Análisis y Diseño de Algoritmos

El Problema de Búsqueda

Arturo Díaz Pérez

- Conjuntos estáticos
- Arboles de decisión para búsqueda
- Conjuntos dinámicos
- Arboles de búsqueda binaria
  - Análisis del peor caso
  - Análisis del caso promedio
- Arboles 2-3

Planteamiento

- Búsqueda
  - Dado x determinar si $x \in S$.
  - Dado x determinar la posición de x, si $x \in S$.
  - Dado x determinar la $R(x)$, si $x \in S$.
- Características del conjunto S.
  - Estático vs. dinámico
  - Ordenado vs. desordenado
  - Llaves duplicadas vs. llaves únicas

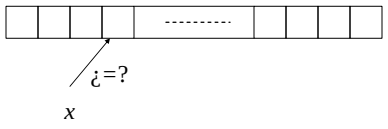
Conjuntos Estáticos

- Inicialmente nos concentraremos en la búsqueda de llaves sobre conjuntos estáticos
  - ← El conjunto de llaves no cambia durante la ejecución
- Dos algoritmos familiares
  - ← Búsqueda binaria
  - ← Búsqueda secuencial
- Determinaremos que el tiempo de ejecución de cualquier algoritmo de búsqueda que depende en comparaciones es en el peor caso $\Omega(\log n)$

Búsqueda Secuencial

```
void BusquedaSecuencial( int A[], int n, int x )
{
   int  i;
   for( i=0; i < n; i++ )
     if( A[i] == x )
       return i;
   return -1;
 }
```

En el peor caso $O(n)$



El arreglo A puede estar en cualquier orden

Búsqueda Binaria

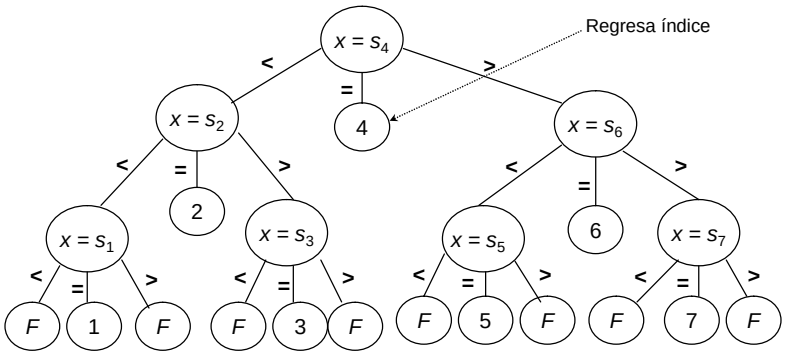
```
void BusquedaBinaria( int A[], int i, int j, int
x)
{
   int k;
   if( i <= j ) {
     k = (i + j) / 2;
     if( A[k] > x )
       return BusquedaBinaria( A, i, k-1, x );
     elseif( A[k] == x )
       return k;
     else /*if ( A[k] < x ) */
       return BusquedaBinaria( A, k+1, j, x );
   } else
     return -1;
}
```

El arreglo A debe estar ordenado

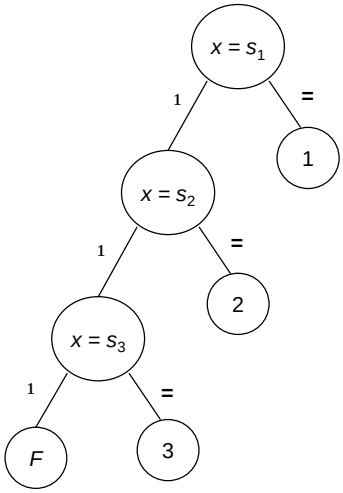
$$T(1) = c$$
$$T(n) = T(n/2) + d$$

Arbol de Decisión para la B. Binaria

Sea $S[1] = s_1, \ldots, S[7] = s_7$



Arbol de Decisión para la B. Secuencial



Arbol de Decisión para la Búsqueda

- Cada hoja representa un punto en el cual el algoritmo se detiene y reporta un índice i donde $x = s_i$ (x es la llave buscada) o reporta una falla
- Un árbol de decisión es ***válido*** para buscar una llave x sobre n llaves si para cada reporte con éxito, existe un camino de la raíz a la hoja que reporta el éxito
- El árbol de decisión es ***podado*** si cada hoja es alcanzable
- Cada algoritmo que busca una llave x en un arreglo de n llaves tiene un árbol de decisión válido podado correspondiente

