Algoritmos y Estructuras de Datos - 2020
Unidad 5 - Árboles

Tabla de Contenidos

Descripción

Contenidos Básicos:

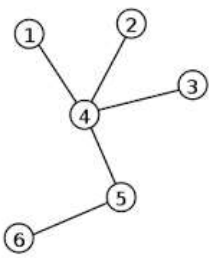
- Definiciones y ejemplos.
- Implementación de árboles con asignación secuencial y encadenada.
- Construcción del árbol.
- Árboles binarios.
- Árboles binarios de búsqueda.
- Árboles binarios de búsqueda equilibrados.
- Árboles rojo-negro.
- Árboles auto balanceados (AVL).
- Árboles B y B+.
- Algoritmos recursivos para recorrer árboles.
- Pre-orden, in-orden y pos-orden.
- Adición y deleción de nodos.

Árboles

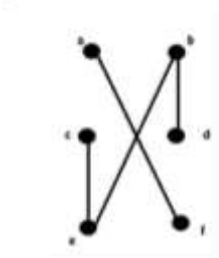
Aquí describiremos construcciones utilizadas para representar datos en forma de árboles. Comparados con las pilas, filas y listas, que son estructuras lineales, los árboles son estructuras no lineales.
En las listas enlazadas, cada nodo apunta al nodo siguiente y/o al anterior. En los árboles, un nodo puede apuntar a más de un nodo adicional.

Definición

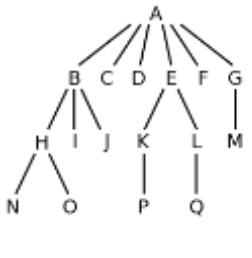
Un árbol es una colección de nodos conectados entre sí, donde para cada par de nodos hay un solo camino que los conecta. No puede contener ciclos, ni aristas múltiples.



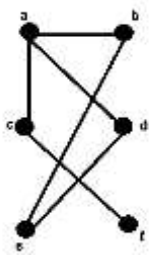
Árbol



No Árbol

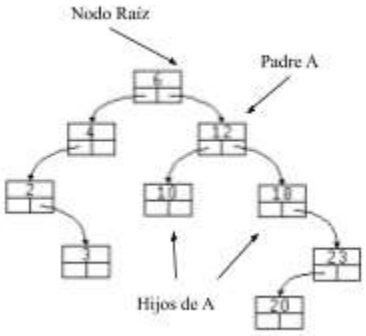


Árbol



No Árbol

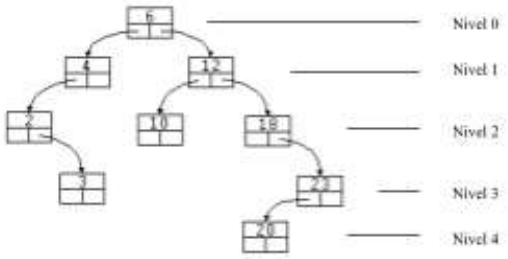
Usualmente se elige un nodo, llamado nodo **Raíz**, considerado el primer elemento del árbol. Cada nodo tiene nodos **Hijos** a los que apunta. Este nodo se llama **Padre** (parent).



Ejemplo de nodo Raíz, Pariente y sus Descendientes

Propiedades

- El nodo raíz es el único nodo que no tiene padres.
- Todos los demás nodos tienen un solo padre cada uno.
- Existen nodos que no tienen hijos, son llamados nodos **hoja**.
- Dos nodos se dicen **hermanos** si tienen el mismo padre.
- Un nodo es **antecesor** de otro nodo si es padre de ese nodo o padre de un antecesor de dicho nodo.
- El nodo raíz es el antecesor de todos los otros nodos.
- Un nodo es **descendiente** de otro nodo si es hijo de dicho nodo o hijo de un descendiente.
- La **profundidad** del árbol es la longitud del camino más largo entre la raíz y un nodo cualquiera.
- El nodo raíz se dice que está a **Nivel** 0. El nivel de cualquier otro nodo es el nivel de su nodo padre más uno.
- El **grado** de un nodo es la cantidad de hijos de dicho nodo
- El **grado** de un árbol es el mayor grado de sus nodos.
- Un nodo define un **sub-árbol** compuesto por dicho nodo y todos los nodos descendientes.
- Un árbol se llama **binario** si cada nodo puede tener como máximo dos hijos.
- En los árboles binarios se etiquetan como **izquierdo** y **derecho** a sus dos hijos.



Ejemplo de árbol binario de profundidad 4, mostrando los diferentes niveles

Implementación

Con Arreglos

Un árbol binario se puede representar en forma secuencial, considerando que en cada nivel *n* puede haber como máximo 2^n nodos.

- Un árbol de profundidad cero, tendrá $2^0 = 1$ nodo (el nodo raíz).
- Un árbol de profundidad uno, tendrá como máximo 2^0 (nivel 0) + 2^1 (nivel 1) nodos, lo que hace 1+2 = 3 nodos.
- Un árbol de profundidad dos, tendrá como máximo 2^0 (nivel 0) + 2^1 (nivel 1) + 2^2 (nivel 2) nodos, lo que hace 1+2+4 = 7 nodos.

- Un árbol de cuatro niveles, tendrá como máximo 2^0 (nivel 0) + 2^1 (nivel 1) + 2^2 (nivel 2) + 2^3 (nivel 3) nodos, lo que hace 1+2+4+8 = 15 nodos.

En general, un árbol de profundidad N, tendrá como máximo $2^0+2^1+ \ldots + 2^N = 2^{N+1}-1$ nodos.
Por lo tanto, se puede utilizar un arreglo de tamaño $2^{N+1}-1$ para guardar la información de los nodos.

Con Listas Enlazadas

Un árbol se puede implementar por medio de estructuras que representan los nodos. Dichas estructuras deben contener la siguiente información:

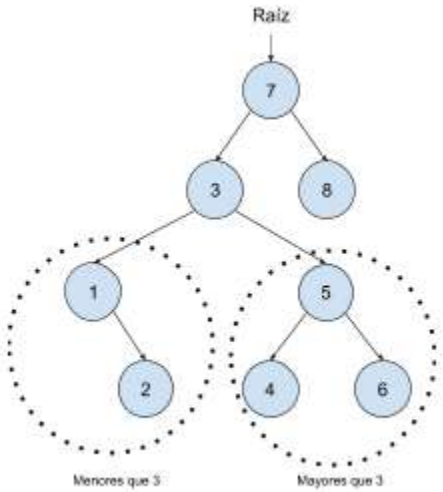
a) El valor guardado en dicho nodo
b) Punteros a los hijos de dicho nodo.

En el caso de árboles binarios, es suficiente el uso de dos punteros. La siguiente estructura es suficiente para representar un nodo (el nombre **btnodo** viene de "**b**inary **t**ree **nodo**"):

```
struct btnodo
 {
   int valor;
   struct btnodo *left;
   struct btnodo *right;
};
```

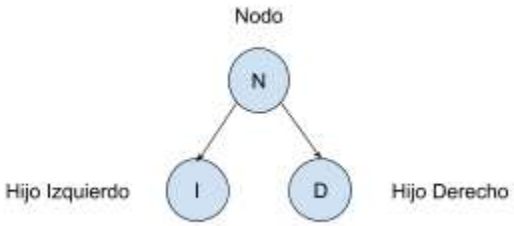
Árboles de Búsqueda Binarios

Un árbol binario es de **búsqueda** si cumple con la siguiente propiedad: para todos los nodos, su valor es mayor que el de todos los nodos descendientes de su hijo izquierdo y menor que el de todos los nodos descendientes de su hijo derecho.



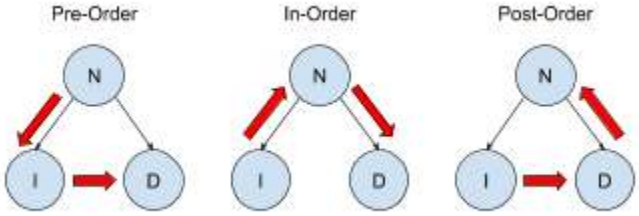
Recorrido

Recorrer un árbol binario consiste en visitar una vez cada nodo. Estando en un nodo, hay tres posibles acciones: a) visitar el nodo, b) visitar el hijo izquierdo, c) visitar el hijo derecho



El orden en que se visitan estos nodos define el recorrido del árbol. Las opciones son:

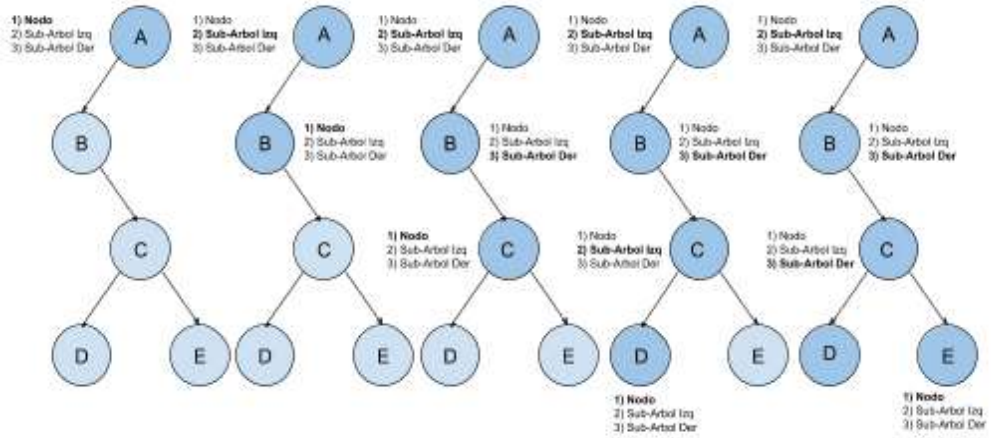
a) **Pre-order** - Visitar primero el nodo y después los sub-árboles - NID
b) **In-order** - Visitar un sub-árbol, después el nodo y después el otro sub-árbol - IND
c) **Post-order** - Visitar los dos sub-árboles y después el nodo - IDN



Recorrer un árbol consiste en comenzar por el nodo raíz y visitar el resto de los nodos utilizando una de estas tres reglas.

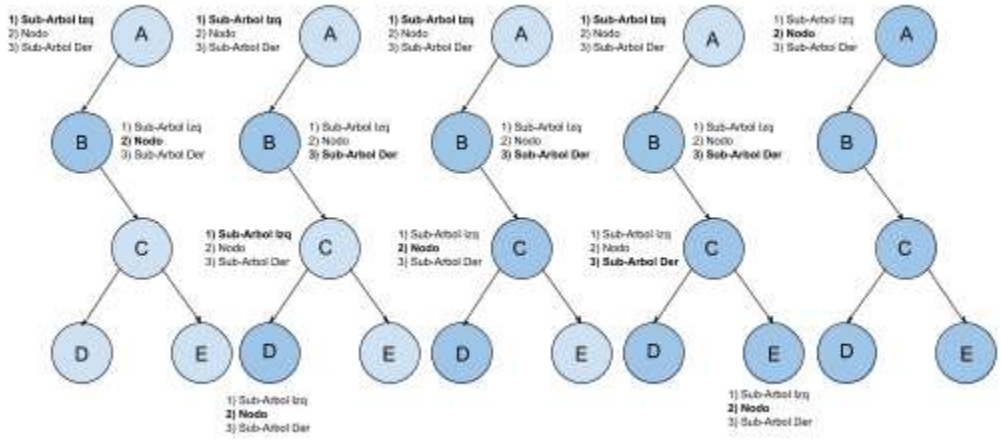
Dado que un árbol binario es recursivo, ya que cada sub-árbol es un árbol en sí mismo, se permite pensar el proceso como un proceso recursivo.

El siguiente ejemplo muestra el recorrido de un árbol en pre-order:



Ejemplo de recorrido pre-order del árbol. Secuencia final: ABCDE.

En el caso de in-order, el recorrido es el siguiente:



Ejemplo de recorrido in-order del árbol. Secuencia final: BDCEA.

El código recursivo para el recorrido pre-order y in-order es el siguiente:

```
void preorder (struct btreenode *p)
 {
  if ( p != null)
   {
    printf("%d", p->info);
    preorder(p->left);
    preorder(p->right);
   }
 }
```

pre-order

```
void inorder(struct btreenode *p)
 {
  if (p != null)
   {
    inorder(p->left);
    printf("%d", p->info);
    inorder(p->right);
   }
 }
```

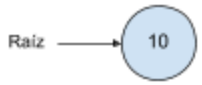
in-order

Nota: es importante recordar que el árbol de búsqueda binario tiene la característica que el valor de un nodo es mayor que el valor de todos los nodos a su izquierda (descendientes a izquierda) y menor que el de todos los nodos a su derecha. Por lo tanto, el recorrido in-order genera siempre un listado ordenado de sus valores.

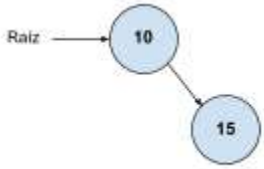
Inserción

Un aspecto importante de un árbol de búsqueda binario es su construcción. Al agregar un nodo, se tiene que considerar que el árbol debe mantener siempre sus propiedades de árbol de búsqueda binario.
A continuación, se muestra un ejemplo: se quiere crear un árbol de búsqueda binario con los siguientes valores: 10, 15, 12, 7, 8, 18, 6, 20.

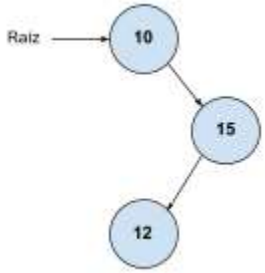
a) El primer paso es el de inicializar el árbol. Se logra definiendo el puntero **root** como nulo.
b) El segundo paso consiste en agregar el primer nodo (10). Como el nodo raíz es nulo, este nodo pasa a ocupar el lugar de nodo raíz.



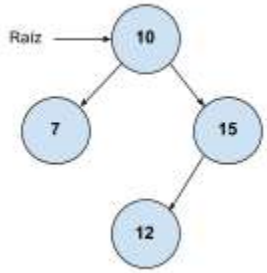
c) Luego se agrega el 15. Como es mayor que 10, debe ir a su derecha.



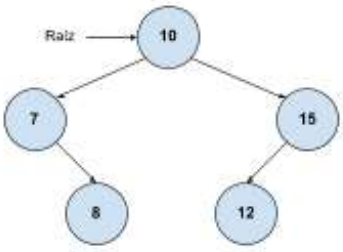
d) Luego se agrega el 12. Como es mayor que 10, debe ir a su derecha. A la derecha del 10 hay un sub-árbol con raíz 15. Como 12 es menor que 15, debe ir a su izquierda. Como está vacío (la izquierda de 15), se agrega ahí.

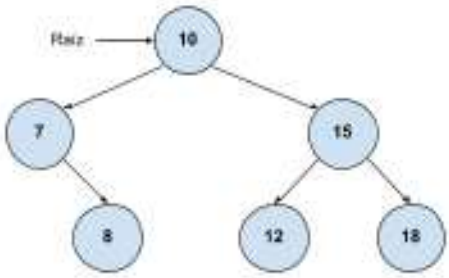


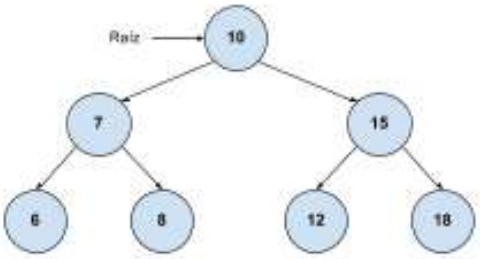
e) Luego se agrega el 7. Como es menor que 10, debe ir a su izquierda. Como está vacío (la izquierda de 10), se agrega ahí.

