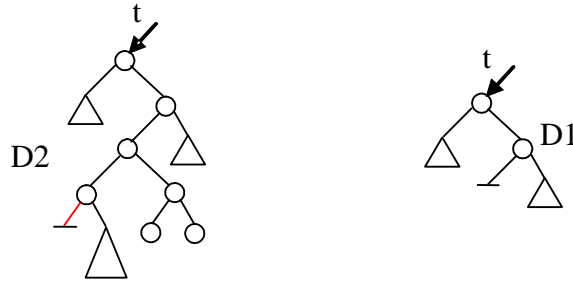


Figura 6.30. Casos en búsqueda del menor descendiente

```

pnodo MenorDescendienteIterativoSD(pnodo t)
/*menor descendiente de subárbol derecho. */
{ pnodo p;
  if (t == NULL) return(NULL); //si árbol vacío retorna NULL
  if (t->right == NULL) return(NULL); // Si no tiene hijo derecho no hay sucesor.
  else p = t->right;
  while ( p->left != NULL) { /* Mientras no tenga hijo izq descender por la izq. */
    p = p->left;
  }
  /*Al terminar el while p apunta al menor descendiente */
  return (p);      /* Retorna el menor */
}

```

6.6.3.4. Sucesor.

Dado un nodo encontrar su sucesor no es el mismo problema anterior, ya que el nodo podría ser una hoja o un nodo sin subárbol derecho. Por ejemplo en la Figura 6.29, el sucesor del nodo con clave 4 es el nodo con clave 5. El sucesor del nodo 2 es el nodo con valor 3.

Se requiere disponer de un puntero al padre del nodo, para que la operación sea de costo logarítmico, en promedio.

Si un nodo tiene subárbol derecho, el sucesor de ese nodo es el ubicado más a la izquierda en ese subárbol (problema que se resolvió en 6.6.3.3); si no tiene subárbol derecho, es el menor ancestro (que está sobre el nodo en la trayectoria hacia la raíz) que tiene a ese nodo en su subárbol izquierdo.

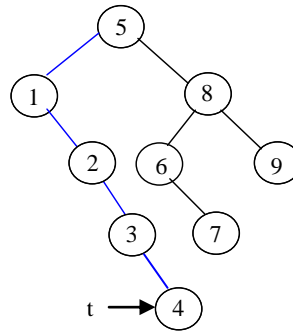


Figura 6.31. Sucesores de distintos nodos.

Algoritmo:

Si el árbol no es vacío.

Si no tiene subárbol derecho:

Mientras exista el padre y **éste apunte al nodo dado por la derecha** se asciende:

Hasta encontrar el primer padre por la izquierda.

Si no existe ese padre, se retorna NULL, *t* era el nodo con valor máximo

Si tiene subárbol derecho, el sucesor es el mínimo del subárbol derecho.

Revisar el algoritmo para los diferentes nodos del árbol de la Figura 6.31, especialmente para los nodos con claves 4 y 9. En este último caso debe asumirse que se ha definido que el padre de la raíz tiene valor NULL.

pnodo Sucesor(pnodo t)

```
{ pnodo p;
  if (t == NULL) return(NULL); //si árbol vacío retorna NULL
  if (t->right == NULL)
  { p = t->padre; //p apunta al padre de t
    while( p!=NULL && t == p->right)
      {t=p; p=t->padre;} //se asciende
    return(p); //
  }
  else
    return( BuscaMinimo (t->right) ); //busca mínimo en subárbol derecho.
}
```

Como en peor caso debe ascenderse una trayectoria del nodo hacia la raíz, el costo será $O(a)$, donde a es la altura del árbol.

6.6.3.5. Nodo descendiente del subárbol izquierdo con mayor valor de clave.

Basta intercambiar left por right y viceversa en el diseño desarrollado en 6.6.3.3.

6.6.3.6. Predecesor.

El código de la función predecesor es la imagen especular del código de sucesor.

6.6.3.7. Buscar

Es una de las operaciones más importantes de esta estructura. Debido a la propiedad de los árboles binarios de búsqueda, si el valor buscado no es igual al de nodo actual, sólo existen dos posibilidades: que sea mayor o que sea menor. Lo que implica que el nodo buscado puede pertenecer a uno de los dos subárboles. Cada vez que se toma la decisión de buscar en uno de los subárboles de un nodo, se están descartando los nodos del otro subárbol. En caso de árboles balanceados, se descarta la mitad de los elementos de la estructura, esto cumple el modelo: $T(n) = T(n/2) + c$, lo cual asegura costo logarítmico.

```
pnodo BuscarIterativo( pnodo t, int valor)
{
    while ( t != NULL)
    {
        if ( t->clave == valor ) return (t);
        else {
            if (t->clave < valor ) t = t->right; //desciende por la derecha
            else t = t->left;                //desciende por la izquierda
        }
    }
    return (t); /* NULL No lo encontró*/
}
```

Es preciso tener implementados los operadores de **igualdad** y **menor que**, en caso de que éstos no existan en el lenguaje, para el tipo de datos de la clave. Por ejemplo si la clave es alfanumérica (un string), una estructura, etc.

Complejidad de la búsqueda.

Si $T(a)$ es la complejidad de la búsqueda en un árbol de altura a .

En cada iteración, el problema se reduce a uno similar, pero con la altura disminuida en uno, y tiene costo constante el disminuir la altura.

Entonces:

$$T(a) = T(a-1) + \Theta(1) \text{ con } T(0) = 0$$

La solución de esta recurrencia, es:

$$T(a) = a * \Theta(1) = \Theta(a)$$

Pero en árboles de búsqueda se tiene que: $\log_2 n \leq a \leq n$

Entonces: $\Theta(\log_2 n) \leq T(a) \leq \Theta(n)$

```

pnodo BuscarRecursivo( pnodo t, int valor )
{
    if ( t == NULL) return (NULL); /* árbol vacío o hijo de hoja */
    else {
        if ( t->clave == valor ) return(t); /* lo encontró */
        else {
            if ( t->clave > valor ) t = BuscarRecursivo ( t->left, valor);
            else t = BuscarRecursivo ( t->right, valor);
        }
    }
    return ( t ); /* ! Si se entiende esta línea, muestra que se entiende el diseño recursivo */
}

```

En caso de activarse una secuencia de llamados recursivos, **los retornos de éstos**, son pasados a través de la asignación a la variable **t**.

Pueden eliminarse las **asignaciones** y el **retorno** final, del diseño anterior, de la siguiente forma:

```

pnodo BuscarRecursivo2( pnodo t, int valor )
{
    if ( t == NULL) return (NULL); /* árbol vacío o hijo de hoja */
    else {
        if ( t->clave == valor )
            return (t); /* lo encontró */
        else
        {
            if ( t->clave > valor )
                return ( BuscarRecursivo2 ( t->left, valor) );
            else
                return ( BuscarRecursivo2 ( t->right, valor) );
        }
    }
}

```

En caso de retorno nulo, no es posible determinar si no encontró el elemento buscado o si se trataba de un árbol vacío.

6.6.4. Operaciones de modificación

6.6.4.1. Insertar nodo

Diseño iterativo.

Primero se busca el sitio para insertar. Si el valor que se desea insertar ya estaba en el árbol, no se efectúa la operación; ya que no se aceptan claves duplicadas. Entonces: se busca el valor; y si no está, se inserta el nuevo nodo.