Profundidad de un Arbol Binario

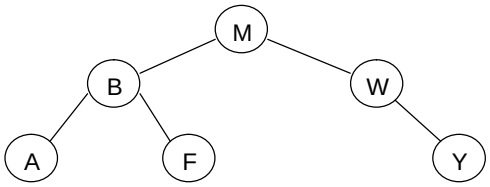
- Recordatorio:
  - Si n es el número de nodos en un árbol binario y d es la profundidad, entonces, $\lfloor \log n \rfloor \leq d$
- Para que un árbol de decisión válido para buscar una llave x sobre n llaves distintas sea podado, el árbol binario debe consistir de al menos n nodos
- Cualquier algoritmo determinista que busca mediante comparaciones una llave x en un arreglo de n llaves distintas debe hacer en el peor caso al menos $\lfloor \log n \rfloor + 1$ comparaciones de llaves

Conjuntos Dinámicos

- Muchas aplicaciones requieren inserciones y supresiones de registros durante la ejecución.
  - Las búsquedas realizadas sobre estos conjuntos se conocen como ***búsqueda dinámica***
  - Ej: Sistemas de reservaciones para aerolíneas
- Algunas estructuras de datos no son adecuadas para conjunto dinámicos
  - Las listas ligadas no permiten hacer búsquedas rápidas
  - Los arreglos son de tamaño fijo

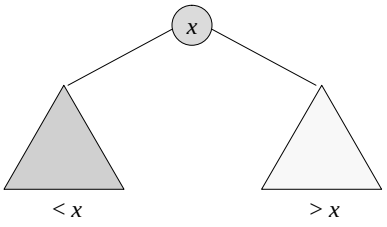
Arbol de Búsqueda Binaria

- El ***árbol de búsqueda binaria*** es una estructura de datos importante para conjuntos dinámicos
  - Un árbol de búsqueda binaria es un árbol binario en donde cada nodo es un objeto el cual contiene cuatro campos: hijo izquierdo, hijo derecho, padre y valor



Arboles de Búsqueda Binaria

- Un árbol de búsqueda binaria debe satisfacer la siguientes propiedades:
  - Si y está en el subárbol izquierdo de x, entonces, llave[y] < llave[x]
  - Si y está en el subárbol derecho de x, entonces, llave[y] > llave[x]



Búsqueda

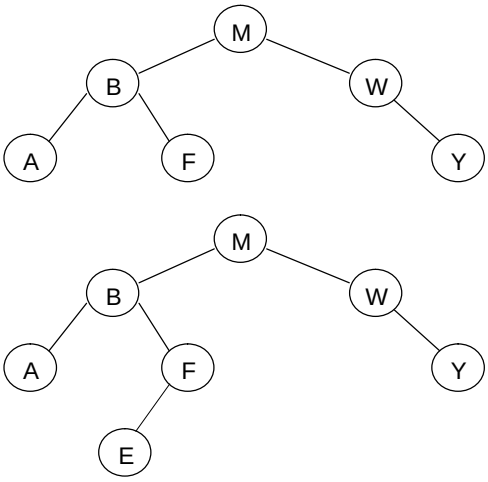
```
Búsqueda( x, k )

begin
    if x = nil or k = llave[x] then
        return x;
    if( k < llave[x] ) then
        Búsqueda( left[x] )
    else
        Búsqueda( right[x] )
end;
```


Recorrido EnOrden

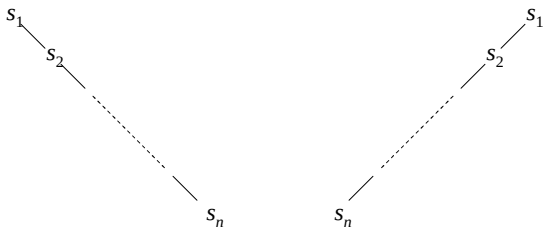
```
RecorridoEnOrden( x )
begin
    if  x <> nil then begin
        RecorridoEnOrden( left[x] );
        writeln( x );
        RecorridoEnOrden( right[x] )
    end
end;
```