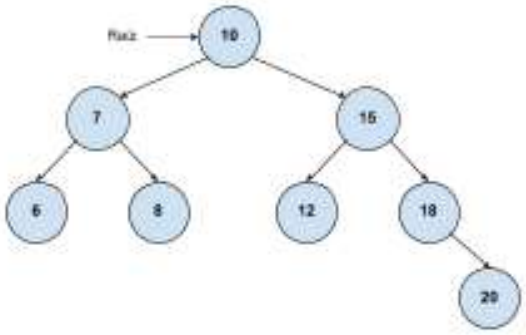


f) De la misma forma se insertan el 8, el 18, el 6 y el 20.



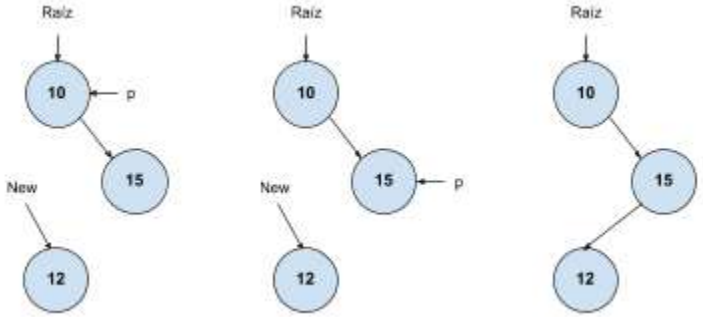






Consideraciones:

a) Para insertar un nodo, primero se debe buscar el lugar donde debe ser insertado.
b) Para cada valor nuevo debe crearse un nodo nuevo y luego ajustar los punteros.
c) Un puntero "p" se usará para recorrer los nodos hasta encontrar un NULL (que es donde se debe insertar el nuevo nodo).
d) Es importante notar que el nuevo nodo siempre se insertará en una posición donde no tendrá, en ese momento, sub-nodos.



Las tres etapas: a) Se crea un nodo nuevo con valor 12. El puntero p apunta al inicio del árbol. Como 12>10 y el 'hijo derecho no es NULL, avanza al próximo nodo (15). b) El puntero apunta a 15. Como 12<15, se observa el sub-arbol izquierdo, que es NULL. Por lo tanto, es el lugar a insertar. c) Se inserta el 12 a la izquierda de 15.

El siguiente código muestra el proceso no recursivo de inserción:

```
tree insert(tree *s,int x)
{
    tree *trail, *p, *q;
    q = (struct tree *) malloc (sizeof(tree));
    q->info = x;
    q->left = null;    // El nuevo nodo no tendrá sub-árboles
    q->right = null;   // El nuevo nodo no tendrá sub-árboles
    p = s;             // Inicializa el puntero a la raíz del árbol
    trail = null;      // nodo previo (donde insertar)
    while (p != null)  // recorre el árbol hasta que no hay más sub-arboles
    {
        trail = p;          // actualiza el "anterior"
        if (x < p->info)    // determina si sigue buscando a derecha o izquierda
            p = p->left;
```

```
        else
            p = p->right;
    }
    // En este punto p=NULL y trail=Nodo donde insertar
    // Caso especial de árbol vacío (trail==NULL)
    if (trail == null) {
        s = q;            // El árbol va a tener un solo nodo, el nuevo
        return (s);
    }
    if(x < trail->info)   // Determina si inserta a izquierda o derecha
        trail->left = q;  // Inserta a izquierda
    else
        trail->right = q; // Inserta a derecha
    return (s);
}
```

El proceso consiste, para cada sub-árbol, en comparar si el nodo es mayor o menor que el nodo y procesar el sub-árbol de la izquierda o derecha, en función del resultado de la comparación. Esto da una idea de recursión:

a) Caso base: el árbol está vacío.
b) Caso recursivo: cada sub-árbol es menor que el árbol original (al menos el nodo raíz no está).

El siguiente código muestra el proceso recursivo de inserción:

```
   tree rinsert (tree *s,int x)
   {
       /* insertion into an empty tree; a special case of insertion */
       if (!s)
       {
           s=(struct tree*) malloc (sizeof(struct tree));
           s->info = x;
           s->left = null;
           s->right = null;
           return (s);
       }
       if (x < s->info)
           s->left = rinsert(x, s->left);
       else
           if (x > s->info)
               s->right = rinsert(x, s->right);
       return (s);
   }
```

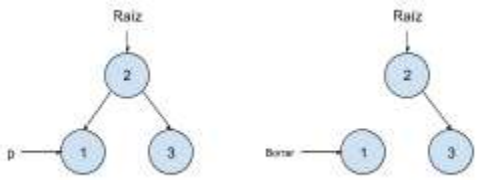
Consideraciones:

a) Si los datos se ingresan ordenados de menor a mayor, el árbol crecerá sólo hacia la derecha (cada valor nuevo va a la derecha del máximo valor en el árbol).
b) De la misma manera, si se ingresan de mayor a menor, el árbol crecerá hacia la izquierda.

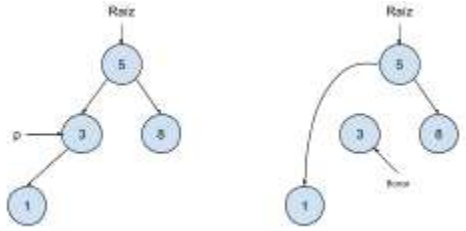
Borrado

El borrado de un nodo del árbol es un proceso más complejo, dependiendo de la posición del nodo:

a) Si el nodo es una hoja (no tiene hijo), es simple removerlo. Se hace nulo el puntero a ese nodo desde el nodo padre.



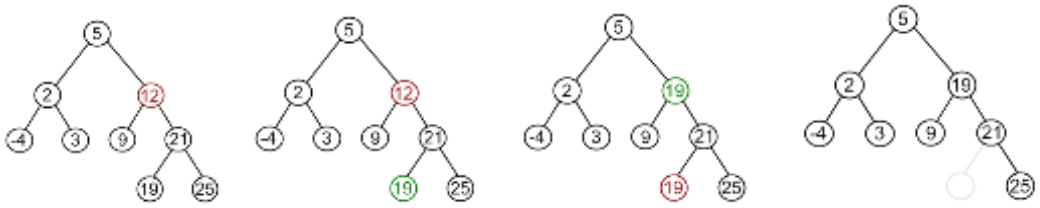
b) Si el nodo tiene un solo hijo, es simple removerlo (el hijo ocupa su lugar).



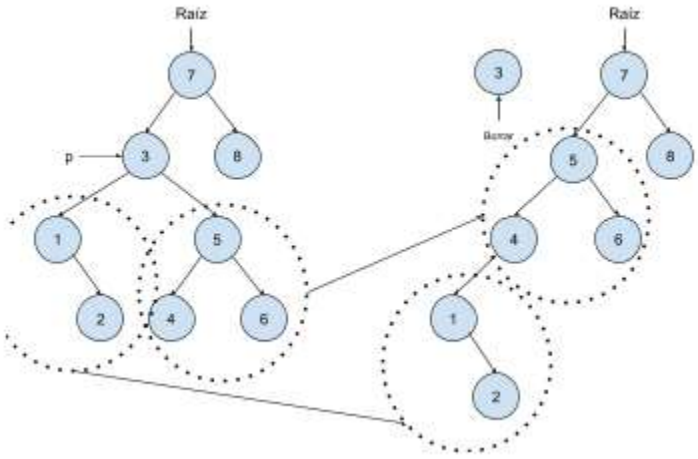
c) Si el nodo tiene dos hijos (con posible sub-árboles más complejos o no), el proceso de borrado de un árbol de búsqueda binario se puede hacer en forma simple de la siguiente manera:

i) Reemplazar su valor por el menor valor de su sub-árbol a derecha o por el mayor valor de su sub-árbol a izquierda.

ii) Eliminar el nodo que se eligió. Tiene un solo hijo, o ninguno, por ser el menor, o el mayor, de un sub-árbol, por lo que puede aplicarse (a) o (b).



d) También se puede hacer el borrado para nodos con dos hijos. El proceso es más complejo. Uno de sus hijos (por ejemplo, el derecho) pasa a ocupar su posición. El otro hijo, el izquierdo (y su sub-árbol) se inserta a la izquierda del menor valor del sub-árbol de la derecha (al ser el menor, su hijo izquierdo es null).



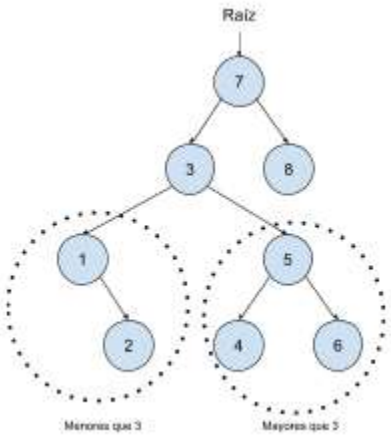
El siguiente código muestra el proceso de eliminación de un nodo:

```
void delete(struct tree *p)
{
    struct tree *temp;
    if (p == null)     // nodo vacío
        printf("Trying to delete a non-existent node");
    else if (p->left == null) // Un solo hijo, a la derecha
    {
        temp=p;
        p = p->right;       // Coloca el hijo de la derecha en lugar del nodo
        free(temp);
    }
    else if (p->right == null) // Un solo hijo, a la izquierda
    {
        temp = p;
        p = p->left;     // Coloca hijo de la izquierda en lugar del nodo
        free (temp);
    }
    else if(p->left != null && p->right!= null) // Dos hijos
    {
        temp = p->right;            // hijo de la derecha
        while (temp->left != null)  // busca el menor valor del arbol a la derecha
        {
            temp = temp->left;
        }
        temp->left = p->left;       // Inserta sub-arbol de la izquierda
        temp = p;
        p = p->right;               // Coloca nodo de la derecha en lugar del nodo
        free (Temp);
    }
}
```

Búsqueda

La búsqueda de un elemento consiste en recorrer el árbol hasta encontrar el elemento buscado. No es necesario recorrer todo el árbol, pues puede utilizarse la información sobre la característica básica de

árboles de búsqueda binario: dado un nodo, todos los valores del sub-árbol izquierdo son menores que su valor y todos los valores del sub-árbol derecho son mayores que su valor.



El proceso de búsqueda es el siguiente:

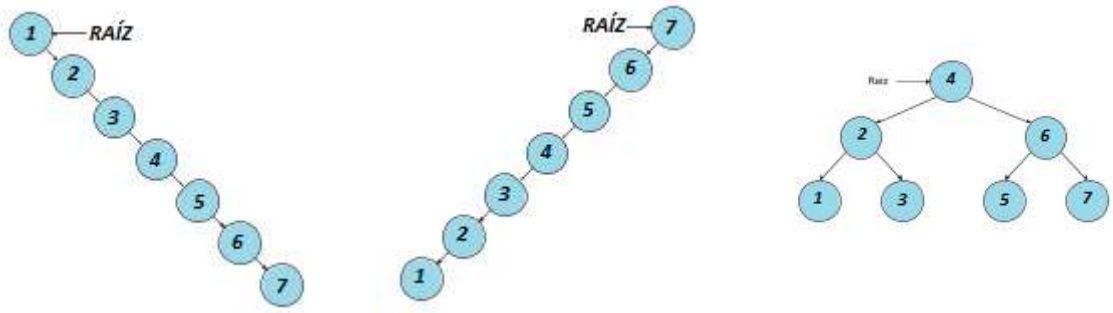
a) Si el valor buscado es igual al valor del nodo, entonces fue encontrado.
b) Si el valor buscado es menor que el valor del nodo, y el nodo tiene un hijo izquierdo, entonces se aplica la búsqueda al nodo hijo izquierdo.
c) Si el valor buscado es mayor que el valor del nodo, y el nodo tiene un hijo derecho, entonces se aplica la búsqueda al nodo hijo derecho.
d) Caso contrario, el valor buscado no está en el árbol.

El siguiente código muestra el proceso de búsqueda no recursiva:

```
search (struct tree *root,int x)
{
    struct tree *p;
    p = root;
    while (p != null && p->info != x)
    {
    if (p->info > x)
        p = p->left;
    else
        p = p->right;
    }
    return (p);
}
```

Árboles binarios de búsqueda equilibrados.

Un árbol binario de búsqueda puede estar muy desbalanceado. Por ejemplo, se puede ver que si se insertan nodos en el orden 1,2,3,4,5,6,7 se obtiene el árbol de la izquierda (a), mientras que si se insertan los nodos en el orden 7,6,5,4,3,2,1 se obtiene el árbol del medio (b) y, finalmente, si se insertan los nodos en el orden 4,2,1,3,6,5,7, se obtiene el árbol de la derecha (c).



(a) (b) (c)

Ejemplo de 3 árboles para los mismos valores dependiendo del orden de inserción de los valores.

Los dos primeros árboles están extremadamente *desbalanceados*, en el sentido que un sub-árbol de la raíz tiene mucha mayor profundidad que el otro. Por ejemplo, en (a) el sub-árbol derecho tiene profundidad 6 y el sub-árbol izquierdo tiene profundidad -1 (árbol vacío). Por lo contrario, en (c), el sub-árbol de la izquierda y el de la derecha tienen la misma profundidad (2).

La desventaja de tener un árbol desbalanceado está en el hecho que una búsqueda requerirá mayor cantidad de comparaciones en algunos casos. Por ejemplo, en el árbol (a) pueden necesitarse hasta 7 comparaciones para encontrar el 7, mientras que en el árbol (c) nunca se necesitarán más de 3 comparaciones.

Algunos de los métodos de balanceo existentes son:

- **AVL tree - Árbol Auto Balanceado - Algoritmo clásico.**
- **Red-black tree - Árbol etiquetado con colores rojo y negro - Algoritmo clásico.**
- Scapegoat tree - Parecido a AVL, con otros algoritmos de balanceo, más flexible.
- Treap - Un tipo de árbol donde se utiliza una clave, o prioridad, aleatoria y se mantiene ordenado los valores y la clave.
- AA tree - Una variante del red-black tree.
- Otros.

Árboles auto balanceados AVL

Los árboles Auto Balanceados (AVL) son árboles binarios que se mantienen siempre balanceados cuando se realizan operaciones de inserción y deleción. Su nombre viene de las iniciales de sus inventores: G.M. **A**del'son-**V**el'skii y E.M. **L**andis.

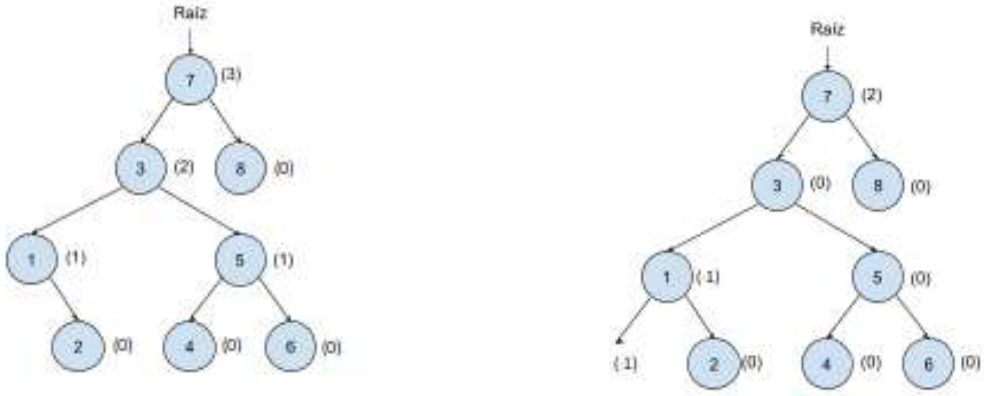
Definiciones

- **Altura de un nodo**. Se define recursivamente:
  a) La altura de un puntero nulo es -1.
  b) La altura de una hoja (nodo con valor, sin nodos hijos con valor) es 0.

c) La altura de un nodo interno es la mayor altura de sus hijos, más uno.