Es preciso almacenar en la variable local q , la posición de la hoja en la que se insertará el nuevo nodo. En la Figura 6.31.a, se desea insertar un nodo con valor 4; se muestra el descenso a partir de la raíz, hasta encontrar el nodo con valor 3, que tiene subárbol derecho nulo.

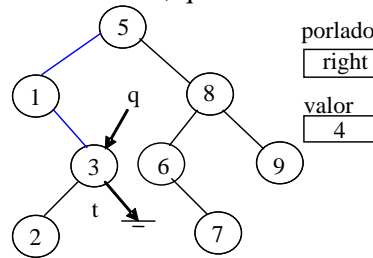


Figura 6.31.a. Variables al salir del while.

```
typedef enum {left, right, vacio} modo;
pnodo InsertarIterativo(pnodo t, int valor)
{ pnodo q= t;
  modo porlado=vacio;
  while ( t != NULL)
  {
    if ( t->clave == valor )
      { /*lo encontré, no inserta. No se aceptan claves repetidas en conjuntos*/
        return (t);
      }
    else
      { q=t ;
        if (t->clave < valor) { t = t->right; porlado=right; }
        else { t = t->left; porlado=left; }
      }
  }
  /*Al salir del while q apunta al nodo donde se insertará el nuevo, y porlado la dirección */
  /* El argumento t apunta a NULL */
  t = CreaNodo(valor); //se pega el nuevo nodo en t.
  if (porlado==left) q->left=t; else if (porlado==right) q->right=t;
  return (t); /* Apunta al recién insertado. Null si no se pudo insertar*/
}
```

Si CreaNodo retorna un NULL, si no había espacio en el heap, sin invocar a exit, se tendrá un retorno nulo, si no se pudo insertar.

Si p y $raiz$ son de tipo pnodo, el siguiente segmento ilustra un ejemplo de uso de la función, considerando la inserción en un árbol vacío:

```
if (raiz==NULL) raiz=InsertarIterativo(raiz, 4);
else if ( (p=InsertarIterativo(raiz, 4))==NULL ) printf("error");
```

Dejando en p el nodo recién ingresado, o el ya existente con ese valor de clave.

Se recorre una trayectoria de la raíz hasta una hoja. Entonces, si a es la altura, la complejidad de la inserción es: $T(a)$.

Una alternativa al diseño iterativo, es mantener un puntero al puntero izquierdo o derecho, en la posición para insertar. En la descripción del descarte iterativo se dan explicaciones más completas sobre la variable local p , que es puntero a un puntero.

En la Figura 6.31.b, se desea insertar un nodo con valor 4; se muestra el descenso a partir de la raíz, hasta encontrar el nodo con valor 3, que tiene subárbol derecho nulo. Nótese que p queda apuntando a un puntero.

```
pnodo Insert2(pnodo t, int valor)
{
    pnodo *p = &t;
    while (*p != NULL) {
        if ((*p)->valor < valor) p = &((*p)->right);
        else if ((*p)->valor > valor) p = &((*p)->left);
        else { /* Ya estaba. No hace nada */
            return (*p);
        }
    }
    return( *p = getnodo(valor) );
}
```

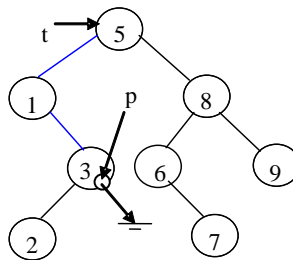


Figura 6.31.b. Variables al salir del while.

La inserción en un árbol vacío debe condicionarse, para poder escribir en la variable externa, denominada *raiz*. Si p y *raiz* son de tipo pnodo, el siguiente segmento ilustra el uso de la función, considerando la inserción en un árbol vacío:

```
if (raiz==NULL) raiz=Insert2(raiz, 4);
else if ( (p=Insert2(raiz, 4))==NULL ) printf("error");
```

Dejando en p el nodo recién ingresado, o el ya existente con ese valor de clave.

Diseño recursivo.

```

pnodo InsertarRecursivo( pnodo t, int valor)
{
  if (t == NULL) t = CreaNodo(valor); //insertar en árbol vacío o en hoja.
  else
    if (valor < t->clave) //insertar en subárbol izquierdo.
      t->left = InsertarRecursivo(t->left, valor);
    else
      if (valor > t->clave) //insertar el subárbol derecho
        t->right = InsertarRecursivo (t->right, valor);
      /* else: valor ya estaba en el árbol. No hace nada. */
  return(t);
}

```

La inserción en un árbol vacío debe poder escribir en la variable externa, denominada *raíz*. El siguiente segmento ilustra el uso de la función:

```
raíz=InsertRecursivo(raíz, 4);
```

Debe notarse que se reescriben los punteros de los nodos que forman la ruta de descenso, lo cual es una sobrecarga de trabajo innecesario, salvo que el último retorno escribe en la raíz. Si desea conocerse si *CreaNodo* falla, retornando *NULL*, habría que comentar la excepción o salir del programa.

Trayectoria en el descenso.

Veremos que en el algoritmo recursivo, la trayectoria recorrida desde la raíz hasta la posición para insertar queda registrada en el stack.

En el árbol de la Figura 6.32, se especifican los valores de los punteros, y se desea insertar un nodo con clave igual a 7.

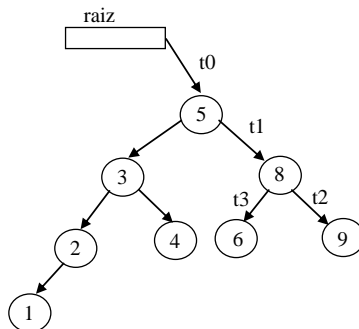


Figura 6.32. Trayectorias en llamados recursivos.

Se ilustra el stack, después del llamado: *InsertarRecursivo(raíz, 7)*

t	valor	Llamado número
t0	7	1

Figura 6.33. Stack después de InsertarRecurso(raiz, 7).

Al ejecutarse el código de la función, se determina que 7 es mayor que 5, y se reinvoca (segundo llamado) con los valores: InsertarRecurso (t1, 7); después del llamado, el stack puede visualizarse:

t	valor	Llamado número
t0	7	1
t1	7	2

Figura 6.34. Stack después de InsertarRecurso(t1, 7).

Al ejecutarse el código de este segundo llamado se determina que 7 es menor que 8 y se genera un tercer llamado a la función con: InsertarRecurso (t3, 7); después de este tercer llamado, el stack queda:

t	valor	Llamado número
t0	7	1
t1	7	2
t3	7	3

Figura 6.35. Stack después de InsertarRecurso(t3, 7).

Al ejecutar el código del tercer llamado, se determina que 7 es mayor que 6, y se produce el cuarto llamado: InsertarRecurso (t3->right, 7); si denominamos por *t4* al valor t3->right, el esquema que muestra las variables en el stack, queda como sigue:

t	valor	Llamado número
t0	7	1
t1	7	2
t3	7	3
t4	7	4

Figura 6.36. Stack después de InsertarRecurso(t3->right, 7).

Al iniciar la ejecución del código del cuarto llamado, se tiene en el stack los valores de los punteros que recuerdan la **trayectoria del descenso** hasta la posición de inserción. Al ejecutar el código del cuarto llamado se determina que *t4* es un puntero nulo, con lo cual, se crea el nodo y se retorna en *t* (que es *t4*, la cuarta encarnación de *t*) el valor de un puntero al nodo recién creado. En este momento se sale del cuarto llamado y el stack queda:

t	valor	Llamado número
t0	7	1
t1	7	2
t3	7	3

Figura 6.37. Stack después del retorno del cuarto llamado.