Inserción




```
InsertaArbol( T, z )
begin
   y ¬ Nil
   x ¬ root[ T ]
   while x ¹ NIL do begin
     y ¬ x;
     if key[z] < key [x ] then
        x ¬ left [ x ]
     else
        x ¬ right [ x ]
   end;
   p [z] ¬ y;
   if y = Nil then
      root[ T ] ¬ z
   else if key[ z ] < key[ y ] then
      left [ y ] ¬ z
   else
      right [ y ] ¬ z
end;
```


Búsqueda: Peor Caso



$s_1 < s_2 < \ldots < s_n$ $\quad$ $s_1 > s_2 > \ldots > s_n$

☞ La construcción del árbol toma

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2} \qquad O(n^2)$$

Búsqueda: Peor Caso

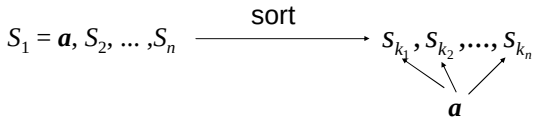
- Suponiendo que se ha construido el árbol con los n elementos,
- Preguntar por la membresía de un elemento tomaría.
  - Mejor caso**:** **1 paso $O(1)$**
  - Peor caso: **n pasos $O(n)$**
  - Caso promedio**:** **$n/2$ pasos $O(n)$**

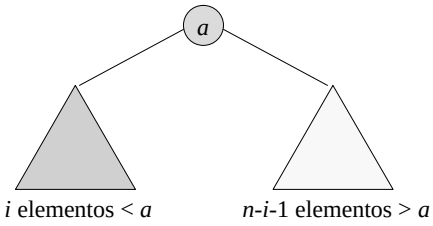
Búsqueda: Caso Promedio

- Es necesario considerar el promedio de todos los posibles árboles que se pueden construir.
- Observaciones
  - Las operaciones que preguntan por la membresía de, insertan o remueven un elemento toman un tiempo proporcional a la longitud del camino de la raíz al nodo correspondiente.
  - Así que se tratará de analizar la longitud de camino promedio en árboles aleatorios.
  - Suposiciones
    - Los árboles se forman únicamente por inserciones
    - Todos los ordenamientos posibles de los n elementos insertados son igualmente probables

Búsqueda: Caso Promedio

- Sea $P(n)$ el número de nodos en promedio en el camino de la raíz a cualquier nodo en un árbol con n nodos
  - $P(0) = 0$, $P(1) = 1$





Búsqueda: Caso Promedio

- Observaciones
  - La longitud de camino promedio en el subárbol izquierdo es $P(i)$
  - La longitud de camino promedio en el subárbol derecho es $P(n\text{-}i\text{-}1)$
  - Longitud promedio es

$$\frac{i\cdot\big(P(i)+1\big)+(n-i-1)(P(n-i-1)+1)+1}{n} \qquad n\geq 2$$

$$P(n)=\frac{1}{n}\sum_{i=0}^{n-1}\left[\frac{i\cdot P(i)+(n\text{-}i\text{-}1)\cdot P(n\text{-}i\text{-}1)}{n}+1\right] \quad , \qquad n\geq 2$$

$$P(n)=1+\frac{1}{n^2}\sum_{i=0}^{n-1}\big[i\cdot P(i)+(n-i-1)\cdot P(n-i-1)\big]=1+\frac{2}{n^2}\sum_{i=0}^{n-1}i\cdot P(i) \quad , \quad n\geq 2$$

Se puede demostrar que $P(n)\leq 1+4\log_2 n$

Búsqueda: Caso Promedio

- Demostración por inducción
  - Para $n = 1$

$$P(1) = 1 \le 1 + 4\log_2 1$$

  - Supongamos que es cierta para todos los $i < n$

$$\begin{aligned} P(n) = 1 + \frac{2}{n^2}\sum_{i=0}^{n-1} i \cdot P(i) &\le 1 + \frac{2}{n^2}\sum_{i=0}^{n-1} i \cdot [1 + 4\log_2 i] \\ &= 1 + \frac{2}{n^2}\left[\sum_{i=0}^{n-1} 4i\log_2 i + \sum_{i=0}^{n-1} i\right] \\ &= 1 + \frac{2}{n^2}\left[\sum_{i=0}^{n-1} 4i\log_2 i + \frac{(n-1)n}{2}\right] = 1 + \frac{8}{n^2}\sum_{i=0}^{n-1} i\log i + \frac{2}{n^2}\left[\frac{n^2 - n}{2}\right] \\ &\le 2 + \frac{8}{n^2}\sum_{i=0}^{n-1} i \cdot \log_2 i \end{aligned}$$