- **Árbol AVL (balanceado).** Un árbol AVL es un árbol binario que cumple con la siguiente propiedad: para todo nodo del árbol, la profundidad de sus hijos derecho e izquierdo difiere como máximo en 1. Un árbol binario vacío es un AVL.
- Sea T^L el ***sub-árbol izquierdo*** de un árbol T.
- Sea T^R el ***sub-árbol derecho*** de un árbol T.
- Sea h(T) la ***altura*** de un árbol
- Sea $h(T^L)-h(T^R)$ el ***factor de balanceo*** de un árbol T.
- Un árbol es ***AVL*** si y sólo si T^L y T^R son AVL y $| h(T^L)-h(T^R) | \leq 1$
- Un árbol es un **árbol de búsqueda AVL** si es un árbol binario de búsqueda y además es AVL.

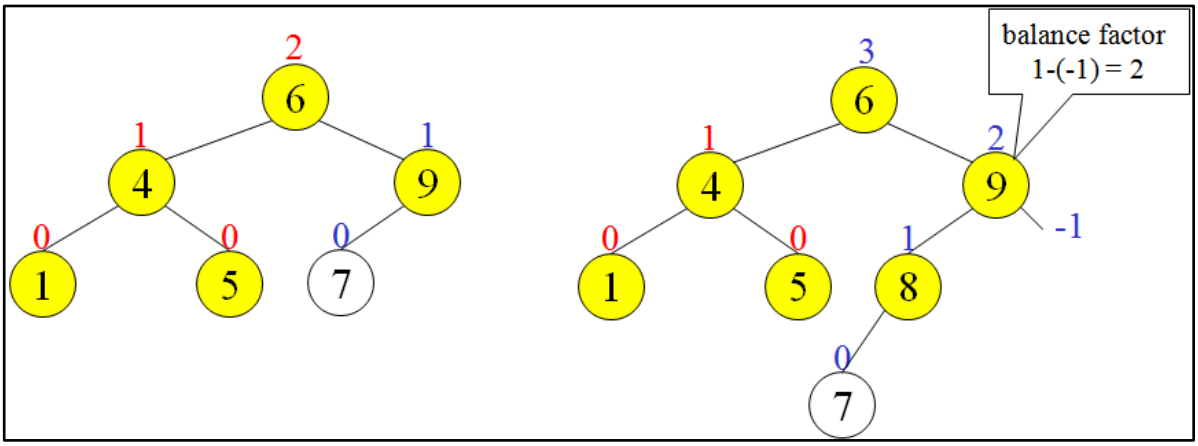
De lo anterior, es claro que un árbol es AVL si su factor de balanceo es -1,0 o 1.



(a) (b)

Ejemplo del cálculo de la altura de cada nodo (a), de abajo hacia arriba, y del cálculo del factor de balanceo de cada nodo (b), que solo dependen de la altura de un nodo y sus dos hijos (un puntero nulo tiene altura cero).

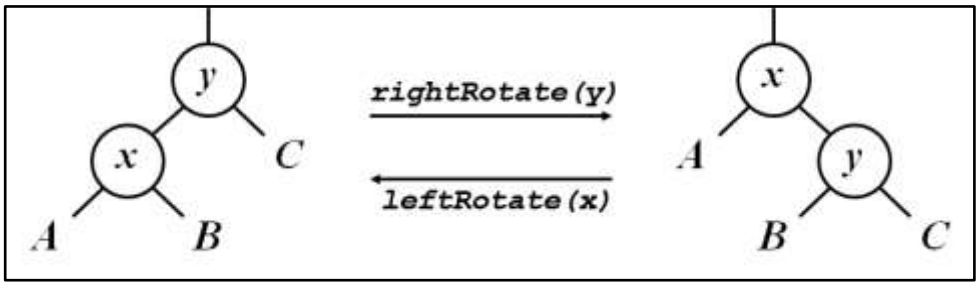
En el siguiente ejemplo, el árbol de la izquierda es AVL y el de la derecha no lo es:



Ejemplo de árbol AVL (izquierda) y árbol no AVL (derecha). Los números indican la altura de los nodos.

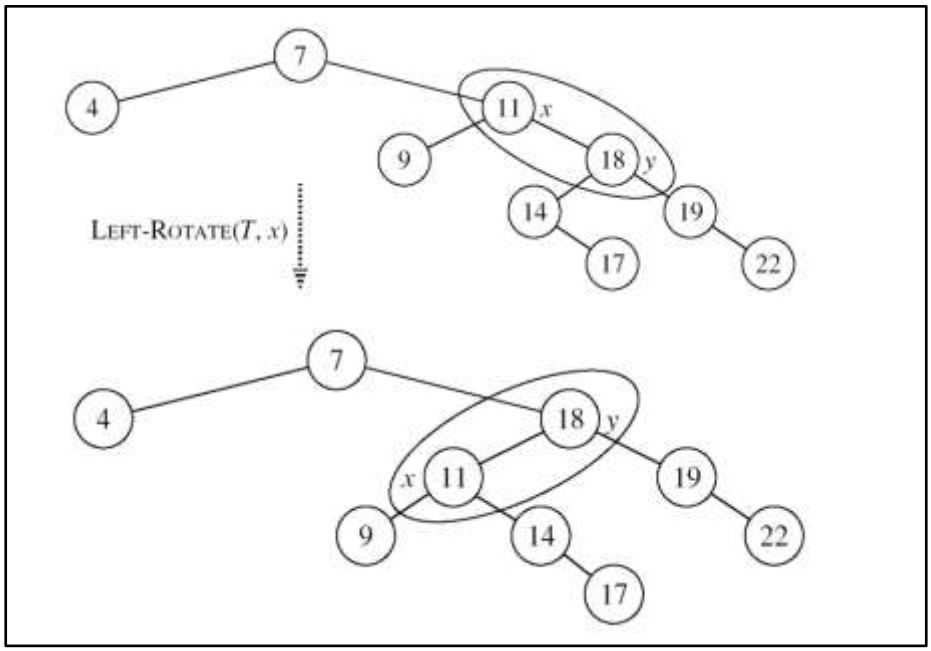
Rotaciones

La clave para reorganizar el árbol son las rotaciones. Las rotaciones preservan el orden de los valores. Aplicando una rotación a un par de nodos, se mantiene la propiedad de ser un BST (Binary Search Tree). Una rotación puede ser a derecha o a izquierda, como muestra el siguiente dibujo, y se aplica siempre a un par padre-hijo:



Rotaciones a derecha y a izquierda. A,B,C indican los sub-árboles de los nodos.

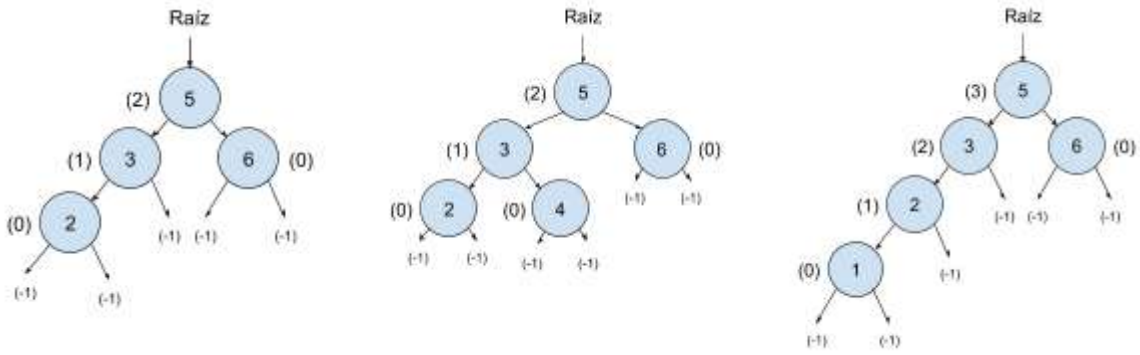
Debido a la propiedad que tienen las rotaciones, de mantener la propiedad de BST, serán la herramienta básica de corrección de árboles AVL cuando, después de una inserción o borrado, se pierda el balanceo del árbol. El siguiente ejemplo muestra el efecto de una rotación a izquierda sobre dos nodos de un árbol:



Efecto de la rotación a izquierda sobre el par de nodos padre-hijo 11 y 18.

Inserción

Como se analizó en el caso de árboles de búsqueda binario, la inserción de un nodo puede afectar el balanceo.



(a) (b) (c)

Ejemplo de inserción en árbol balanceado. El árbol en (a) está balanceado. Al agregar el elemento 4, el árbol continúa balanceado (b), pero si se le agrega, al primer árbol, el elemento 1, se pierde el balanceo (c). Entre paréntesis se muestra la altura de cada nodo.

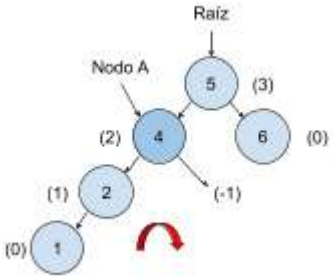
Para solucionar este problema, al insertar un nodo nuevo, puede ser necesario una *corrección* del árbol para mantenerlo balanceado. Esto se puede realizar por medio de rotaciones.

(a) (b)

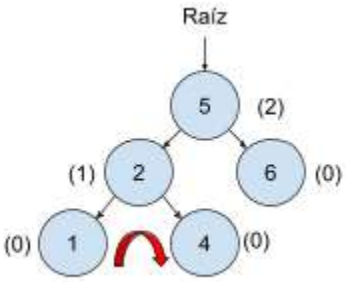
Ejemplo de rotación de los nodos 2,4 para mantener el árbol balanceado, después de la inserción. a) el árbol está desbalanceado después de la inserción del nodo 1 (el nodo 5 tienen un factor de balanceo 2-0=2). b) el árbol queda balanceado después de la rotación. En paréntesis vemos la altura de cada nodo.

La corrección se hace de la siguiente manera:

a) Se identifica un nodo cuyo factor de balanceo no es -1,0,1 y que es el antecesor más cercano al nodo insertado. Se llama nodo A.
b) Se realizan las siguientes correcciones dependiendo del caso:
    i) **Caso 1** - Situación LL (Left-Left) si el nodo nuevo está en el sub-árbol a la izquierda del sub-árbol a la izquierda de A.
    En el ejemplo se observa que A está desbalanceado (su hijo derecho, nulo, tiene profundidad -1.) Se aplica una rotación a la derecha de A y su hijo izquierdo (4 y 2).

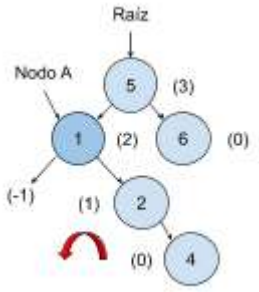


Rotación a derecha

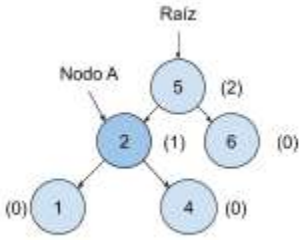


Árbol balanceado

ii) **Caso 2** - Situación RR (Right-Right) si el nodo nuevo está en el sub-árbol a la derecha del sub-árbol a la derecha de A. Es similar al caso 1. Se aplica una rotación a la izquierda de A y su hijo derecho (1 y 2).

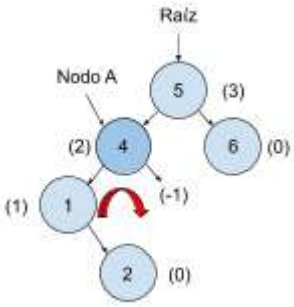


Rotación a Izquierda

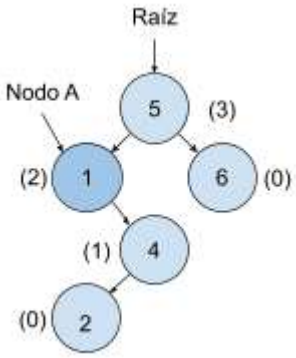


Árbol Balanceado

iii) **Caso 3** - Situación LR (Left-Right) si el nodo nuevo está en el sub-árbol a la derecha del sub-árbol a la izquierda de A. Se podría intentar una corrección con una sola rotación, como en el caso 1, pero el árbol continúa desbalanceado después de la rotación.



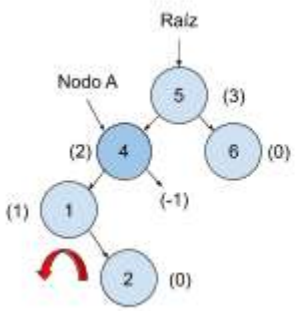
Rotación a Izquierda



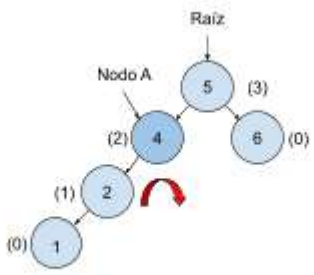
El árbol resultante sigue estando

desbalanceado

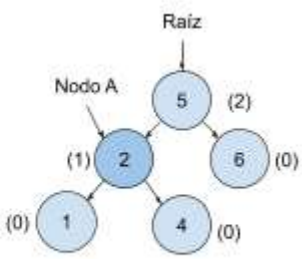
Por lo tanto, se precisa una corrección un poco más compleja. Se aplica primero una rotación a derecha del hijo a izquierda de A y el nieto a derecha del hijo a izquierda. Luego se aplica una rotación a derecha de A y su nuevo hijo izquierdo.



Rotación a Izquierda

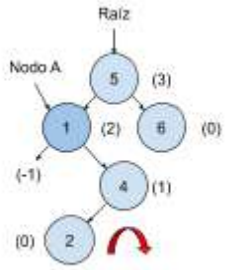


Rotación a Derecha

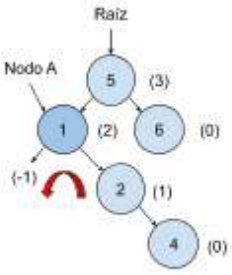


Arbol Balanceado

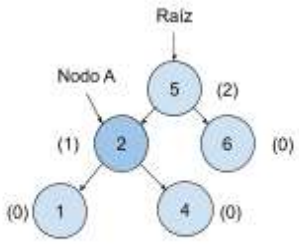
iv) **Caso 4** - Situación RL (Right-Left) si el nodo nuevo está en el sub-árbol a la izquierda del sub-árbol a la derecha de A. Se aplica primero una rotación a izquierda del hijo a derecha de A y el nieto a izquierda del hijo a derecha. Luego se aplica una rotación a izquierda de A y su nuevo hijo derecho.