Se regresa a la ejecución del código del tercer llamado, efectuando la asignación:

$t \rightarrow \text{right} = \langle \text{valor del puntero retornado } t4 \rangle$

Pero como, dentro del tercer llamado, t tiene el valor de $t3$, esta instrucción pega efectivamente el nuevo nodo. Esta instrucción es la última del tercer llamado, con lo cual termina retornando el valor $t3$. El stack queda ahora:

t	valor	Llamado número
t0	7	1
t1	7	2

Figura 6.38. Stack después del tercer retorno.

Y reanudamos la ejecución del segundo llamado, que había quedado pendiente. Efectuando la asignación:

$t \rightarrow \text{left} = \langle \text{valor retornado de } t \text{ por el tercer llamado } t3 \rangle$

Sobrescribiendo un valor igual al existente, copia en $t1 \rightarrow \text{left}$ el valor de $t3$. Obviamente esta escritura es un costo adicional de la recursividad. Terminando así el segundo llamado, el cual retorna el valor de $t1$.

El stack queda:

t	valor	Llamado número
t0	7	1

Figura 6.39. Stack después del retorno del segundo llamado.

Y se reanuda la ejecución, efectuando:

$t \rightarrow \text{right} = \langle \text{valor retornado de } t \text{ por el segundo llamado. } t1 \rangle$

La cual sobrescribe, nuevamente en forma innecesaria en $t0 \rightarrow \text{left}$ el valor de $t1$.

De esta forma finaliza, recién, la primera invocación, retornando el valor de $t0$.

La expresión: InsertarRecursivo(raiz, 7), después de ejecutada, tiene el valor de $t0$.

Entonces la forma de invocar a esta función es:

```
raiz=InsertarRecurso(raiz, 7);
```

De este modo la inserción en un árbol vacío, liga correctamente el nodo agregado.

Si el valor del nodo que se desea insertar es igual a uno ya perteneciente al árbol, el llamado también retorna *t0*.

Si el llamado a CreaNodo falla por no disponer de memoria en el heap, y en su diseño se hubiera retornado un NULL (sin invocar a exit) no habría forma de conocer que la inserción falló.

El diseño recursivo recorre desde la raíz hasta el punto de inserción, quedando la ruta de descenso en el stack; luego de la inserción, recorre la ruta en sentido inverso; lo cual permite agregar alguna operación a los nodos involucrados. La acción debe realizarse antes del retorno. Ver el punto 6.6.5.11 Inserción en la raíz.

La operación insertar en el lugar de la raíz, es más compleja, ya que requiere modificar el árbol. Requiere primero insertar el nodo, y luego efectuar rotaciones para llevar ese nodo al lugar de la raíz, preservando la propiedad de un árbol binario de búsqueda. También se puede implementar si se dispone de una función que parta un árbol en dos (split).

6.6.4.2. Descartar nodo

Descarte recursivo.

Primero se busca el nodo cuyo valor de clave es igual al valor pasado como argumento. Si no lo encuentra retorna NULL. Si lo encuentra, la operación requiere mayor análisis, ya que se producen varios casos. Lo importante es mantener la vinculación entre el resto de los elementos del árbol.

a) El nodo que se desea descartar es una hoja.

En este caso, la operación es trivial, basta escribir un puntero con valor **nulo**. La estructura se conserva.

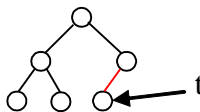


Figura 6.40. Descartar hoja

b) El nodo que se desea descartar es un nodo interno.

i) con un hijo

En este caso, el padre debe apuntar al nieto, para conservar la estructura de árbol. Ya sea que sólo tenga hijo derecho o izquierdo. Esto implica mantener un puntero al padre, en el descenso.

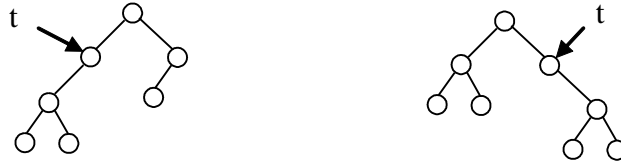


Figura 6.41. Descartar nodo con un subárbol

ii) con dos hijos.

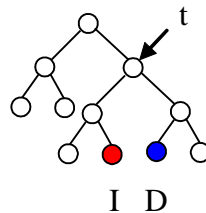


Figura 6.42. Descartar nodo con dos hijos.

Para conservar la estructura del árbol, se debe buscar I, el mayor descendiente del hijo izquierdo; o bien D, el menor descendiente del hijo derecho. Luego reemplazar la hoja obtenida por el nodo a descartar. Se implementa la operación buscando D.

```
pnodo Descartar(pnodo t, int valor)
{ pnodo temp;
  if (t == NULL) printf("Elemento no encontrado\n");
  else
    if (valor < t->clave) /* por la izquierda */
      t->left = Descartar(t->left, valor);
    else
      if (valor > t->clave) /* por la derecha */
        t->right = Descartar(t->right, valor);
      else /* se encontró el elemento a descartar */
        if (t->left && t->right) /* dos hijos */
        {
          /* reemplázelo con el menor del subárbol derecho. D*/
          temp = MenorDescendiente(t->right);
          t->clave = temp->clave; //copia el nodo
          t->right = Descartar(t->right, temp->clave); /*borrar la hoja */
        }
        else
        { /* un hijo o ninguno */
          temp = t;
          if (t->left == NULL) /* sólo hijo derecho o sin hijos */
```

```

        t = t->right;
    else
        if (t->right == NULL) /* solamente un hijo izquierdo */
            t = t->left;
        free(temp); /*libera espacio */
    }
    return(t);
}

```

La complejidad del descarte de un nodo es mayor que la inserción o la búsqueda.

La operación puede implementarse en forma iterativa.

Descarte iterativo.

En **algoritmos iterativos**, es preciso mantener la información de la trayectoria del descenso en una variable auxiliar; en los algoritmos recursivos, se mantiene esa información en el stack. Bastaría tener dos punteros, uno al nodo actual, y otro al anterior; sin embargo existe la dificultad, a diferencia de listas simplemente enlazadas, de que el anterior podría ser un descenso por la izquierda o por la derecha. Para solucionar lo anterior, se mantiene un *puntero al puntero anterior*.

Veamos una ilustración, para mantener un puntero a un puntero al anterior:

Si se tiene la definición e inicialización: `pnodo *p = &t;`

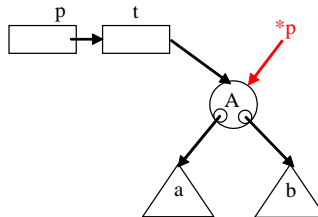


Figura 6.43. Puntero a puntero a nodo.

Entonces `(*p)->clave` tiene el valor de la clave A. Note que `*p` equivale a la variable t.

Si se efectúa la asignación: `p = &((*p)->right);` se modifica el diagrama según:

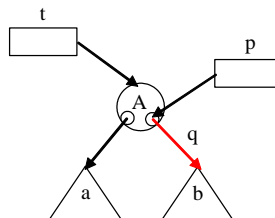


Figura 6.44. Memorización de trayectoria de descenso.

Ahora `*p` contiene la dirección del puntero ilustrado como q en el diagrama.

La asignación: $p = \&((*p)\text{->left})$; deja a p apuntando al puntero izquierdo del nodo con valor A.