Búsqueda: Caso Promedio

Para todo $i \le \lfloor n/2 \rfloor$ se tiene que

$$i\log_2 i \le i\log_2 \frac{n}{2} = i[\log_2 n - 1]$$

Para todo $n > i > \lfloor n/2 \rfloor$ se tiene que

$$i\log_2 i < i\log_2 n$$

$$\begin{aligned} P(n) &\le 2 + \frac{8}{n^2}\left[\sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} i \cdot \log_2 i + \sum_{i=\lfloor \frac{n}{2} \rfloor + 1}^{n-1} i \cdot \log_2 i\right] \\ &\le 2 + \frac{8}{n^2}\left[\log_2 n \sum_{i=0}^{n-1} i - \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} i\right] \le 2 + \frac{8}{n^2}\left[\log_2 n \sum_{i=0}^{n-1} i - \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} i\right] \\ P(n) &\le 2 + \frac{8}{n^2}\left[\frac{n(n-1)}{2}\log_2 n - \frac{(\frac{n}{2})(\frac{n}{2}+1)}{2}\right] = 2 + \frac{8}{n^2}\left[\frac{n^2}{2}\log_2 n - \frac{n}{2}\log_2 n - \frac{n^2}{8} - \frac{n}{4}\right] \\ &< 2 + \frac{8}{n^2}\left[\frac{n^2}{2}\log_2 n - \frac{n^2}{8}\right] = 1 + 4\log_{2n} \end{aligned}$$

Búsqueda: Caso Promedio

Dado que $P(n) \leq 1 + 4\log_2 n \quad , \quad n \geq 0$

se puede probar que $P(n) \leq c \log_2 n$

para alguna constante $c > 0$

Por lo tanto, se dice que $P(n)$ es un $O(\log_2 n)$

Este resultado es válido para el promedio de la operación de búsqueda sobre todas las entradas con n llaves

¿Cómo mejorar las cosas?

- Los resultados se transforman a cotas para el peor caso cuando el árbol binario está balanceado

- Otros enfoques:
  - Arboles AVL
  - Arboles rojinegros
  - Arboles B: una clase de árboles.
    - Arboles 2-3: un caso especial
  - Tablas de hash
    - No adecuadas para el problema de ordenamiento.
    - Con una buena función de dispersión, la inserción, supresión y búsqueda toma un tiempo casi constante

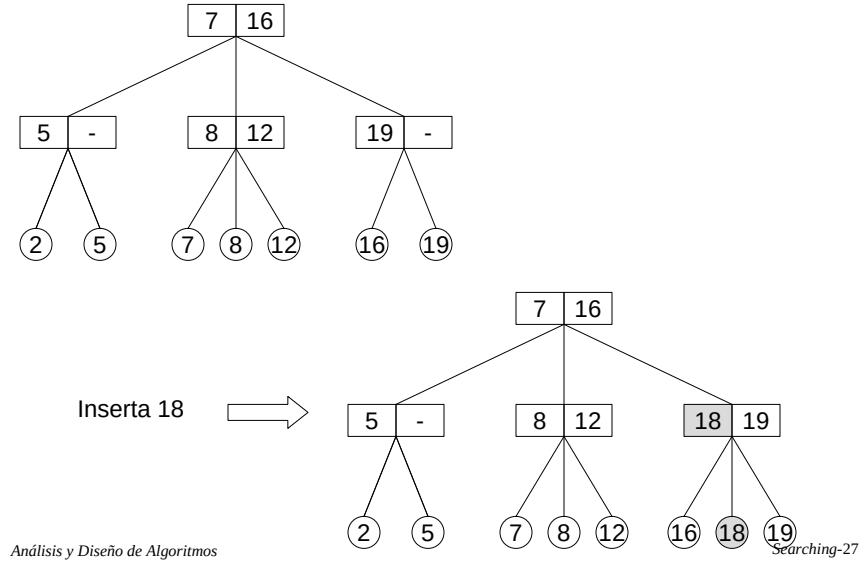
Arboles 2-3

- Arbol 2-3
  - Cada nodo interior tiene dos o tres hijos.
  - Los caminos de la raíz a las hojas tienen todos la misma longitud.
  - Un árbol con cero nodos o un nodo es un caso especial de un árbol 2-3.
  - Un árbol 2-3 se utiliza para representar conjuntos cuyos elementos mantienen un orden lineal , '<'.
  - Todos los elementos se colocan en las hojas; si un elemento a está colocado a la izquierda de b, entonces, se debe satisfacer que $a < b$.