Rotación a derecha



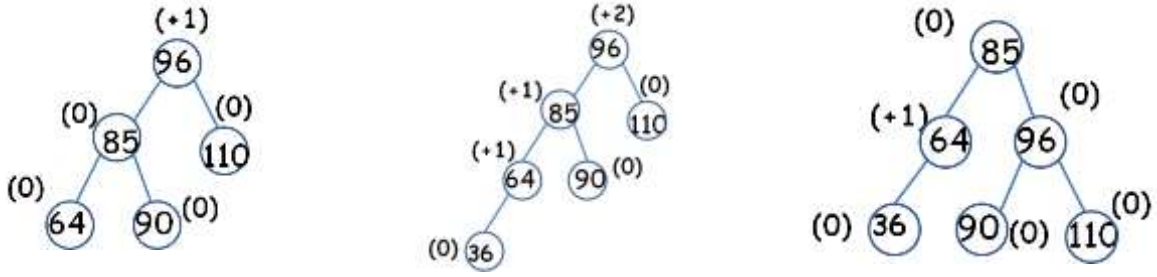
Rotación a izquierda



Arbol Balanceado

En el ejemplo siguiente se ve cómo se aplica el caso LL. El árbol de la izquierda es un árbol AVL, donde el factor de balance de los nodos nunca es mayor que 1 o menor que -1. Luego de la inserción del nodo 36, el nodo 96 (raíz) tiene un factor de balance 2, por lo tanto, debe ser corregido. La situación planteada es un LL (Left-Left), por lo que se deberá aplicar simplemente una rotación a derecha entre el nodo 96 y su nodo hijo a izquierda (85).

El sub-árbol hijo a la derecha de 85, es el 90. Se pasa el 90 como nodo hijo a la izquierda de 96 (lo cual va a ser siempre posible, porque es el hijo que se libera, por el 85). Luego se agrega 96 como hijo a la derecha de 85 (que quedó liberado en el paso anterior) y finalmente 85 pasa a ocupar el lugar de 96 (en este caso, como nodo raíz).

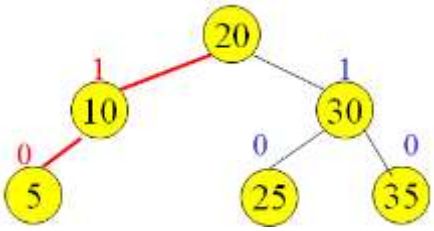


(a) (b) (c)

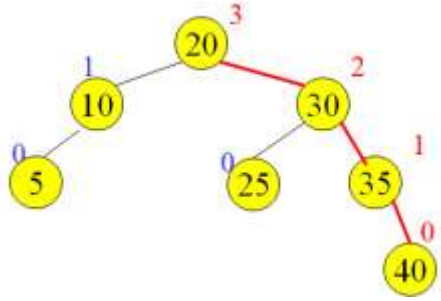
Ejemplo de rotación LL. Se inserta un nuevo valor 36 al árbol de la izquierda (a), resultando en un árbol desbalanceado. El primer nodo desbalanceado, predecesor del nuevo nodo, es el 96, con un factor de balance de 2 (b). El nodo presenta una situación LL, por lo que se precisa una rotación a derecha, con su hijo a izquierda, para obtener un árbol balanceado (c).

Es importante notar que, siguiendo este proceso, toda acción de inserción (y rotación) termina en un árbol AVL.

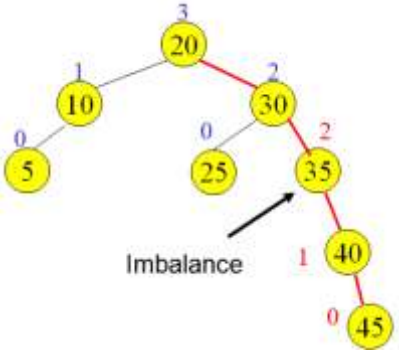
Ejemplo



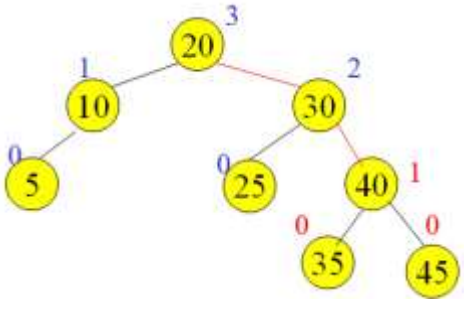
Paso 1: Árbol balanceado original.



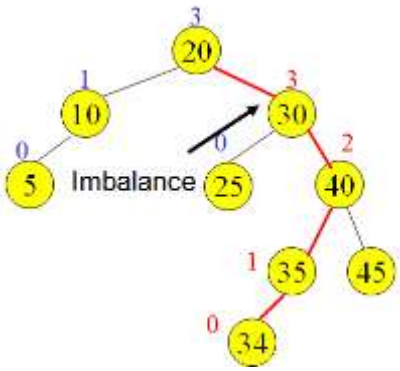
Paso 2: Resultado luego de insertar 40. El árbol continúa siendo balanceado.



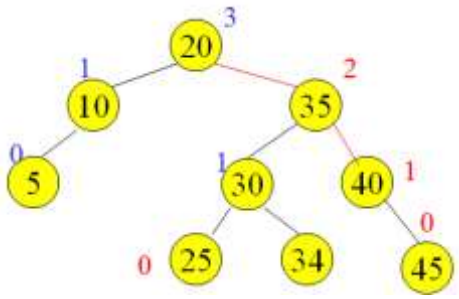
Paso 3: Resultado de insertar 45. El árbol no está más balanceado.



Paso 4: Se corrige el árbol haciendo una rotación a izquierda en el nodo 35 (por ser un caso RR).



Paso 5: Resultado de insertar 34. El árbol está nuevamente desbalanceado.

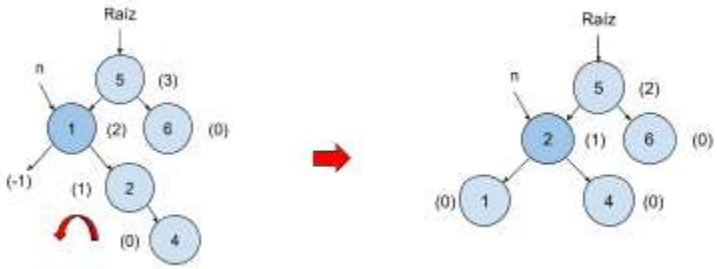


Paso 6: La situación en 30 es RL (el 34 está en el sub-árbol izquierdo del hijo derecho de 30). Se realiza una corrección usando dos rotaciones, uno a derecha (entre 40 y 35) y luego uno a izquierda (entre 30 y 35). El árbol está nuevamente balanceado.

Pseudocódigo

Pseudocódigo de rotación a izquierda:

```
RotarIzquierda(nodo *n)
{
   Node *p;
   p = n.right;
   n.right = p.left;
   p.left = n;
   n = p;
   return;
}
```



Pseudocódigo para inserción:

```
Insert(tree *T, int x) : {
if( T = null ) {
  T = (*tree) malloc (....);
  T.value = x;
  T.heigh = 0;
  return;
}
Case
  // Es duplicado, no se inserta
  x == T.value : return ;
  // Inserta a izquierda
  x < T.value > : Insert(T.left, x);
              if ((height(T.left)- height(T.right)) = 2){
                   if (T.left.data > x ) then // LL
                       T = RotarDerecha(T);
```

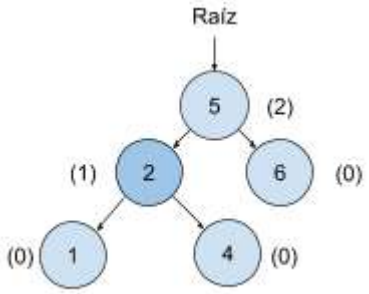
```
                    else                        // LR
                       T = RotarIzquierdaDerecha(T);
                }
  // Inserta a derecha
  x > T.data :  Insert(T.right, x);
                if ((height(T.right)- height(T.left)) = 2){
                     if (x > T.right.data  ) then //(RR)
                        T = RotarLaIzquierda(T);
                     else                          //RL
                        T = RotarDerechaIzquierda(T);
                }
Endcase
  T.height = max(height(T.left),height(T.right)) + 1;
  return;
}
```

Borrado

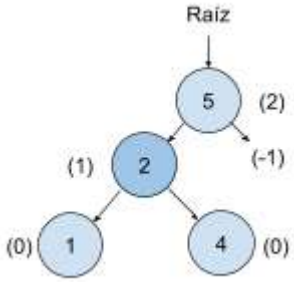
El proceso de borrado de un árbol de búsqueda binario es el siguiente:

a) Si el nodo a borrar es una hoja, se elimina el nodo.
b) Si el nodo a borrar tiene un solo hijo, se reemplaza dicho nodo por su hijo.
c) Si el nodo a borrar tiene dos hijos:
   i) Reemplazar su valor por el menor valor de su sub-árbol a derecha, o por el mayor valor de su sub-árbol a izquierda.
   ii) Eliminar el nodo que se eligió, en forma recursiva (esta vez, tiene un solo hijo, porque es el mínimo o máximo de un sub-árbol).

Similarmente al proceso de inserción, el proceso de borrado de un nodo puede convertir un árbol AVL en uno no balanceado.



Árbol balanceado

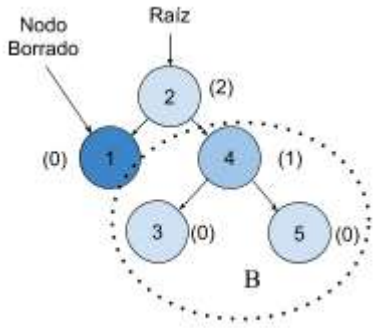


Árbol desbalanceado

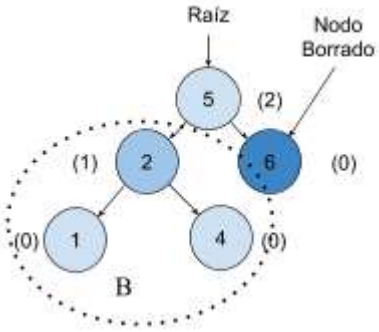
Ejemplo de árbol balanceado que pierde su propiedad después de la remoción de un nodo. Los números junto a los nodos indican la profundidad del nodo. Luego de eliminar el nodo 6, el nodo 5 tiene factor de balance 1-(-1) = 2.

Pueden ser necesarias una o más rotaciones para balancear el árbol. El proceso de rebalancear el árbol consiste en las siguientes operaciones:

a) Buscar el antecesor más cercano al nodo borrado con factor de balance -2 o 2. Se lo llama nodo A.
b) Clasificar la situación como L (Left) o R (Right), dependiendo de si el nodo eliminado estaba a la izquierda o derecha de A.
c) Sea B la raíz del sub-árbol de A que NO contiene al nodo eliminado. B es el sub-árbol derecho si la situación es L y el sub-árbol izquierdo si la situación es R.

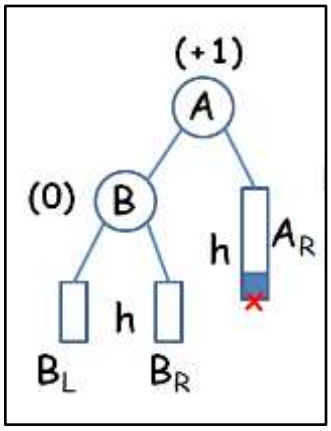


Situación L

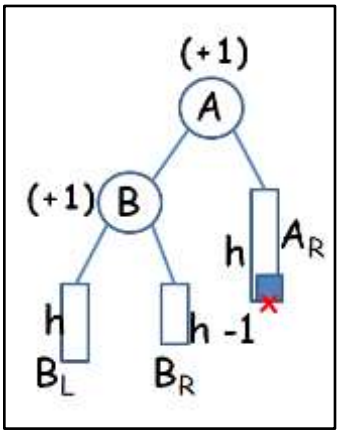


Situación R

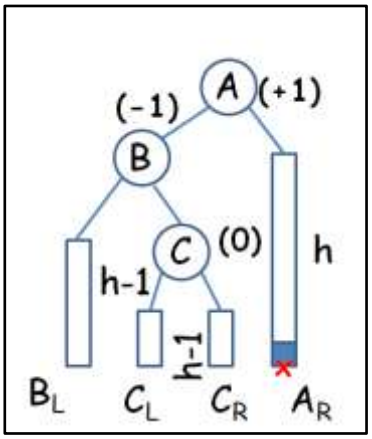
d) Subclasificar la situación en función del factor de balanceo fb(B) de B.



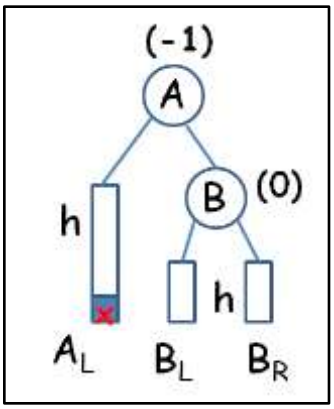
CASO R0
Es una situación R y fb(B) = 0. Se aplica una rotación a derecha entre A y B.



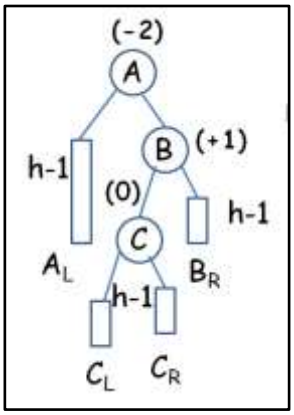
CASO R1
Es una situación R y fb(B) = 1. Se aplica una rotación a derecha entre A y B.



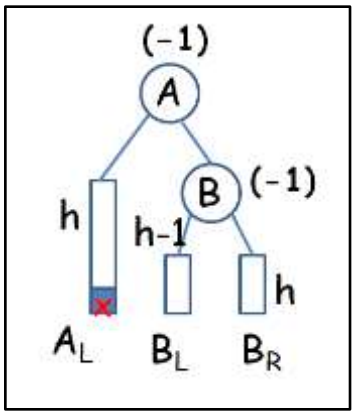
CASO R-1
Es una situación R y fb(B) = -1. Primero rotación a izquierda entre B y su hijo a derecha, luego rotación a derecha con A.



CASO L0
Es una situación L y fb(B) = 0.
Se aplica una rotación a izquierda entre A y B.



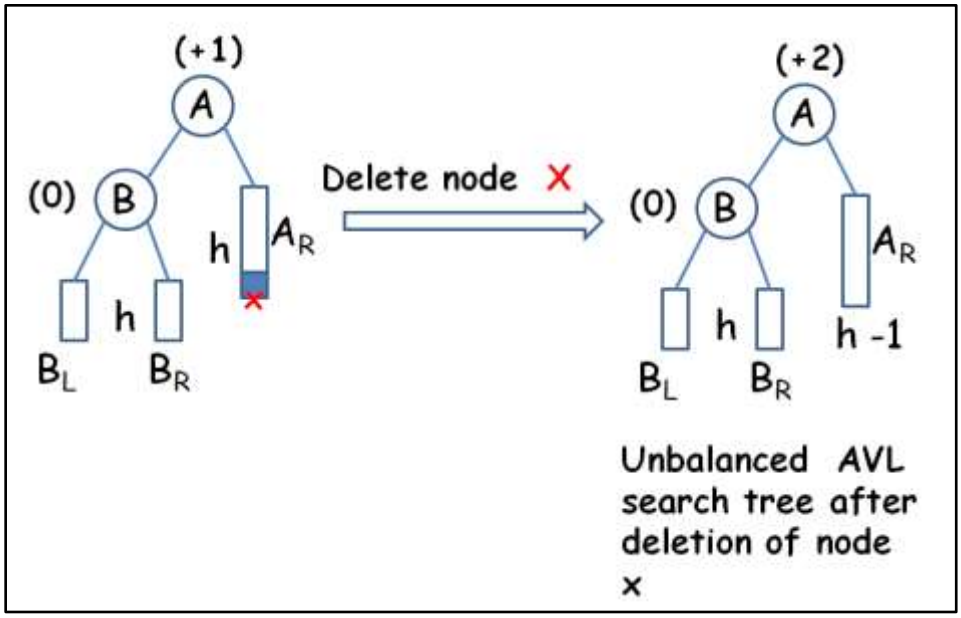
CASO L1
Es una situación L y fb(B) = 1.
Primero rotación a derecha entre B y su hijo a izquierda, luego rotación a izquierda con A.