Con estos conceptos se elabora el siguiente algoritmo iterativo.

```
pnodo DescarteIterativo(pnodo t, int valor)
{
    pnodo *p = &t;
    pnodo temp;
    while (*p != NULL) {
        if ((*p)->clave < valor) p = &((*p)->right);
        else if ((*p)->clave > valor) p = &((*p)->left);
        else { /* La encontré */
            if ((*p)->left == NULL) {temp = *p; *p = (*p)->right; free(temp); }
            else if ((*p)->right == NULL) {temp = *p; *p = (*p)->left; free(temp); }
            else /* Tiene ambos hijos */
                *p = Descarte_Raiz(*p);
            return t;
        }
    }
    Error(); /*No encontró nodo con clave igual a valor */
    return t;
}
```

El descarte de la raíz, o de un nodo con dos hijos, está basado en encontrar el menor descendiente del hijo derecho, o el mayor descendiente del hijo izquierdo.

```
pnodo Descarte_Raiz(pnodo t)
{
    pnodo *p = NULL, temp;
    if (rand()%2) { /*Existen dos soluciones */
        /* Busca mayor descendiente del subárbol izquierdo */
        p = &(t->left);
        while ((*p)->right != NULL)
            p = &((*p)->right);
    } else {
        /* o Busca menor descendiente del subárbol derecho */
        p = &(t->right);
        while ((*p)->left != NULL)
            p = &((*p)->left);
    }
    t->clave = (*p)->clave; /*copia los valores del encontrado en la raíz. */
    if ((*p)->left == NULL) {
        temp = *p;
        *p = (*p)->right;
    } else { /* ((*p)->right == NULL) */
        temp = *p;
        *p = (*p)->left;
    }
}
```

```

    }
    free(temp);
    return t;
}

```

6.6.4.3. Descartar árbol

Debe notarse que primero deben borrarse los subárboles y luego la raíz.

```

pnodo deltree(pnodo t)
{
    if (t != NULL) {
        t->left = deltree(t->left);
        t->right = deltree(t->right);
        free(t);
    }
    return NULL;
}

```

6.6.5. Otras operaciones

6.6.5.1. Profundidad del árbol.

```

int Profundidad(pnodo t)
{
    int left=0, right = 0;
    if(t==NULL) return 0;    //Si árbol vacío, profundidad 0
    if(t->left != NULL) left = Profundidad(t->left); //calcula profundidad subárbol izq.
    if(t->right != NULL) right = Profundidad(t->right); //calcula profundidad subárbol der.
    if( left > right) //si el izq tiene mayor profundidad
        return left+1; //retorna profundidad del subárbol izq + 1
    else
        return right+1; //retorna profundidad del subárbol der + 1
}

```

El algoritmo se ha descrito mediante los comentarios.

6.6.5.2. Altura del árbol.

```

int Altura(pnodo T)
{
    int h, max;
    if (T == NULL) return -1;
    else {
        h = Altura (T->left);
        max = Altura (T->right);
        if (h > max) max = h;
        return(max+1);
    }
}

```

6.6.5.3. Contar hojas

```

int NumerodeHojas(pnodo t)
{
    int total = 0;
    //Si árbol vacío, no hay hojas
    if(t==NULL) return 0;
    // Si es hoja, la cuenta
    if(t->left == NULL && t->right == NULL) return 1;
    //cuenta las hojas del subárbol izquierdo
    if(t->left!= NULL) total += NumerodeHojas(t->left);
    //cuenta las hojas del subárbol derecho
    if(t->right!=0) total += NumerodeHojas(t->right); //
    return total; //total de hojas en subárbol
}

```

Nuevamente el algoritmo está descrito a través de los comentarios.

6.6.5.4. Contar nodos del árbol.

```

int ContarNodos(pnodo t)
{
    if (t == NULL) return 0;
    return (1 + ContarNodos(t->left) + ContarNodos(t->right) );
}

```

6.6.5.5. Contar nodos internos.

Tarea

6.6.5.6. Contar nodos con valores menores que un valor dado.

Tarea

6.6.5.7. Partir árbol.

Algoritmo iterativo, con pasos por referencia.

La descripción de las funciones split y join, son buenos ejemplos para mostrar si se domina el concepto de punteros en el lenguaje C.

```

pnodo split(int key, pnodo t, pnodo *l, pnodo *r)
{
    while (t != NULL && t->clave != key) {
        if (t->clave < key) {
            *l = t;
            t = t->right;
            l = &((*l)->right);
        } else {
            *r = t;
            t = t->left;
            r = &((*r)->left); //plop
        }
    }
}

```



```

    if (t == NULL) {
        *l = NULL;
        *r = NULL;
    } else { /* t->clave == key */
        *l = t->left;
        *r = t->right;
    }
    return t;
}

```

6.6.5.8. Insertar nueva raíz.

```

pnodo InsertarRaiz(int key, pnodo t)
{
    pnodo l, r;
    t = split(key, t, &l, &r);
    if (t == NULL) {
        t = CreaNodo(key);
        t->left = l;
        t->right = r;
    } else {
        t->left = l;
        t->right = r;
        Error();
    }
    return t;
}

```

6.6.5.9. Unir dos árboles.

Los árboles que deben ser unidos cumplen las siguientes relaciones de orden: $a < A < b$ y $d < D < e$ y tal que $b < d$. Podrían ser dos árboles que se generan al eliminar la raíz de un árbol binario de búsqueda.

Al inicio se tienen las siguientes variables

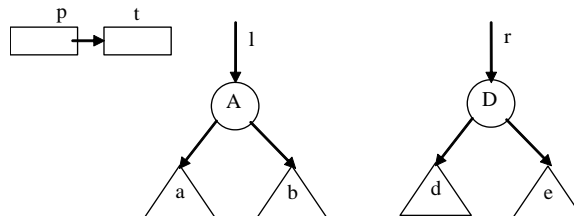


Figura 6.45. Variables en unir dos subárboles.

Si los dos subárboles no son vacíos, en la primera iteración se da valor inicial a t , la variable de retorno que apuntará a la nueva raíz. Ésta puede ser l o r , dependiendo del azar.

Luego se mantiene en p : la dirección del puntero derecho de la raíz, si comienza por el subárbol izquierdo; o la dirección del puntero izquierdo de la raíz, si comienza por el subárbol derecho.

Finalmente cada iteración finaliza haciendo descender l por la derecha o r por la izquierda.

En el supuesto que comienza por el subárbol izquierdo, la situación luego de la primera iteración del while, efectuando la parte del if, es:

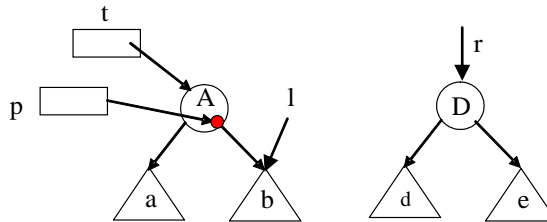


Figura 6.46. Parte del if dentro del while.

Si en la segunda iteración, realiza la parte del else, después de efectuado éste, queda:

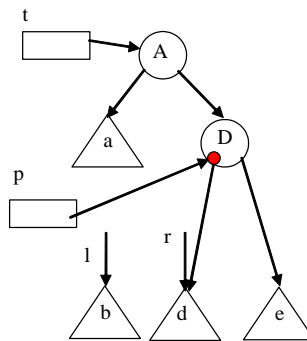


Figura 6.47. Parte del else dentro del while.

Lográndose que l y r apunten a dos subárboles. Se repite el while, hasta que uno de los dos subárboles quede vacío. La alternativa final, pega el subárbol restante, a la estructura.