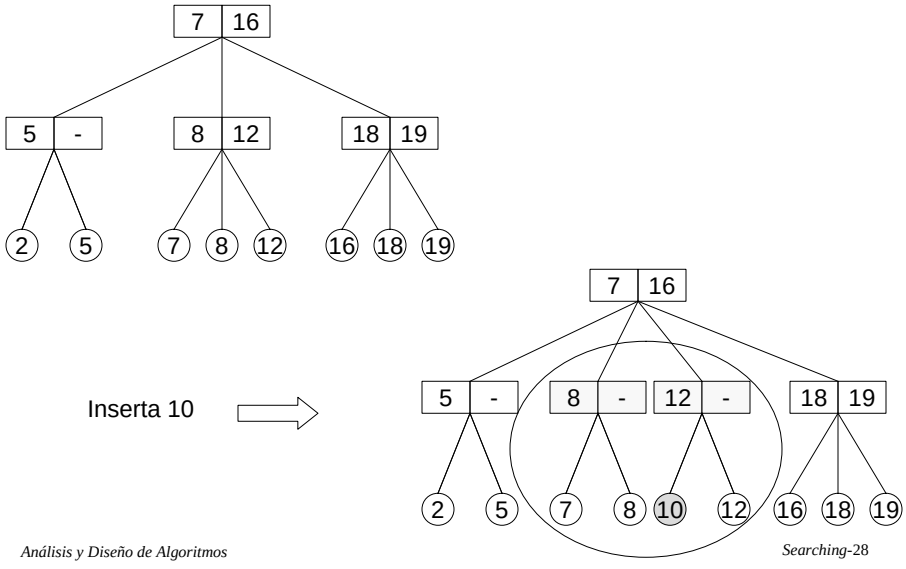
Arboles 2-3

- Arbol 2-3
  - En cada nodo interior se almacena la "llave " del elemento más pequeño que es un descendiente del segundo hijo.
  - Si existe un tercer hijo, se almacena también la "llave" del elemento más pequeño que es un descendiente del tercer hijo.
  - Un árbol 2-3 de k niveles tiene entre 2^{k-1} y 3^{k-1} hojas.
  - Para representar un conjunto de n elementos se requieren al menos $1+\log_3 n$ niveles y no más de $1+\log_2 n$ niveles.