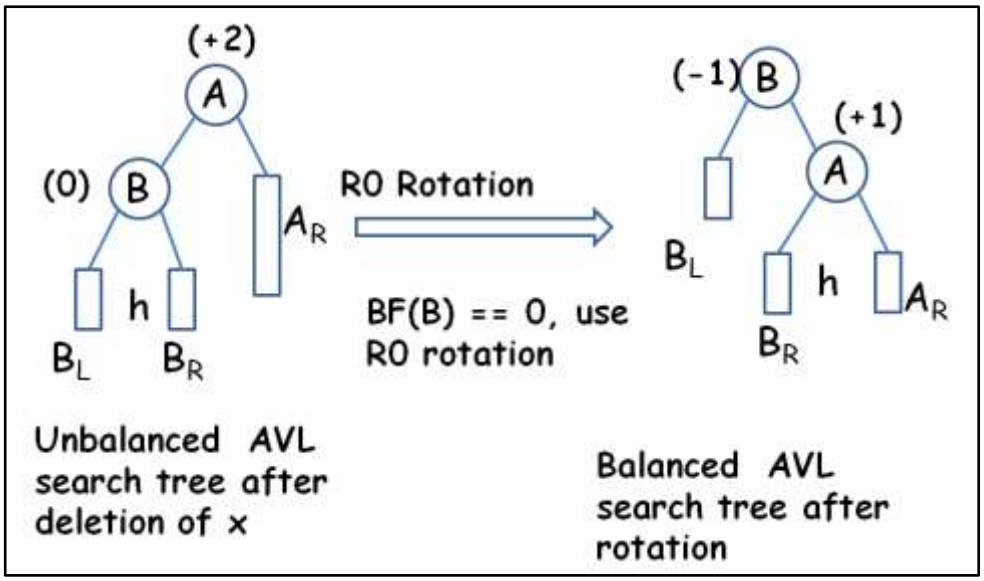
CASO L-1
Es una situación L y fb(B) = -1.
Se aplica una rotación a izquierda entre A y B.

e) Repetir desde el punto a), hasta que estén todos los nodos balanceados, o sea, el factor de balance de todos los nodos, desde el punto de borrado hasta la raíz, sea 1,0 o -1.

Ejemplos

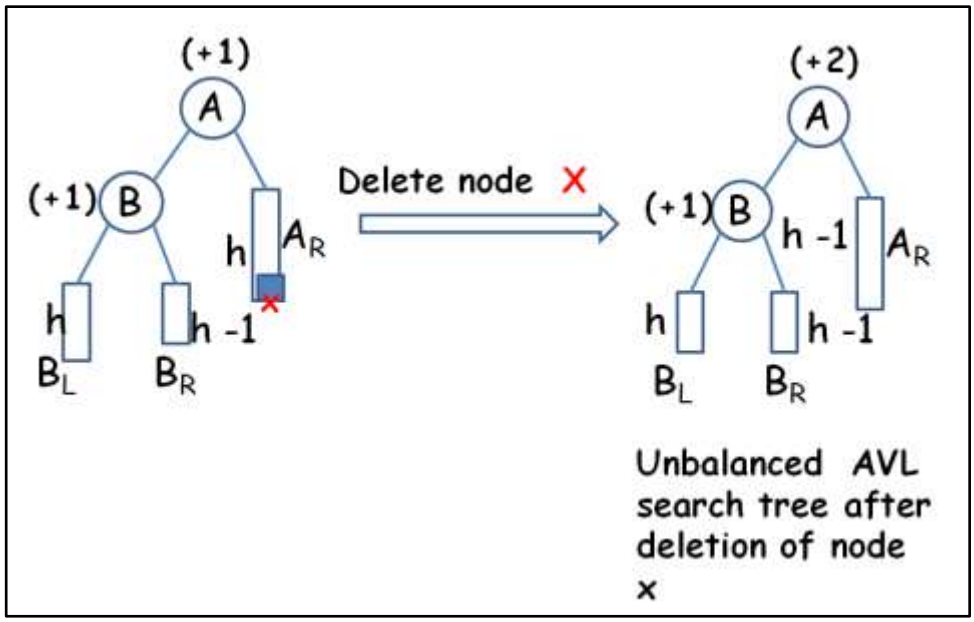


(a)

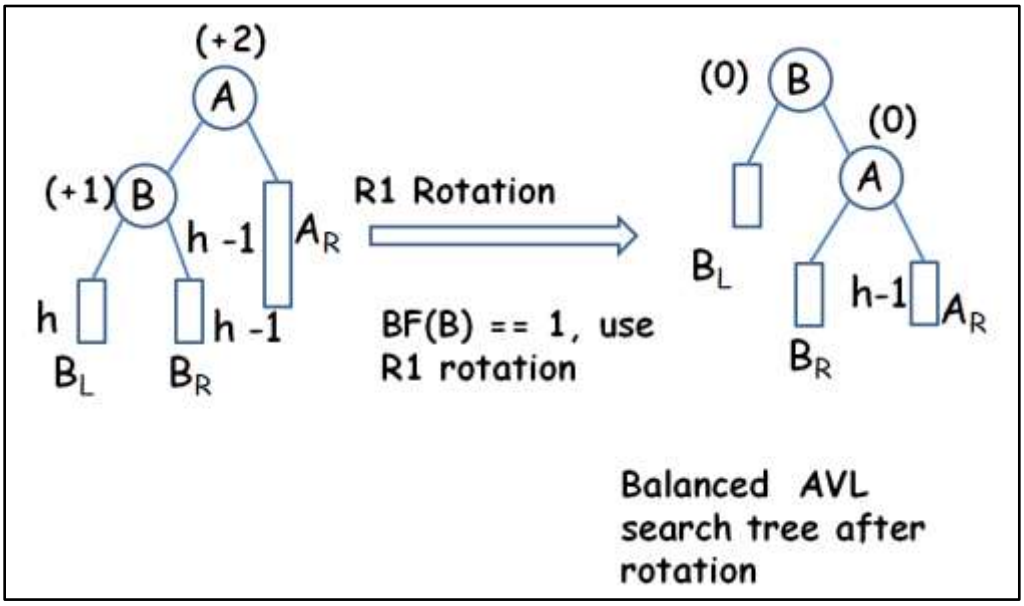


(b)

***Ejemplo de situación R0**. Se aplica una rotación a derecha entra A y B. B ocupa el lugar de A. El hijo derecho de B pasa a ser hijo izquierdo de A.*

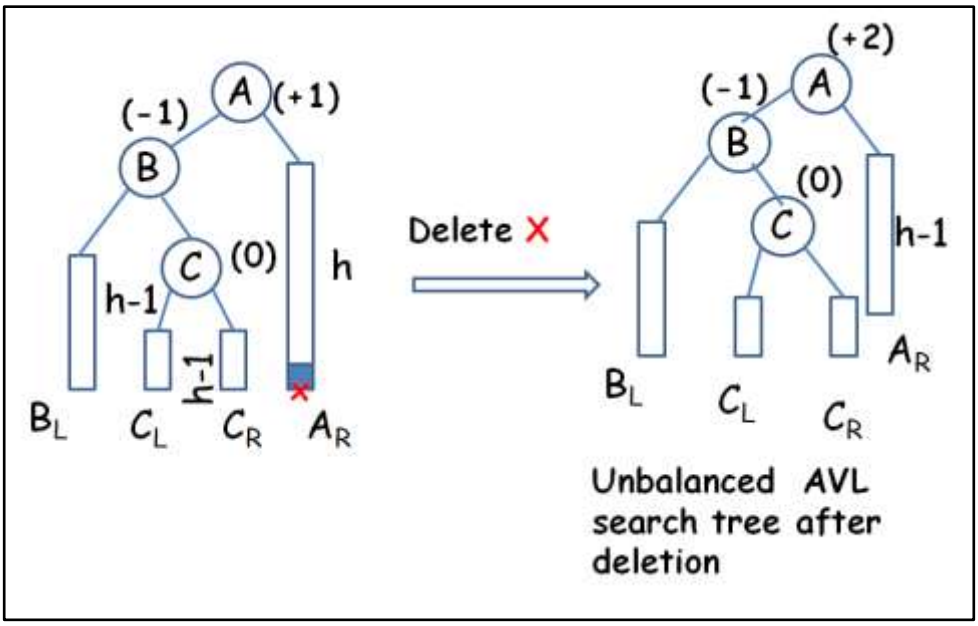


(a)

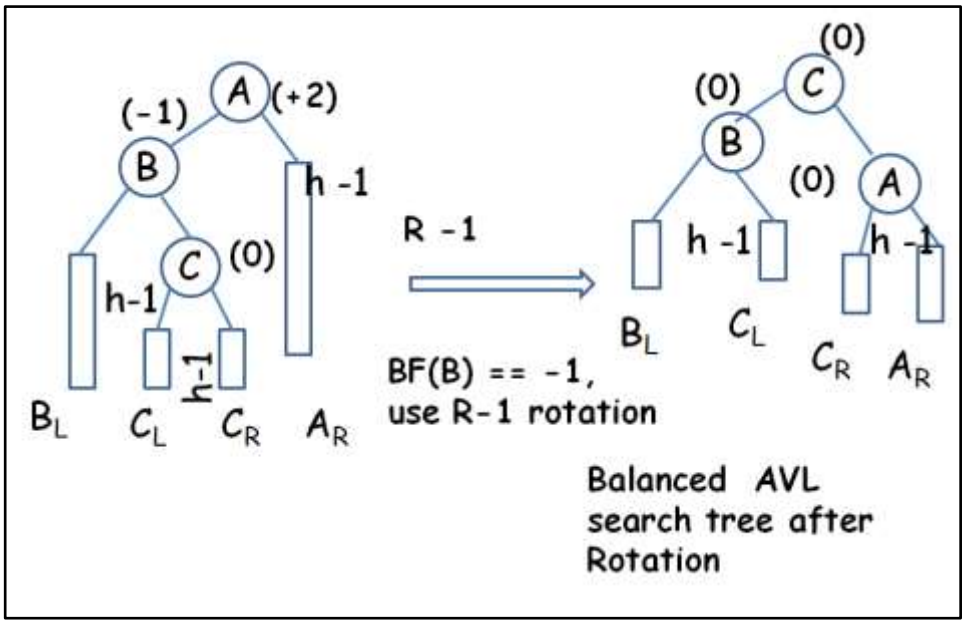


(b)

Ejemplo de situación R1. *Se aplica una rotación a derecha entre A y B. B ocupa el lugar de A. El hijo derecho de B pasa a ser hijo izquierdo de A.*

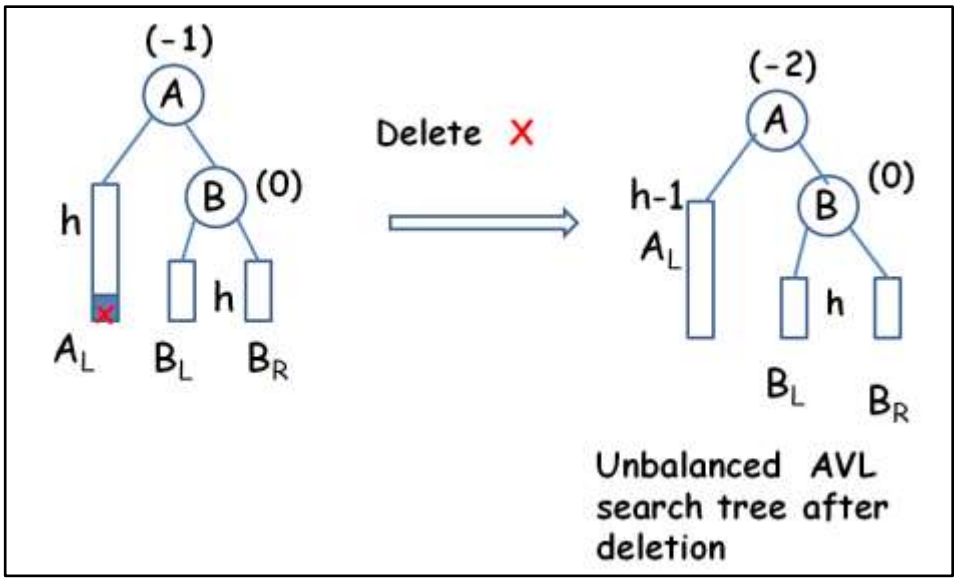


(a)

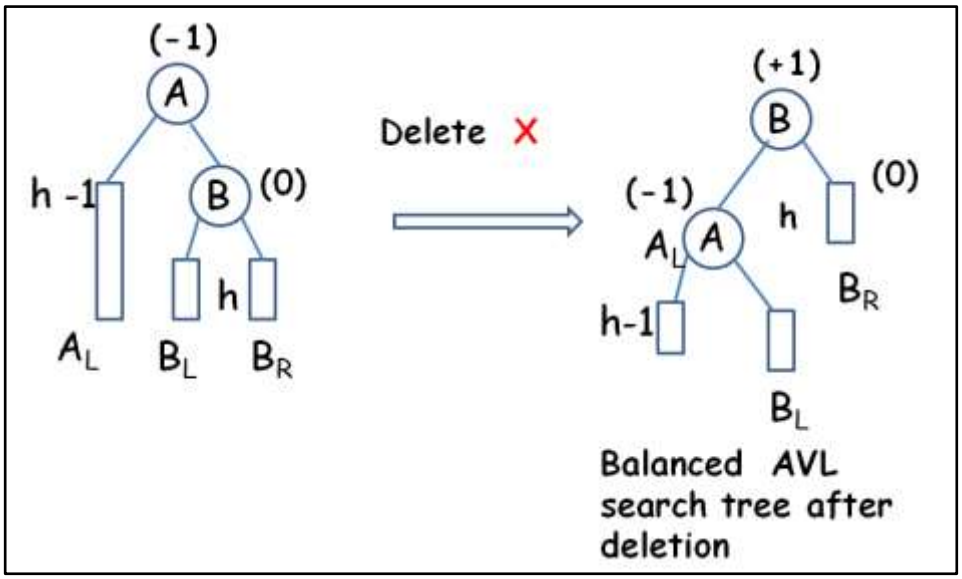


(b)

Ejemplo de situación R-1. *Se aplica primero una rotación a izquierda entre B y su hijo a derecha, luego una rotación a derecha con A. El hijo derecho de B (llamado C) ocupa el lugar de A. El hijo izquierdo de C pasa a ser hijo derecho de B. El hijo derecho de C pasa a ser hijo izquierdo de A. B pasa a ser hijo izquierdo de C. A pasa a ser hijo derecho de C.*