La solución trivial de agregar el subárbol derecho al nodo con valor mayor de clave del subárbol cuya raíz es l , alarga la altura en la suma de las alturas individuales. Lo mismo ocurre si se une l al menor elemento de r .

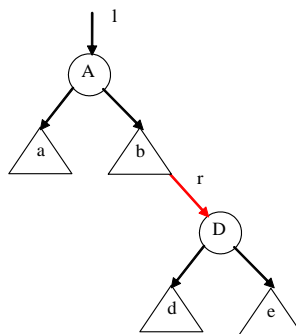


Figura 6.48. Unión de los árboles.

La solución, propuesta a continuación, intenta disminuir la altura total del nuevo árbol.

```
pnodo join(pnodo l, pnodo r)
{
    pnodo t = NULL;
    pnodo *p = &t;
    while (l != NULL && r != NULL) {
        if (rand()%2) { //cara y sello.
            *p = l;
            p = &((*p)->right);
            l = l->right;
        } else {
            *p = r;
            p = &((*p)->left);
            r = r->left;
        }
    }
    if (l == NULL) *p = r;
    else /* (r == NULL) */ *p = l;
    return t;
}
```

La operación de descartar la raíz, DescartarRaiz, también puede implementarse en base a la función join, que une dos árboles conservando la propiedad de árbol binario de búsqueda.

La operación de descartar la raíz y pegar los subárboles, se ilustra a continuación.

```
pnodo DescartarRaiz(pnodo t)
{
    pnodo temp = t;
    t = join(t->left, t->right);
    free(temp);
    return t;
}
```

Las rutinas split y join usan intensivamente punteros. Entenderlas es un indicador que esos conceptos y sus principales usos han logrado ser dominados.

6.6.5.10. Rotaciones

Rotaciones simples a la izquierda y a la derecha

Un esquema de las variables, que permiten diseñar las funciones, se ilustra en las Figuras 6.49 y 6.50.

```

pnodo lrot(pnodo t)
{
    pnodo temp=t;
    t = t->right;
    temp->right = t->left;
    t->left = temp;
    return t;
}

```

```

pnodo rrot(pnodo t)
{
    pnodo temp = t;
    t = t->left;
    temp->left = t->right;
    t->right = temp;
    return t;
}

```

Las siguientes funciones son mejores que las anteriores, ya que tienen una asignación menos, sólo escriben en tres punteros.

```

/* Rotación Izquierda*/
pnodo rotL(pnodo t)
{
    pnodo temp = t->right;
    t->right = temp->left;
    temp->left = t;
    return (temp);
}

```

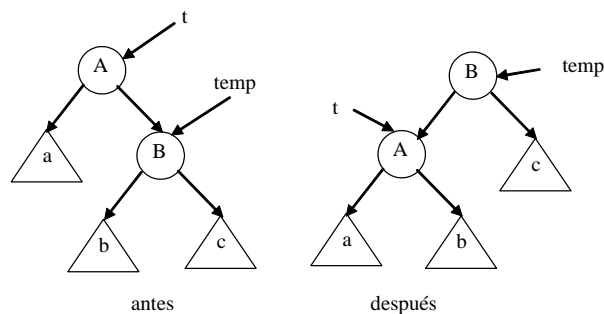


Figura 6.49. Rotación izquierda.

```

/* Rotación derecha*/
pnodo rotR(pnodo t)
{
    pnodo temp = t->left;
    t->left = temp->right;
    temp->right = t;
    return (temp);
}

```

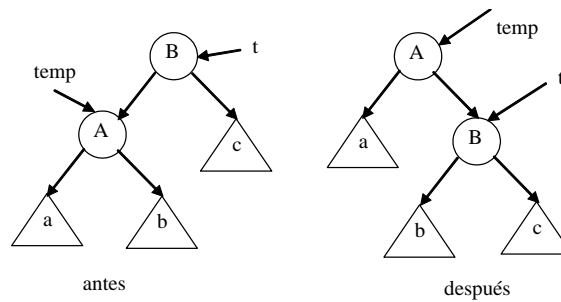


Figura 6.50. Rotación derecha.

Las rotaciones pueden cambiar la forma del árbol, pero no la relación de orden; en el caso de las figuras anteriores se preserva la relación: $a < A < b < B < c$.

Solamente afectan a los nodos rotados y sus hijos inmediatos, los ancestros y los nietos no son modificados.

Pueden diseñarse funciones que pasen argumentos por referencia. Se ilustra el diseño de la rotación simple a la derecha:

```
void rightRotRef( pnode * t) //por referencia
{
    pnode temp = (*t)->left;
    (*t)->left = temp->right;
    temp->right = *t;
    *t = temp; //modifica la variable pasada por referencia
}
```

En este caso se debe pasar la dirección de la variable donde debe escribir la función.

`rightRotRef(&(root->right));`

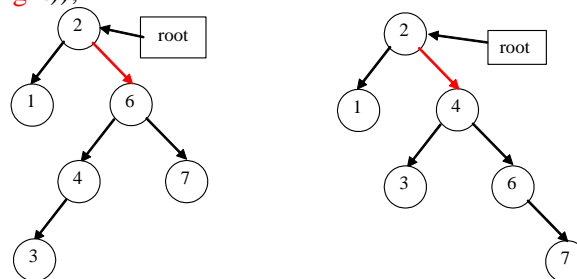


Figura 6.51. Ejemplo de Rotación derecha de nodos 4-6.

El diagrama a la derecha ilustra la rotación efectuada, la raíz no cambia.

En el diseño basado en retornos por punteros, es preciso escribir en una variable, mediante el retorno de la función.

La siguiente asignación realiza la misma acción que la invocación al procedimiento anterior.

`root->right=rotR(root->right);`

6.6.5.11. Inserción en la raíz.

En la inserción común en un árbol binario de búsqueda, los elementos recién insertados quedan alejados de la raíz, lo que implica que toma más tiempo encontrar los elementos insertados más recientemente. Lo cual puede ser un inconveniente si en la aplicación los elementos recién insertados tienden a ser buscados más a menudo que los insertados hace más tiempo; es decir si existe localidad temporal en las referencias.

La inserción en la raíz coloca el nodo que será insertado en la posición de la raíz actual. De este modo los nodos más recientemente insertados quedan más cercanos a la raíz que los más antiguamente insertados.

Para mantener la propiedad del árbol binario de búsqueda, se inserta de manera convencional, como una hoja, y luego mediante rotaciones se lo hace ascender a la posición de la raíz.

En el diagrama, se ilustra la inserción de un nodo con valor de clave 5, en la posición de una hoja. Luego se lo hace ascender, rotando el par 4-5 a la izquierda; luego el par 5-6 a la derecha; y finalmente el par 2-5 a la izquierda.

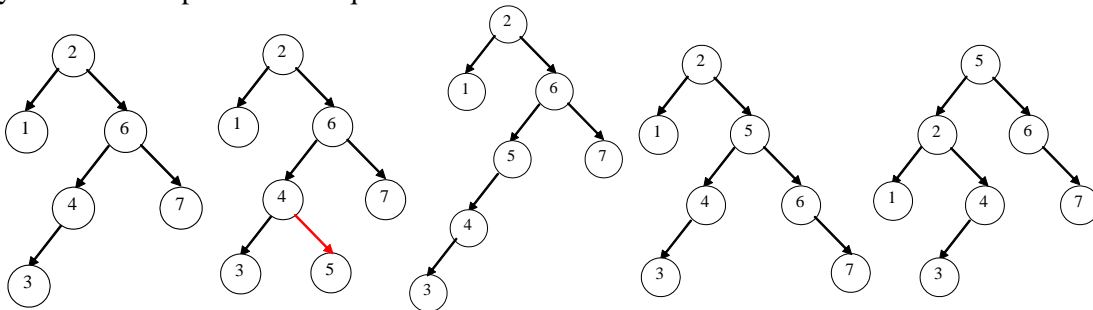


Figura 6.52. Inserción de nodo con clave 5 en la raíz.