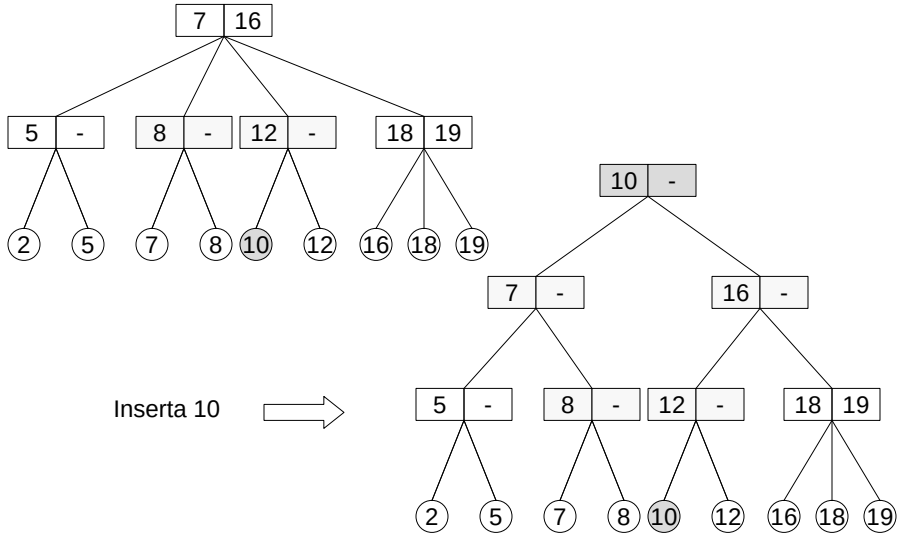
Arboles 2-3: Inserción



Arboles 2-3: Inserción

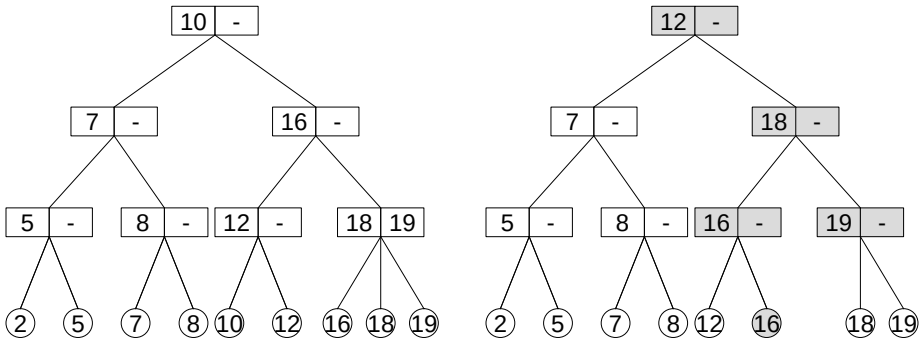


Arboles 2-3: Inserción



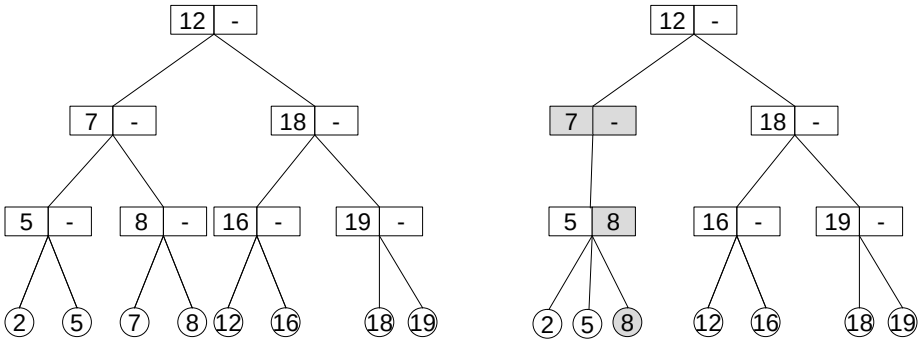
Arboles 2-3: Supresión

Suprime 10



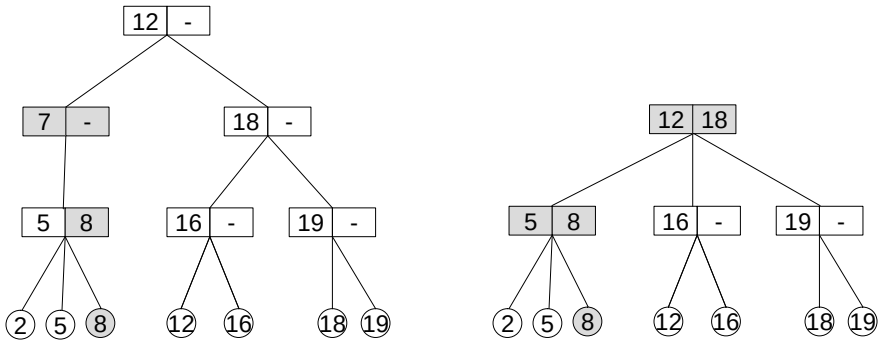
Arboles 2-3: Supresión

Suprime 7



Arboles 2-3: Supresión

Suprime 7



Arboles 2-3: Eficiencia

- Eficiencia
  - Claramente la operación de búsqueda toma un tiempo $\Omega(\log_3 n)$ y $O(\log_2 n)$.
  - Las operaciones de inserción y supresión navegan de la raíz a las hojas y, en el peor caso, de regreso a la raíz. Así, ambas toman también un tiempo $\Omega(\log_3 n)$ y $O(\log_2 n)$.
  - ¿Qué hay acerca del caso promedio?