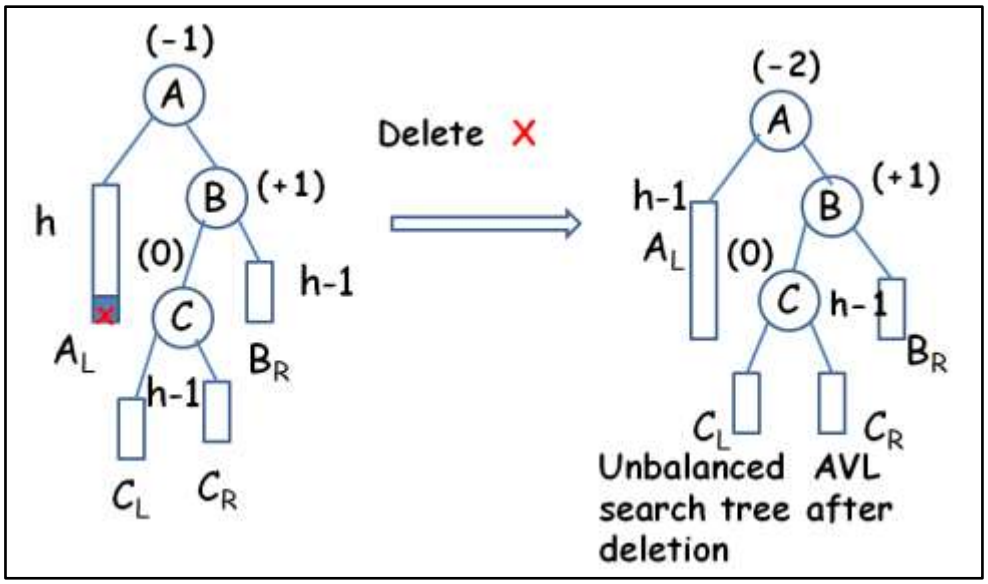


(a)

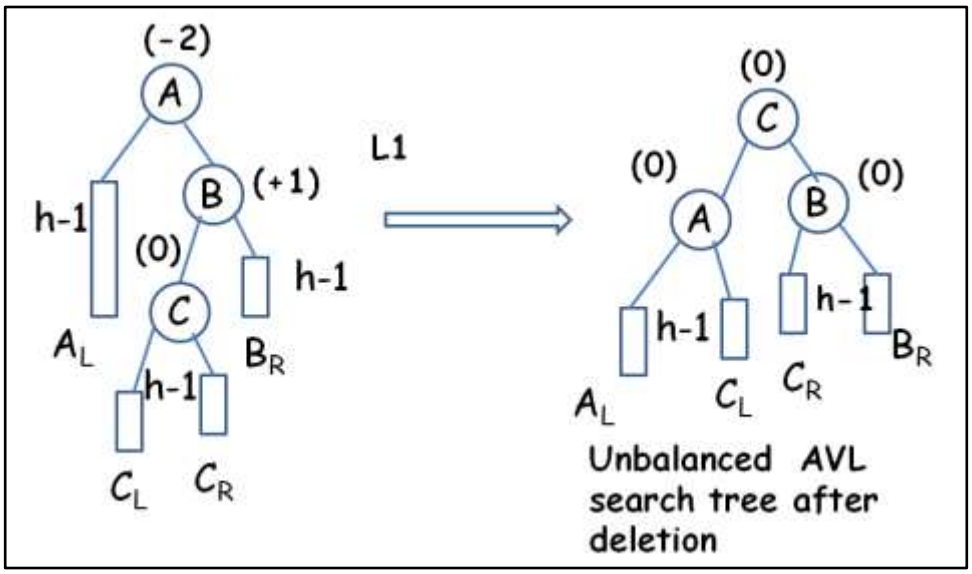


(b)

Ejemplo de situación L0*. Se aplica una rotación a izquierda entre A y B. B pasa ocupa el lugar de A. El hijo izquierdo de B pasa a ser hijo derecho de A.*

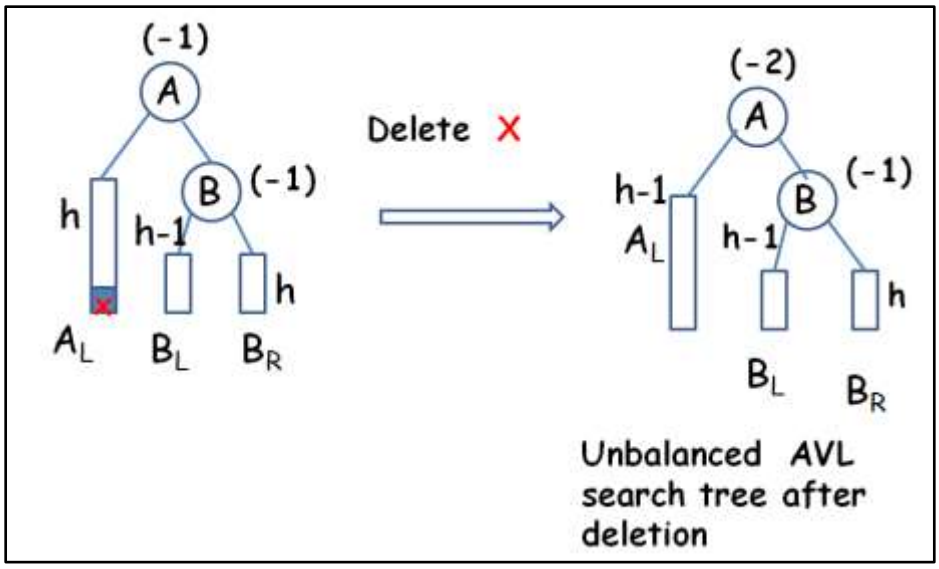


(a)

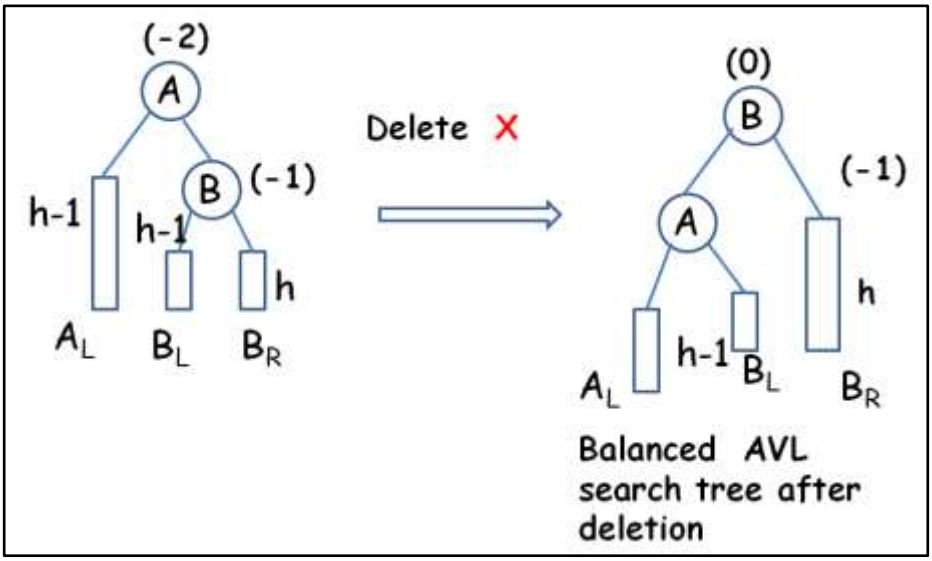


(b)

Ejemplo de situación L1. *Se aplican primero una rotación a derecha entre B y su hijo a izquierda, luego una rotación a izquierda con A. El hijo izquierdo de B (llamado C) ocupa el lugar de A. El hijo derecho de C pasa a ser hijo izquierdo de B. El hijo izquierdo de C pasa a ser hijo derecho de A. B pasa a ser hijo derecho de C. A pasa a ser hijo izquierdo de C.*



(a)



(b)

Ejemplo de situación L-1. *Se aplica una rotación a izquierda entre A y B. B pasa ocupa el lugar de A. El hijo izquierdo de B pasa a ser hijo derecho de A.*

Ventajas

a) Las búsquedas son en tiempo O(log n), pues el árbol está siempre balanceado.
b) Inserciones y borrados son siempre en O(log n).
c) El balanceo, luego de la inserción, tiene un tiempo constante.

Desventajas

Dificultad para programar

Árboles Rojo-Negro

Un árbol Rojo-Negro (RB-tree) es un tipo de árbol de búsqueda binaria auto balanceado. Su implementación es más compleja que la de los árboles AVL, pero es más eficiente (menor cantidad de operaciones en el peor caso, para inserciones y deleciones). Se pueden realizar operaciones de búsqueda, inserción y deleción en tiempo O(log n).

Definición

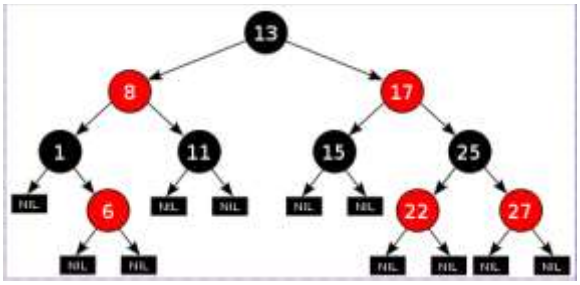
Un árbol Rojo-Negro es un árbol de búsqueda binario en el cual cada nodo tiene asignado un color, rojo o negro. Se espera que el RB-Tree mantenga tres propiedades constantes:

1) **Orden** - propiedad de los árboles binarios de búsqueda - el valor de un nodo es mayor que el valor de sus descendientes a izquierda y menor que el valor de sus descendientes a derecha.
2) **Altura** - Cualquier camino desde un nodo cualquiera a sus hojas descendientes contiene la misma cantidad de nodos negros.
3) **Color** - Los hijos de un nodo rojo solo pueden ser negros.

La especificación completa incluye los siguientes requerimientos:

1. Todo nodo es negro o rojo.
2. Todos los nodos hojas (punteros nulos) son negros.
3. Si un nodo es rojo, sus hijos son negros.
4. Todo camino desde un nodo a cualquier hoja descendiente contiene la misma cantidad de nodos negros.
5. El nodo raíz es negro.

Observación: Todos los nodos hojas son nodos vacíos, que se adicionan a continuación de los nodos con valor.



Ejemplo de árbol Rojo-Negro.

La razón de incluir los requerimientos adicionales es que permiten que se cumplan las siguientes propiedades:

1) El camino más largo entre la raíz y una hoja nunca es mayor que el doble del camino más corto entre la raíz y una hoja.
2) Esto resulta en que el árbol mantiene cierto balance.

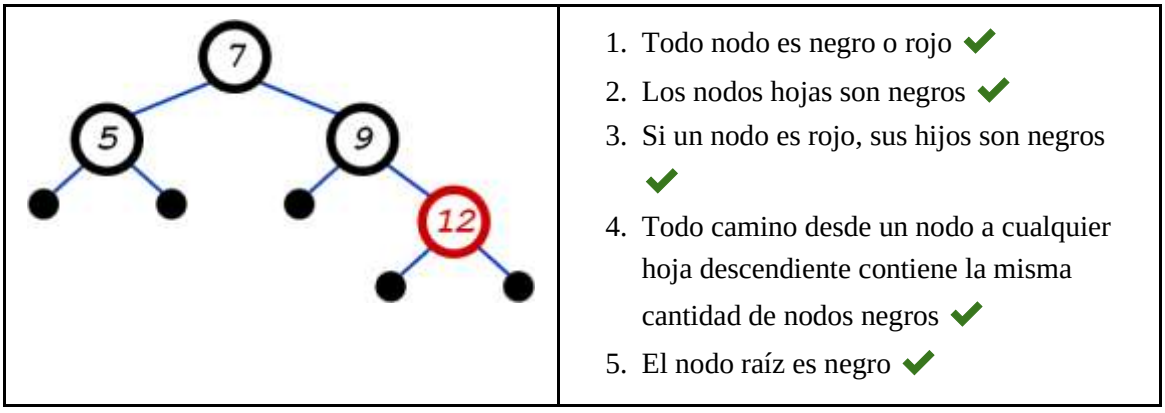
3) Inserciones, deleciones y búsqueda requieren, en el peor caso, tiempo proporcional a la altura del árbol.

El costo de las ventajas adicionales es el de una mayor complejidad en las inserciones y deleciones. El recorrido y la búsqueda son similares a las vistas anteriormente, dado que el árbol sigue siendo un BST.

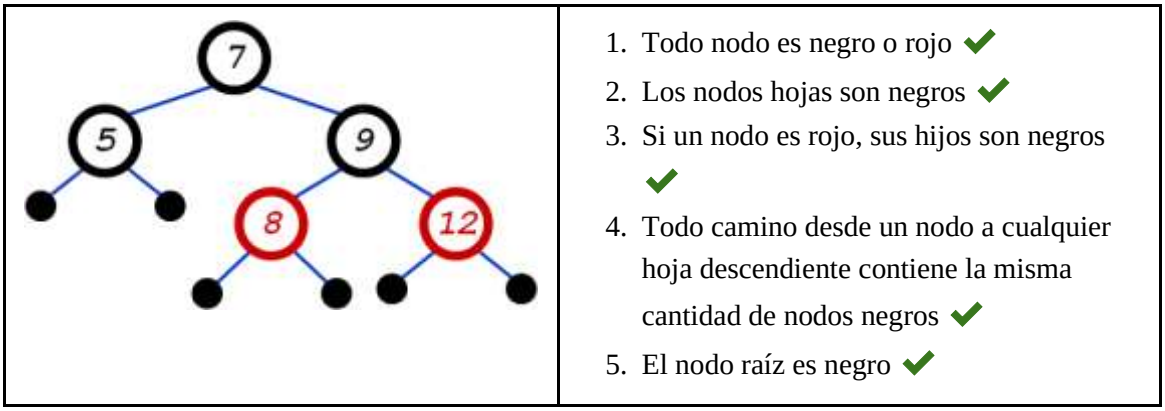
Inserción

A continuación, se verá un problema con las inserciones:

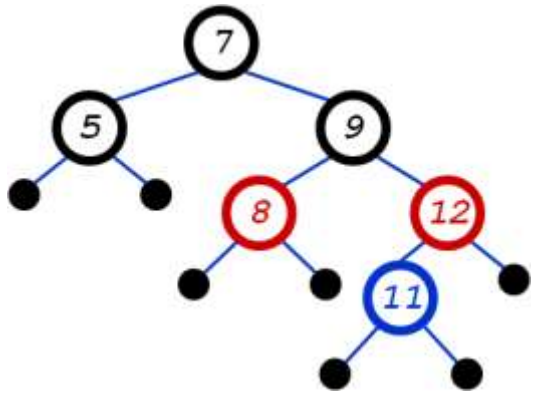
a) Se comienza con un árbol coloreado que cumple con las 5 condiciones:



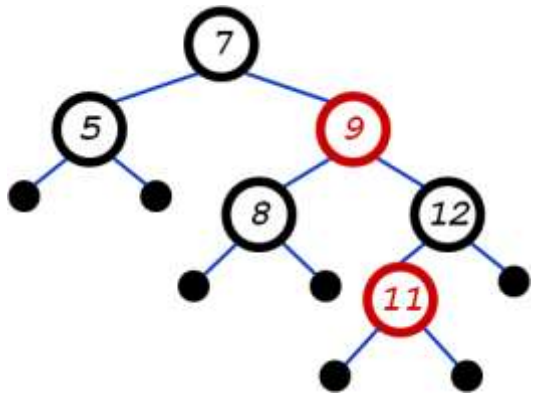
b) Se inserta el número 8 (como en el caso de BST). ¿De qué color debería ser? Rojo, para cumplir con la condición 4 (si fuera negro, el camino de 7 a 8 tendría 3 nodos negros, mientras que el camino de 7 a 12 tendría solo dos nodos negros).