El algoritmo está basado en observar que cuando se desciende por la izquierda desde un nodo n , durante la búsqueda para insertar, se debe rotar hacia la derecha en n . Y si se desciende por la rama derecha, ese nodo debe rotarse hacia la izquierda. En el nodo con clave 6, se descendió por la izquierda; luego el nodo 6 se rota a la derecha.

A medida que se desciende, para buscar la posición para insertar, se registra el nodo; luego después de la inserción, se retorna a cada uno de estos nodos, en orden inverso, y se realiza la rotación.

```

/*Inserta un nodo y lo convierte en la nueva raíz */
pnodo InserteRaiz(pnodo t, int valor)
{
    if (t == NULL) return ( CreaNodo(valor));
    if (valor < t->clave)
    {
        t->left = InserteRaiz(t->left, valor);
        t = rotR(t);
    }
    else {
        t->right = InserteRaiz(t->right, valor);
        t = rotL(t);
    }
    return t;
}

```

6.6.5.12. Imprimir la forma del árbol.

Las dos rutinas siguientes permiten desplegar un árbol, y pueden ser útiles para verificar las funciones que los manipulan.

```

void printNodo(pnodo t, int h)
{
    int i;
    for(i=0; i<h; i++) putchar('t'); //se emplean tabs para desplegar niveles.
    if(t==NULL) {putchar('*') ; putchar('\n') ;}
    else printf("%d\n", t->clave);
}

```

```

void Mostrar(pnodo t, int h)
{
    if(t==NULL) printNodo(t, h);
    else {Mostrar(t->right, h+1) ; printNodo(t, h); Mostrar(t->left, h+1);}
}

```

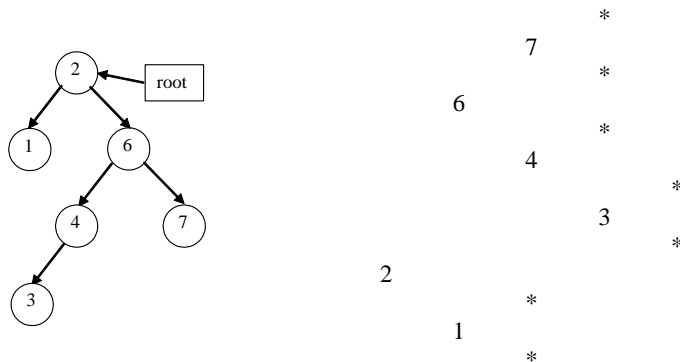


Figura 6.53. Impresión de la forma de un árbol.

La ejecución de `Mostrar(root,0)`; muestra un ejemplo del despliegue de un árbol cuyo diagrama se ilustra en la Figura 6.53 izquierda.

Problemas resueltos.

P6.1.

Para la siguiente estructura de un nodo de un árbol binario de búsqueda:

```
typedef struct tnode
{
    int valor;
    char *v;    //se apunta a un string
    struct tnode *left, *right;
} nodo, *pnodo;
```

- Diseñar función que borre el subárbol apuntado por t , liberando todo el espacio que haya sido solicitado en forma dinámica.
- Diseñar función que cuente en el subárbol, apuntado por t , los nodos que tengan valores menores o iguales que k .

Solución.

- Puede diseñarse considerando:

Si el subárbol es vacío, retorna NULL

Si no es vacío: Borra subárbol izquierdo y luego el derecho; después de lo cual borra el nodo.

Previo a borrar el nodo; es decir, antes de borrar el puntero v , debe consultarse si existe un string, en caso de haberlo, se borra éste primero, y luego el nodo.

Una posible implementación es la siguiente:

```
pnodo BorrarArbol(pnodo t)
{
    if (t != NULL) {
        t->left = BorrarArbol (t->left);
        t->right = BorrarArbol (t->right);
        if (t->v != NULL) free(t->v);
        free(t);
    }
    return NULL;
}
```

Es preciso que la función retorne un puntero a nodo, para pasar los datos de los retornos de los llamados recursivos. El esquema anterior, borra primero las hojas. El recorrido del árbol es subárbol izquierdo, subárbol derecho y finalmente la raíz (en orden).

Para agregar un string s a un nodo apuntado por t , puede emplearse:


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

```
char * ColocaString( pnode t, char * s)
{
    if ( (t->v = (char *) malloc( (strlen(s)+1)*sizeof(char) ) ) != NULL ) strcpy( t->v, s);
    return ( t->v);
}
```

El espacio adicional se requiere debido a que strlen retorna el largo del string, sin incluir el carácter de fin de string.

La function retorna NULL, si no pudo pegar el string al nodo, en caso contrario retorna un puntero al inicio del string.

b) Contar en subárbol apuntado por t los nodos, que tengan valores menores o iguales que el valor entero k.

```
int ContarMenor_o_Igual(pnode t, int k)
{
    if (t == NULL) return 0;    //Si es subárbol nulo, no lo cuenta
    else
        if (t->valor < k)
        { // Si el valor es menor que k, contar el nodo y además los de ambos subárboles
            return (1 + ContarMenor_o_Igual (t->left,k) + ContarMenor_o_Igual (t->right,k) );
        }
        else
            if (t->valor == k)
            { // Si valor igual a k, contar el nodo y además sólo los del subárbol izquierdo

                return (1 + ContarMenor_o_Igual (t->left, k) );
            }
            else
            { // Si valor mayor que k, contar sólo los nodos del subárbol izq. Sin incluirlo
                return ( ContarMenor_o_Igual (t->left, k) );
            }
        }
}
```

El valor retornado por la función corresponde al número de nodos que cumplen la condición.

P6.2.

1. Se tiene el siguiente árbol de búsqueda.

- ¿Cuáles son los órdenes posibles en los que llegaron las claves para formar el árbol?.
- En un listado post-orden quienes figuran antes y después del valor 6.
- En un listado post-orden quienes figuran antes y después del valor 2.
- Dibujar el árbol, luego de: insertar el nodo con valor 5, y descartar los nodos con valores 4 y luego el 7. Indicar alternativas de solución, si las hubiera.

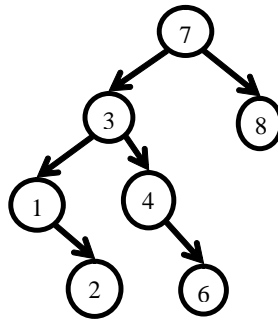


Figura P6.1.

Solución.