A2-3: Estructuras

```
struct objeto  {
    int  llave;
    .
   /*Campos adicionales para la
        definición del objeto  */
    .
};


struct nodo interior  {
    int llave1, llave2;
    struct nodo23 *hijo1, *hijo2, *hijo3;
}

struct nodo23 {
    int tipo;   /* interior 0, hoja 1 */
    union {
        struct objeto e;
        struct nodo_interior n;
    };
};
```

A2-3: Agrega1

```
void agrega1( NODO23 *nodo, OBJETO x, NODO23 **nuevo, int *min )
{
    NODO23  *nuevo1;  *w;
    int     min1;
    *nuevo = NULL;
    if( nodo es hoja ) {
        if( x no es el elemento en el nodo ) {
            /* Crear un nuevo nodo apuntado por nuevo */
            /* Copiar x en el nuevo nodo */
            *min = x.llave;
        }
    } else {
        /* Sea w el hijo del nodo a cuyo subárbol pertenece x */
        agrega1( w, x, &nuevo1, &min1);
        if( nuevo1 != NULL )  {
        /*Insertar el apuntador nuevo1 entre los hijos de nodo a la
          derecha de w  */
            if( nodo tiene cuatro hijos ) {
              /* Crear un nuevo nodo apuntado por nuevo  */
              /* Pasar el tercero y cuarto hijo de nodo a nuevo */
              /* Ajustar llave1 y llave2 en nodo y en nuevo */
              /* Hacer min la llave más chica entre los hijos del nuevo nodo */
            }
        }
    }
}
```


A2-3: Agrega

```
void AGREGA( OBJETO x, NODO23 *S )
{
    NODO23   *nuevo;
    int      mínimo;
    NODO23   tmp;

    /*  Verificar si S es vacío o consiste de un solo elemento.
        Incluir aquí un procedimiento de inserción adecuado  */
    agrega1( *S, x, &nuevo, &mínimo );
    if( nuevo != NULL ) {
        /* Se crea un nuevo nodo cuyos hijos serán *S y nuevo */
        tmp = *S;
        *S = ( struct nodo23 *) malloc ( sizeof( struct nodo23 ) );
        (*S)->llave1 = mínimo;
        (*S)->hijo1 = tmp;
        (*S)->hijo2 = nuevo;
        (*S)->hijo3 = NULL;
    }
}
```

A2-3: Borra1

```
int borra1( NODO23 *nodo, OBJETO x )
{
     int solo_uno, borrado;
     NODO23  *w;

     borrado = 0;
     if( los hijos de nodo son hojas ) {
          if( x está entre las hojas ) {
               /* Quitar x  */
               /* Correr las hijos de nodo a la derecha de x una posición a la
                  izquierda */
               if( nodo tiene solo un hijo )
                    borrado = 1;
          }
     } else {
          /* Sea w el hijo de nodo que puede tener a x como descendiente */
           solo_uno = borra1( w, x );
           if( solo_uno ) {
               if( w es el primer hijo de nodo ) {
                    if( y, el segundo hijo de nodo, tiene 3 hijos ) {
                         /* Hacer que el primer hijo de y sea el segundo hijo de w */
                    } else {
                         /* hacer el hijo de w, el primer hijo de y */
                         /*Quitar w de entre los hijos de nodo  */
                         if( nodo tiene ahora un hijo )
                             borrado = 1;
                    }
```


A2-3: Borra1

```
               . . .
               } else if ( w es el segundo hijo de nodo ) {
                    if( y, el primer hijo de nodo, tiene 3 hijos ) {
                         /* Hacer que el tercer hijo de y sea el primer hijo de w*/
                    } else {
                         if( z, el tercer hijo de nodo , existe y tiene 3 hijos ) {
                              /*Hacer que el primer hijo de z sea el segundo hijo de w  */
                         } else {
                              /*Hacer que el hijo de w sea el tercer hijo de y */
                              /*Quitar w de entre los hijos de nodo */
                              if( nodo tiene ahora un solo hijo )
                                   borrado = 1
                         }
                    }
               } else
                    if( w es el tercer hijo del nodo ) {
                         if( y, el segundo hijo de nodo, tiene 3 hijos ) {
                              /* Hacer que el tercer hijo de y, sea el  primer hijo de w  */
                         } else {
                              /*Hacer que el hijo de w sea el tercer hijo de y */
                              /* Quitar w de entre los hijos del nodo  */
                         }
                    }
          }
     }
     return borrado;
}
```