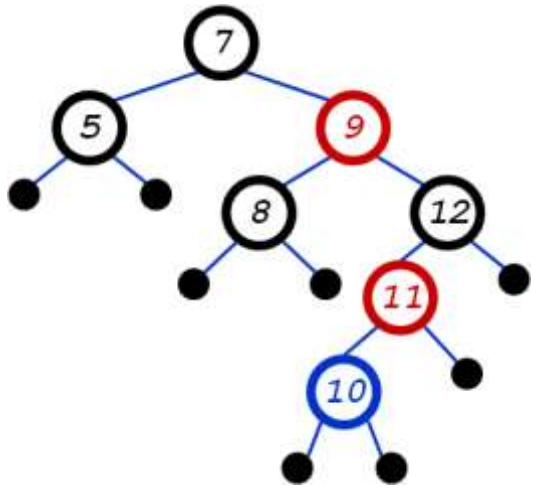
c) Se inserta el número 11 (como en el caso de BST). ¿De qué color debería ser? Si es rojo, no se cumpliría la condición 3 (el 11 es hijo del nodo 12, que es rojo, y solo puede tener descendientes negros). Si es negro, no se cumple la condición 4 (pues el camino del 7 al 11 tendría 3 nodos negros, mientras el camino del 7 al 8 tiene 2 nodos negros).

	1. Todo nodo es negro o rojo ✔ 2. Los nodos hojas son negros ✔ 3. Si un nodo es rojo, sus hijos son negros ✘ 4. Todo camino desde un nodo a cualquier hoja descendiente contiene la misma cantidad de nodos negros ✘ 5. El nodo raíz es negro ✔

d) Solución, se re-colorea TODO el árbol. Ahora se cumplen todas las propiedades:

	1. Todo nodo es negro o rojo ✔ 2. Los nodos hojas son negros ✔ 3. Si un nodo es rojo, sus hijos son negros ✔ 4. Todo camino desde un nodo a cualquier hoja descendiente contiene la misma cantidad de nodos negros ✔ 5. El nodo raíz es negro ✔

e) Ahora se inserta el 10. No puede ser rojo, pues no cumpliría la condición 3. No puede ser negro, pues no cumpliría la condición 4. ¿Se puede re-colorear? NO.

	1. Todo nodo es negro o rojo ✔ 2. Los nodos hojas son negros ✔ 3. Si un nodo es rojo, sus hijos son negros ✘ 4. Todo camino desde un nodo a cualquier hoja descendiente contiene la misma cantidad de nodos negros ✘ 5. El nodo raíz es negro ✔

Considerando el camino 7-5 que solo tiene dos nodos, y sabiendo que el nodo raíz es negro, hay dos opciones:

1) El nodo 5 es rojo. En este caso todos los caminos de la raíz a una hoja deberían tener un solo nodo negro. Ese nodo sería el 7. Por lo tanto, todos los nodos descendientes de 7 deberían ser rojos. Pero entonces no se cumpliría la condición 3 (9 y 8 serían rojos, por ejemplo).
2) El nodo 5 es negro. En este caso todos los caminos de la raíz a una hoja deberían tener dos nodos negros. Como el nodo raíz es negro, solo queda un nodo negro para cada camino. En particular, el camino 7,9,12,11,10 tiene 5 nodos. No importa cuál de esos nodos se pinta de negro, quedarán 3 nodos rojos, de los cuales dos serán consecutivos.

Por lo tanto, no existe ninguna forma de pintar de rojo y negro el nuevo árbol tal que se cumplan las 5 condiciones. Eso sucede porque el árbol resultante no es lo suficientemente balanceado. La solución es rebalancear el árbol de forma tal que exista un coloreado del árbol compatible con las 5 condiciones.

A partir de esta operación, la inserción en un árbol rojo-negro seguirá los siguientes pasos:

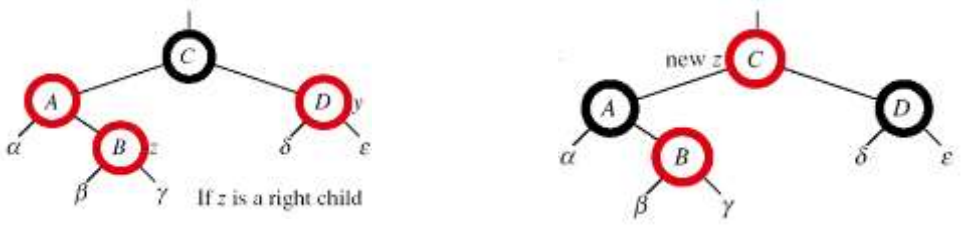
a) Insertar el nuevo nodo como en el caso de los BST.
b) Colorear el nuevo nodo como rojo.
c) Si no cumple más con las 5 propiedades, corregir el árbol usando rotaciones.

Para ello, se analiza que sucede con las propiedades si insertamos siempre un nodo rojo:

1) Todo nodo es negro o rojo - ok, se mantiene. ✔
2) Los nodos hojas son negros - Ok. Los nodos hojas son los hijos (null) del nuevo nodo. ✔
3) Si un nodo es rojo, sus hijos son negros - puede no cumplirse si el nodo padre del nuevo nodo es rojo. ✘
4) Todo camino desde un nodo a cualquier hoja descendiente contiene la misma cantidad de nodos negros - OK. se agregó un nodo rojo. ✔
5) El nodo raíz es negro - Puede no cumplirse si el nuevo nodo es la raíz. En este caso, simplemente se pinta la raíz de negro. ✘

Por lo tanto, excepto por el caso 5 (nodo raíz rojo, que se soluciona pintándolo de negro) el único problema que se genera es el posible incumplimiento de la condición 3 (que el nuevo nodo tenga un "padre" rojo). Dicho caso se deberá corregir en forma iterativa, llevando el problema "hacia arriba" en función de las diferentes opciones del problema:

a) **CASO 1.** El nodo nuevo (rojo), etiquetado z, tiene un "tío" rojo, como se ve en la figura abajo a la izquierda. En este caso, si se cambian el nodo C a rojo y los nodos A y D a negro, se mantiene la cantidad de nodos negros en cada camino (**propiedad 4**), y no hay (de C para abajo) dos nodos rojos consecutivos, como se ve en la figura de la derecha, por lo que se mantiene la **propiedad 3**.



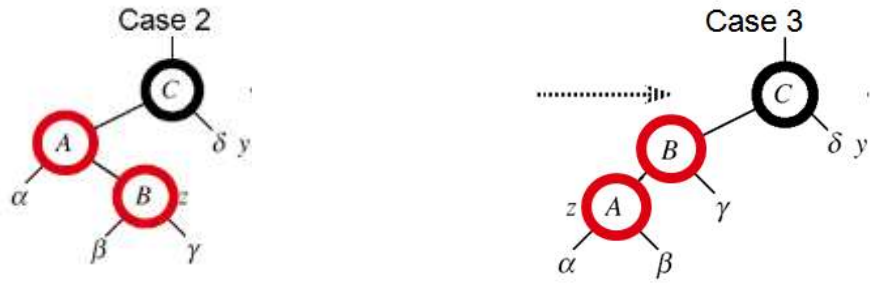
El problema que se puede presentar es que el padre de C sea rojo. En ese caso, se llama *z* a C, y se aplica una de las soluciones, dependiendo de la nueva situación, con respecto al nuevo *z*. Esta solución es similar en el caso que el nuevo nodo esté a la izquierda de su padre, como se ve en la figura abajo:

b) **CASO 2.** El nodo nuevo (rojo), etiquetado *z*, es hijo izquierdo y tiene un "tío" negro, etiquetado *y*, como se ve en la figura abajo a la izquierda, el tío puede ser un puntero nulo. En este caso, se aplica una rotación a la derecha sobre los nodos B y C, etiquetando de negro el nodo B y de rojo el nodo C. Como el hijo a derecha de C era negro (por definición del caso) y el nodo a derecha de B era también negro (por ser hijo de un nodo rojo), entonces el nuevo C rojo tendrá dos nodos rojos, lo que mantiene la **propiedad 3.**

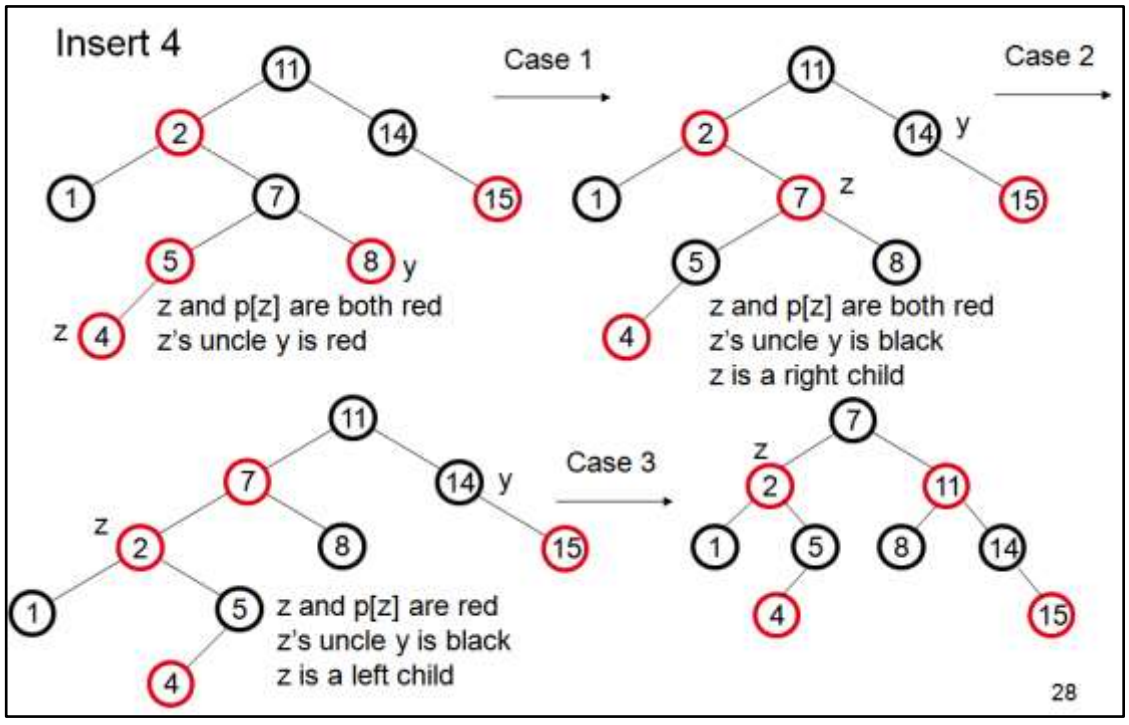
La **propiedad 4** también se mantiene: los descendientes de A, B y C (a derecha) deben tener la misma cantidad de nodos negros (originalmente). Por lo tanto, todos los sub-árboles alfa, beta, gamma y delta tienen la misma cantidad de nodos negros. Eso implica que en el nuevo sub-árbol bajo B, todos los caminos partiendo de B hacia abajo contienen la misma cantidad de nodos negros. Igualmente, para A y C.

c) **CASO 3.** El nodo nuevo (rojo), etiquetado *z*, es hijo derecho y tiene un "tío" negro, etiquetado *y*, como se ve en la figura abajo a la izquierda. En este caso se realiza una rotación a izquierda de los nodos A y B, convirtiendo este problema a la situación anterior, y resolviéndolo de la misma manera. Como no hay cambios de colores, no se altera la **propiedad 4**. Como los 3 sub-árboles alfa, beta y gamma deben

comenzar con nodos negros (porque previamente no se violaba la propiedad 3), la **propiedad 3** se sigue cumpliendo.



Los casos anteriores cubren los posibles quiebres de las propiedades 3 o 4, tanto al agregar un nodo, como en un nodo "abuelo" de un nodo corregido, luego de aplicar una corrección. Como en cada aplicación de una corrección, se sube el problema por lo menos dos nodos hacia arriba, en algún momento se llega al nodo raíz. Si es rojo, se pinta de negro y se termina el proceso.
El siguiente ejemplo muestra todo el proceso de insertar el número 4 en un árbol Rojo-Negro:



Ejemplo de inserción.

Resumen

a) Un **Árbol** es una estructura de datos dinámica con nodos y aristas, donde hay un solo camino entre dos nodos, y un nodo se define como raíz.
b) Un **Árbol Binario** es un árbol donde cada nodo puede tener como máximo 2 hijos.

c) Un **Árbol Binario de Búsqueda** es un árbol binario, donde todos los valores a la izquierda de un nodo son menores que su valor, y todos los valores a la derecha son mayores que su valor.
d) Un **Árbol Binario de Búsqueda Balanceado** es un árbol binario de búsqueda donde se cumple una relación de equilibrio entre los subárboles de cada nodo.
e) Un **Árbol Binario de Búsqueda Auto Balanceado** es un árbol binario de búsqueda con implementaciones de inserción y deleción que permiten mantener el árbol balanceado.
f) Implementaciones de **Árboles Binarios de Búsqueda Auto Balanceados** son **Árboles AVL** y **Árboles Rojo-Negro**.

Árboles B y B+

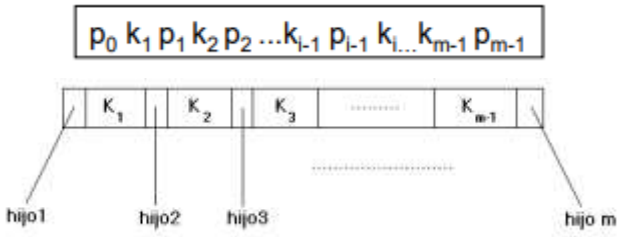
El problema que surge con los árboles binarios de búsqueda (ABB) es cuando se usa un almacenamiento secundario. En este caso, la búsqueda de un elemento requeriría muchos accesos a disco (un acceso a disco es extremadamente lento si se lo compara con un acceso a memoria). Por ejemplo, para un millón de elementos se tiene que:

$$Número\ de\ accesos\ a\ disco = O(h) = O(\log_2 1.000.000) \approx 20$$

La solución es conseguir un mayor grado de ramificaciones para así tener menor altura en el árbol. La altura de un árbol M-ario (multicamino) completo es $O(\log_M N)$. Para el ejemplo del millón de elemento y $M = 10$ se tendría que:

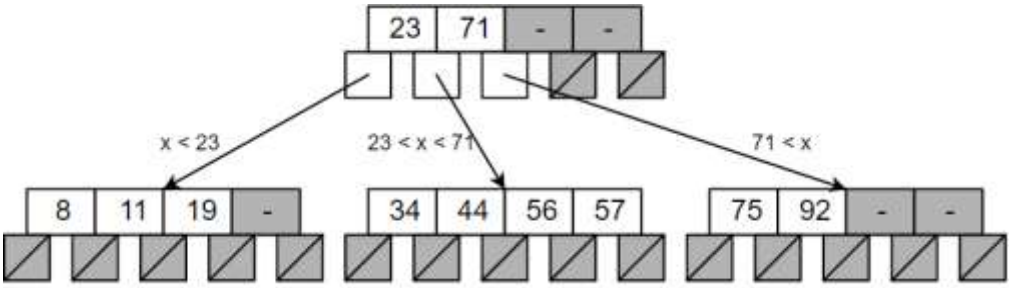
$$Número\ de\ accesos\ a\ disco = O(h) = O(\log_{10} 1.000.000) = 6$$

La idea principal es poder almacenar más datos en cada nodo del árbol, sin que el incremento suponga un trabajo extra de localización de un elemento en el nodo. Este nodo se llamará página:



- Una página es un nodo donde los k_i son elementos tales que $k_{i-1} < k_i$ donde $0 < i < m$ y p_i donde $0 \leq i < m$ son apuntadores a subárboles.
- $\forall i = 1, \dots, m-2$, p_i apunta a una página cuyas clases son mayores o iguales que k_i.
- p_{m-1} apunta a una página cuyas claves son mayores o iguales que k_{m-1}.
- A cada página se accede en bloque.