- Llega primero el 7. Luego pueden llegar el 3 o el 8. Luego del 3 pueden llegar el 1 o el 4. El 2 debe llegar después del 1; y el 6 después del 4.
- 2 1 6 4 3 8 7. El 1 antes del 6, y el 4 luego de éste.
- El 2 es el primero, luego viene el 1. No hay nada antes del 2, ya que es el primero.
- Luego de insertar el 5 y descartar el 4, se tienen dos posibles soluciones para descartar el nodo con valor 7:

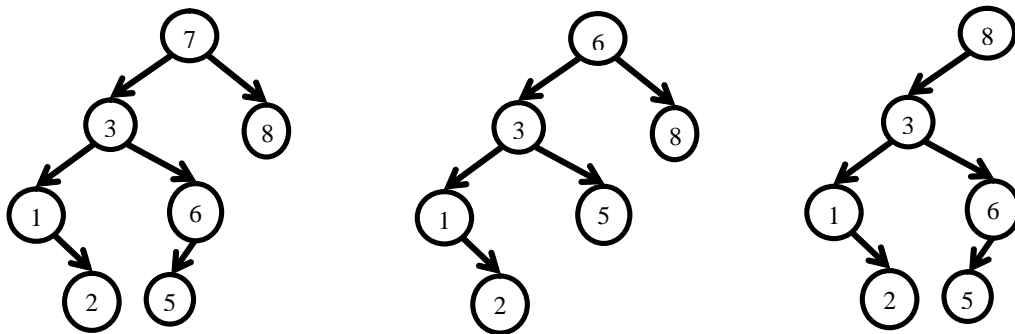


Figura P6.2.

P6.3.

Para un árbol binario de búsqueda, diseñar una función, con el siguiente prototipo:
`int trayectoria(pnodo t, int valor);`

Retorna 1 si lo encontró, 0 si árbol vacío o no lo encontró.

- Que imprima la trayectoria desde el nodo con clave igual al argumento valor hasta la raíz, si éste se encuentra en el árbol, y “no encontrado” en caso contrario.
- Discuta la forma de diseño, si se desea imprimir los valores de los nodos desde la raíz, hasta el nodo que tiene igual valor de clave que el argumento dado.

Solución.

a) El diseño recursivo almacena en el stack los punteros a los nodos en la ruta de descenso desde la raíz al nodo con el valor buscado, si éste se encuentra en el árbol. Si no se encuentra el valor buscado, se desciende hasta encontrar un valor nulo. Entonces debe descenderse, pero no imprimir si se llega a sobrepasar una hoja en el descenso.

Un diseño posible es mediante una variable flag, para indicar si debe o no imprimirse los valores, luego de los retornos de la función recursiva.

```
static int flag=0;
int imprimetrayectoria(pnodo t, int valor)
{ if (t == NULL)
  { printf("\nArbol vacío o no lo encontró!\n");
    flag=0; return(flag);
  }
  else
    if (valor < t->valor)
      { imprimetrayectoria(t->left,valor);
        if(flag==1) printf(" %d", t->valor);
      }
    else
      if (valor > t->valor)
        { imprimetrayectoria(t->right,valor);
          if(flag==1) printf(" %d", t->valor);
        }
      else /* lo encontró. */
        { printf("\n%d", t->valor);
          flag=1;
        }
    return(flag);
}
```

No es necesaria la variable flag, ya que se puede emplear el retorno entero de la función, como puede apreciarse en el siguiente diseño:

```
int trayectoria2(pnodo t, int valor)
{ if (t == NULL)
  { printf("\nArbol vacío o no lo encontró!\n"); return(0); }
  else
    if (valor < t->valor)
      { if( trayectoria2(t->left,valor)) printf(" %d", t->valor); }
    else
      if (valor > t->valor)
        { if( trayectoria2(t->right,valor)) printf(" %d", t->valor); }
      else /* lo encontró. */
        { printf("\n%d", t->valor); return(1); }
    return(1);
}
```

El stack es lifo.

b) La impresión desde la raíz hasta el nodo con el valor buscado, puede resolverse empleando una cola. Se encolan los valores desde la raíz, pero considerando que si se llega a un puntero nulo, no deben imprimirse. Es más directo en este caso un diseño iterativo.

```
int ImpIterativo(pnodo t, int valor)
{
    creacola();
    while ( t != NULL)
    {
        if ( t->valor == valor )
        {
            encole(); putchar('\n'); /*lo encontré*/
            imprimacola(); return (1);
        }
        else
        {
            encole(); //fifo
            if (t->valor < valor) t = t->right; else t = t->left;
        }
    }
    /*Al salir del while es árbol vacío o no lo encontré */
    if(plista->next==NULL) printf("\nArbol vacío!");
    else { printf("\nNo lo encontré!"); liberacola(); }
    return (0);
}
```

Los tipos y macros para manejar la cola:

```
typedef struct lnode
{
    int valor;
    struct lnode *next;
} nodol, * pnodol; //tipos de la cola

#define creacola() pnodol pfondo= malloc(sizeof(nodol)); \
pnodol plista=pfondo; \
plista->next=NULL;

#define encole() pfondo->next=malloc(sizeof(nodol)); \
pfondo=pfondo->next; \
pfondo->next=NULL; \
pfondo->valor=t->valor;

#define imprimacola() while(plista->next!=NULL) \
{ pfondo=plista; plista=plista->next; \
printf(" %d ",plista->valor); free(pfondo); \
}
```

Cuando el valor no se encuentra, habría que liberar el espacio de la cola.

```
#define liberacola() while(plista->next!=NULL) \
                    { pfondo=plista;plista=plista->next; free(pfondo); \
                    }
```

Otra solución es reemplazar en el código de la parte a), los printf por la acción de empujar los valores a imprimir a un stack. Luego de terminada la función si el stack queda vacío, entonces la clave no se encontró; si no está vacío, se imprimen desde el tope. De esta forma se imprimen los valores desde la raíz hasta la hoja encontrada.

Ejercicio propuestos.

E6.1.

Desarrollar programa que efectúe listado postorden para multiárbol descrito por arreglos, con hijo más izquierdista y hermano derecho.

E6.2.

Para un árbol binario de búsqueda, determinar procedimiento que escriba sólo las hojas.

- a) Desde la hoja más izquierdista hasta la hoja más derechista.
- b) Desde la hoja más derechista hasta la hoja más izquierdista.

E6.3.

Se tiene un árbol binario de búsqueda.

```
typedef struct tnode
{
    int valor;
    struct tnode *left;
    struct tnode *right;
} nodo, * pnodo;
```

Diseñar función no recursiva que borre la raíz de un subárbol, que se pasa como argumento y retorne un puntero a la nueva raíz.

Índice general.

CAPÍTULO 6.	1
ÁRBOLES BINARIOS DE BÚSQUEDA.	1
6.1. DEFINICIONES.	1
<i>Ejemplos de definiciones.</i>	2
6.2. ÁRBOL BINARIO.	2
<i>Definición de tipos de datos.</i>	3
6.3. ÁRBOL BINARIO DE BÚSQUEDA.	3
6.4. CÁLCULOS DE COMPLEJIDAD O ALTURA EN ÁRBOLES.	4
6.4.1. Árbol completo.	4
6.4.2 Árboles incompletos con un nivel de desbalance.	6
6.4.3. Árboles contruidos en forma aleatoria.	7
6.4.4. Número de comparaciones promedio en un árbol binario de búsqueda.	11
6.4.4.1. Árbol binario externo	11
6.4.4.2. Largos de trayectorias interna y externa.	13
6.4.4.3. Búsquedas exitosas y no exitosas.	15
6.5. RECORRIDOS EN ÁRBOLES.	19
6.5.1. En orden:	19
6.5.2. Pre orden:	19
6.5.3. Post orden:	19
6.5.4. Ejemplo de recorridos.	19
6.5.5. Árboles de expresiones.	20
6.5.6. Árboles de derivación.	20
6.6. OPERACIONES EN ÁRBOLES BINARIOS.	20
6.6.1. Operaciones básicas	21
6.6.1.1. Crear árbol vacío.	21
6.6.1.2. Crea nodo inicializado con un valor de clave.	21
6.6.1.3. Ejemplo de uso.	21
6.6.2. Operaciones de recorrido	21
6.6.2.1. Mostrar en orden	21
6.6.2.2. Mostrar en post-orden	23
6.6.2.3. Mostrar en pre-orden.	23
6.6.3. Operaciones de consulta.	23
6.6.3.1. Seleccionar el nodo con valor mínimo de clave.	23
6.6.3.2. Seleccionar el nodo con valor máximo de clave.	24
6.6.3.3. Nodo descendiente del subárbol derecho con menor valor de clave.	25
6.6.3.4. Sucesor.	26
6.6.3.5. Nodo descendiente del subárbol izquierdo con mayor valor de clave.	27
6.6.3.6. Predecesor.	27
6.6.3.7. Buscar	28
Complejidad de la búsqueda.	28
6.6.4. Operaciones de modificación.	29
6.6.4.1. Insertar nodo	29
Diseño iterativo.	29
Diseño recursivo.	32

Trayectoria en el descenso	32
6.6.4.2. Descartar nodo	35
Descarte recursivo.....	35
Descarte iterativo.	37
6.6.4.3. Descartar árbol.....	39
6.6.5. Otras operaciones	39
6.6.5.1. Profundidad del árbol.....	39
6.6.5.2. Altura del árbol.	39
6.6.5.3. Contar hojas	40
6.6.5.4. Contar nodos del árbol.	40
6.6.5.5. Contar nodos internos.	40
6.6.5.6. Contar nodos con valores menores que un valor dado.	40
6.6.5.7. Partir árbol.	40
6.6.5.8. Insertar nueva raíz.....	41
6.6.5.9. Unir dos árboles.	41
6.6.5.10. Rotaciones.....	43
6.6.5.11. Inserción en la raíz.	46
6.6.5.12. Imprimir la forma del árbol.....	47
PROBLEMAS RESUELTOS.....	48
P6.1.	48
P6.2.	49
P6.3.	50
EJERCICIO PROPUESTOS.....	53
E6.1.	53
E6.2.	53
E6.3.	53
ÍNDICE GENERAL.	54
ÍNDICE DE FIGURAS.....	56

Índice de figuras.

FIGURA 6.1. ÁRBOLES.	2
FIGURA 6.2. ÁRBOL BINARIO.	2
FIGURA 6.3. ÁRBOL BINARIO DE BÚSQUEDA.	3
FIGURA 6.4. NO ES ÁRBOL BINARIO DE BÚSQUEDA.	4
FIGURA 6.5. VARIOS ÁRBOLES BINARIOS DE BÚSQUEDA CON DISTINTA FORMA.	4
FIGURA 6.6. ÁRBOL COMPLETO DE NIVEL 1.	5
FIGURA 6.7. ÁRBOL COMPLETO DE NIVEL 2.	5
FIGURA 6.8. ÁRBOL COMPLETO DE NIVEL 3.	5
FIGURA 6.9. ÁRBOLES INCOMPLETOS DE NIVEL 2.	6
FIGURA 6.10. RAÍZ CON VALOR I.	7
FIGURA 6.11. ALTURA DE ÁRBOL GENERADO ALEATORIAMENTE.	10
FIGURA 6.12. ALARGUE DE ALTURA DE ÁRBOL GENERADO ALEATORIAMENTE.	11
FIGURA 6.13. NODOS INTERNOS Y EXTERNOS.	11
FIGURA 6.14. $P(1): N_E = N_I + 1$	12
FIGURA 6.15. $P(2): N_E = N_I + 1$	12
FIGURA 6.16. PRIMER CASO DE $P(N+1): N_E = N_I + 1$	13
FIGURA 6.17. SEGUNDO CASO DE $P(N): N_E = N_I + 1$	13
FIGURA 6.18. $P(1): E(N) = I(N) + (2N + 1)$	13
FIGURA 6.19. $P(2): E(N) = I(N) + (2N + 1)$	14
FIGURA 6.20. PRIMER CASO DE $P(N): E(N) = I(N) + (2N + 1)$	14
FIGURA 6.21. SEGUNDO CASO DE $P(N): E(N) = I(N) + (2N + 1)$	15
FIGURA 6.22. EVALUACIÓN DE $U(0)$	16
FIGURA 6.23. $S(N)$ ES $\Theta(\log_2(N))$	18
FIGURA 6.24. ÁRBOL CON CLAVES $\{N_0, N_1, N_2, N_3, N_4, N_5\}$	20
FIGURA 6.25. ÁRBOL QUE REPRESENTA A: $(A * B) / (C + D)$	20
FIGURA 6.26. ÁRBOL DE DERIVACIÓN.	20
FIGURA 6.27. VARIABLES EN BUSCARMINIMOITERATIVO.	24
FIGURA 6.28. CONDICIONES EN BUSCAMINIMO.	24
FIGURA 6.29. MENOR DESCENDIENTE SUBÁRBOL DERECHO.	25
FIGURA 6.30. CASOS EN BÚSQUEDA DEL MENOR DESCENDIENTE.	26
FIGURA 6.31. SUCESORES DE DISTINTOS NODOS.	27
FIGURA 6.31.A. VARIABLES AL SALIR DEL WHILE.	30
FIGURA 6.31.B. VARIABLES AL SALIR DEL WHILE.	31
FIGURA 6.32. TRAYECTORIAS EN LLAMADOS RECURSIVOS.	32
FIGURA 6.33. STACK DESPUÉS DE INSERTARRECURSIVO(RAIZ, 7).	33
FIGURA 6.34. STACK DESPUÉS DE INSERTARRECURSIVO(T1, 7).	33
FIGURA 6.35. STACK DESPUÉS DE INSERTARRECURSIVO(T3, 7).	33
FIGURA 6.36. STACK DESPUÉS DE INSERTARRECURSIVO(T3->RIGHT, 7).	33
FIGURA 6.37. STACK DESPUÉS DEL RETORNO DEL CUARTO LLAMADO.	34
FIGURA 6.38. STACK DESPUÉS DEL TERCER RETORNO.	34
FIGURA 6.39. STACK DESPUÉS DEL RETORNO DEL SEGUNDO LLAMADO.	34
FIGURA 6.40. DESCARTAR HOJA.	35
FIGURA 6.41. DESCARTAR NODO CON UN SUBÁRBOL.	36
FIGURA 6.42. DESCARTAR NODO CON DOS HIJOS.	36
FIGURA 6.43. PUNTERO A PUNTERO A NODO.	37

FIGURA 6.44. MEMORIZACIÓN DE TRAYECTORIA DE DESCENSO.	37
FIGURA 6.45. VARIABLES EN UNIR DOS SUBÁRBOLES.	41
FIGURA 6.46. PARTE DEL IF DENTRO DEL WHILE.	42
FIGURA 6.47. PARTE DEL ELSE DENTRO DEL WHILE.	42
FIGURA 6.48. UNIÓN DE LOS ÁRBOLES.	42
FIGURA 6.49. ROTACIÓN IZQUIERDA.	44
FIGURA 6.50. ROTACIÓN DERECHA.	45
FIGURA 6.51. EJEMPLO DE ROTACIÓN DERECHA DE NODOS 4-6.	45
FIGURA 6.52. INSERCIÓN DE NODO CON CLAVE 5 EN LA RAÍZ.	46
FIGURA 6.53. IMPRESIÓN DE LA FORMA DE UN ÁRBOL.	47
FIGURA P6.1.	50
FIGURA P6.2.	50