A continuación, se puede ver un ejemplo de un árbol según esta descripción:



Árboles B

Los árboles B fueron propuestos por Bayer y McCreight en 1972. En las ciencias de la computación, los árboles B son estructuras de datos de árbol que se encuentran comúnmente en las implementaciones de bases de datos y sistemas de archivos. Su principal utilidad se encuentra en la gestión de los índices en bases de datos.

Los árboles B son un caso especial de árboles equilibrados, cuyos nodos pueden tener más de dos hijos y cuyas ramas están ordenadas a modo de árbol binario de búsqueda. Los árboles B mantienen los datos ordenados y las inserciones y eliminaciones se realizan en tiempo logarítmico amortizado.

La idea tras los árboles B es que los nodos internos deben tener un número variable de nodos hijo dentro de un rango predefinido. Cuando se inserta o se elimina un dato de la estructura, la cantidad de nodos hijo varía dentro de un nodo. Para que siga manteniéndose el número de nodos dentro del rango predefinido, los nodos internos se juntan o se parten. Dado que se permite un rango variable de nodos hijo, los árboles B no necesitan rebalancearse tan frecuentemente como los árboles binarios de búsqueda autobalanceados. Pero, por otro lado, pueden desperdiciar memoria, porque los nodos no permanecen totalmente ocupados. Los límites (uno superior y otro inferior) en el número de nodos hijo son definidos para cada implementación en particular.

Los árboles B tienen ventajas sustanciales sobre otras implementaciones cuando el tiempo de acceso a los nodos excede al tiempo de acceso entre nodos. Este caso se da usualmente cuando los nodos se encuentran en dispositivos de almacenamiento secundario como los discos rígidos. Al maximizar el número de nodos hijo de cada nodo interno, la altura del árbol decrece, las operaciones para balancearlo se reducen y aumenta la eficiencia. Usualmente este valor se coloca de forma tal que cada nodo ocupe un bloque de disco, o un tamaño análogo en el dispositivo.

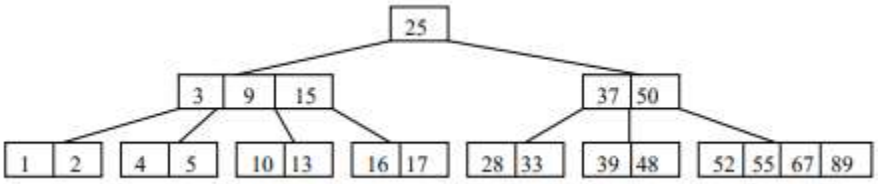
Definición

Un árbol B de orden m, donde m es el máximo número de hijos que puede tener cada nodo, es un árbol que satisface las siguientes propiedades:

a) Cada página, excepto la raíz y las páginas hojas, tienen entre $\frac{m}{2}$ y m hijos.

b) Cada página, excepto la raíz, contiene entre $\frac{m}{2} - 1$ y $m - 1$ elementos.
c) La página raíz, o es una hoja o tiene entre 2 y m hijos.
d) Las páginas hojas están todas al mismo nivel.

A continuación, se muestra un árbol B de orden 5:



Este árbol satisface las propiedades pedidas para ser un árbol B ya que:
- ✓ Cada nodo, excepto la raíz, tiene entre 3 y 5 hijos.
- ✓ Cada página, excepto la raíz, contiene entre 2 y 4 elementos.
- ✓ La página raíz tiene 2 hijos.
- ✓ Las páginas hojas están a la misma altura.

Algoritmo de Búsqueda

Es una generalización del proceso de búsqueda en ABB. Se comienza en la raíz y se recorre el árbol hacia abajo, escogiendo el subnodo de acuerdo a la posición relativa del valor buscado respecto a los valores de cada nodo.
El procedimiento para buscar el elemento x se describe en los siguientes pasos:

1. Situarse en el nodo raíz y comprobar si tiene la clave a buscar. Si la clave es encontrada entonces hemos terminado el procedimiento.
2. Si la búsqueda es infructuosa se estará en una de las siguientes situaciones:
   a) $k_{i-1} < x < k_i$ para $1 < i \leq n$. La búsqueda continúa en la página apuntada por p_{i-1}.
   b) $k_n < x$. La búsqueda continúa en la página apuntada por p_n.
   c) $x < k_1$. La búsqueda continúa en la página apuntada por p_0.
3. Si en algún caso el apuntador p_i es nulo, es decir, si no hay página hijo, entonces no hay ningún elemento x en todo el árbol y se acaba la búsqueda.

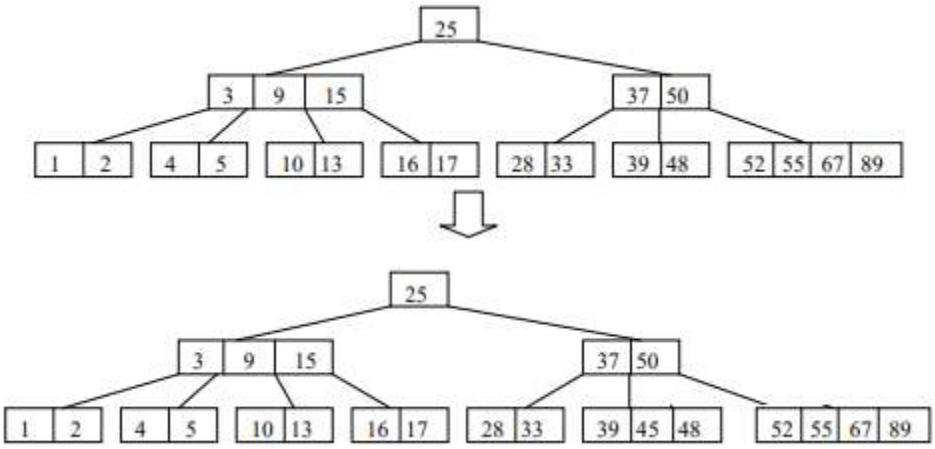
Si n (número de elementos de la página) es suficientemente grande, se puede utilizar la búsqueda binaria. En caso contrario, una búsqueda secuencial será suficiente.

Algoritmo de Inserción

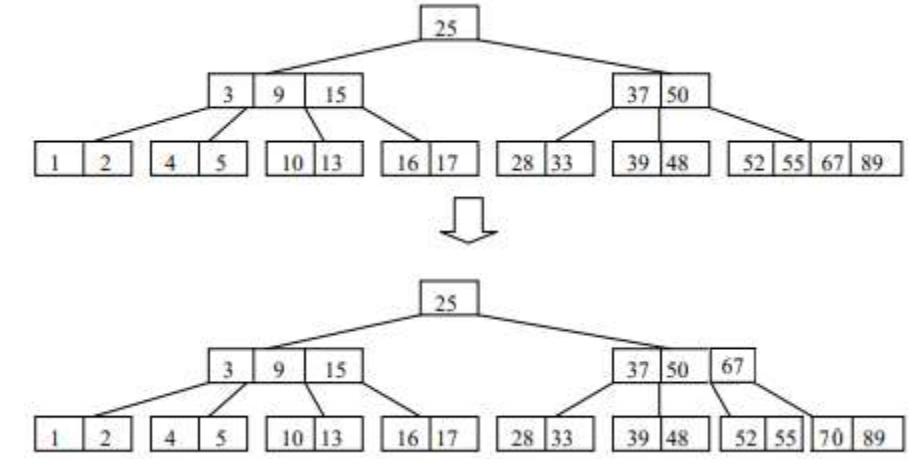
Para insertar una nueva clave usaremos el siguiente algoritmo cuyos pasos se describen a continuación:

1. Se busca la clave a insertar en el árbol siguiendo el algoritmo anterior.
2. Si la clave no está en el árbol, la búsqueda termina en un nodo hoja.
3. Si el nodo hoja no está lleno ($n < m - 1$), la inserción es posible en dicho nodo y el proceso termina.
4. Si la hoja está llena ($n = m - 1$), se divide el nodo (incluyendo virtualmente la nueva clave) en dos nodos. La clave central sube en el árbol por el camino de búsqueda para ser insertada en el nodo ascendente. En esta ascensión puede ocurrir que se llegue al nodo raíz y éste se encuentre lleno. En ese caso, aumenta en 1 la altura del árbol.

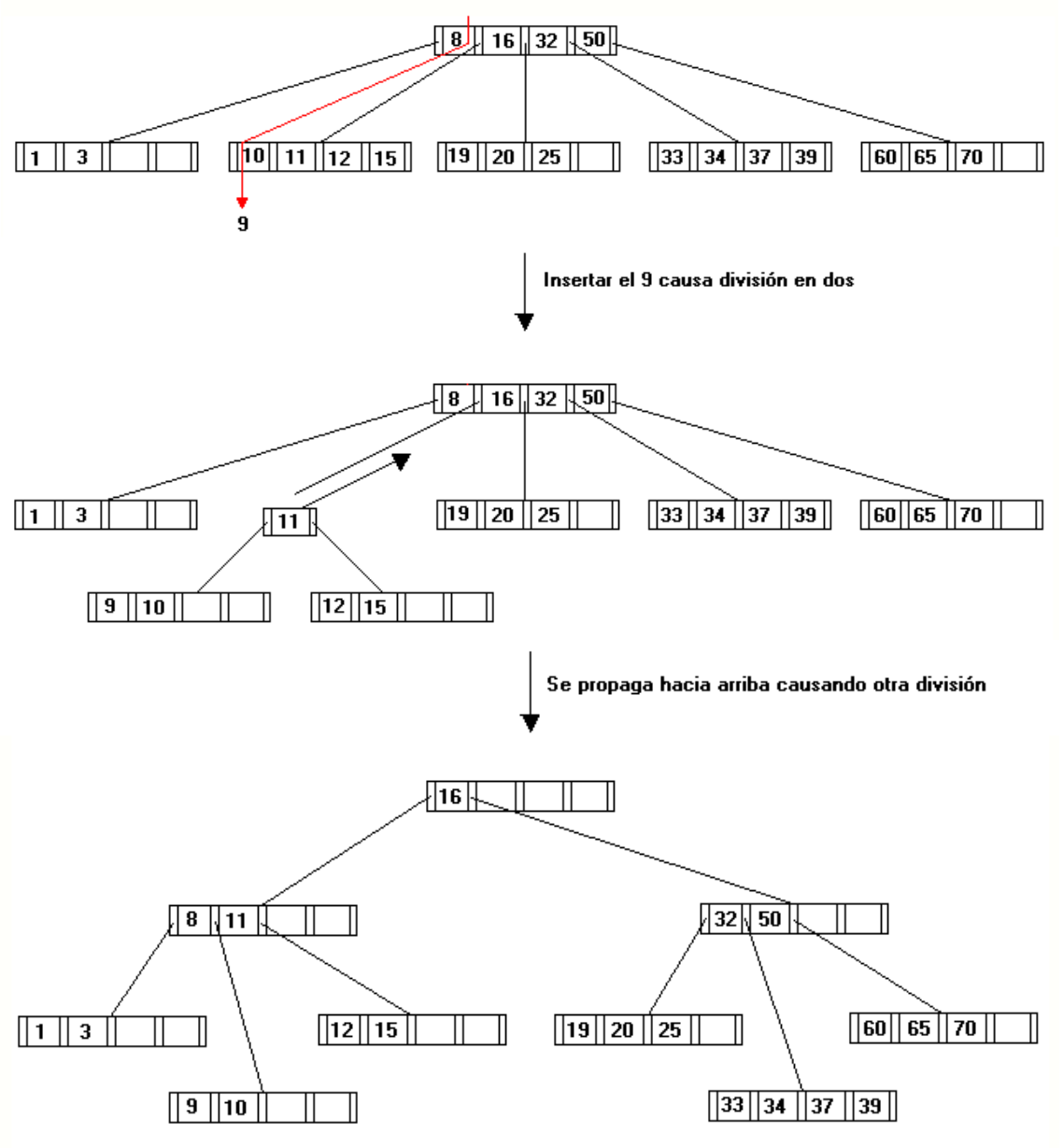
Veamos ejemplos de los distintos casos mencionados:
Ejemplo 1: Inserción de la clave 45 en un nodo que no está lleno.



Ejemplo 2: Inserción de la clave 70 en un nodo que se encuentra lleno.

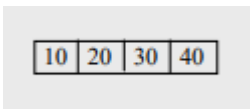


Ejemplo 3: Inserción de la clave 9 en un nodo que se encuentra lleno.

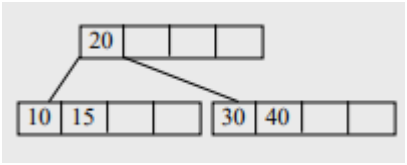


Ejemplo 4: Insertar las siguientes claves en un árbol B inicialmente vacío de orden $m = 5$: $20 - 40 - 10 - 30 - 15 - 35 - 7 - 26 - 18 - 22 - 5 - 42 - 13 - 46 - 27 - 8 - 32 - 38 - 24 - 45 - 25$

Insertamos las primeras 4 claves:

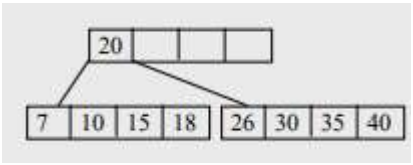


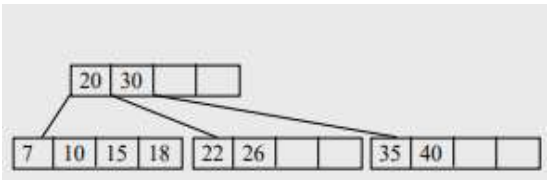
Insertamos el 15:



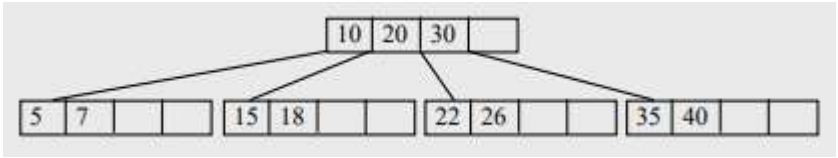
Insertamos las claves 35, 7, 26 y 18:

Insertamos el 22:

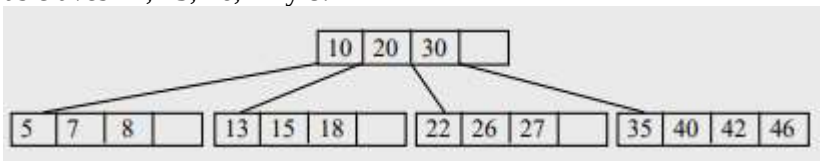




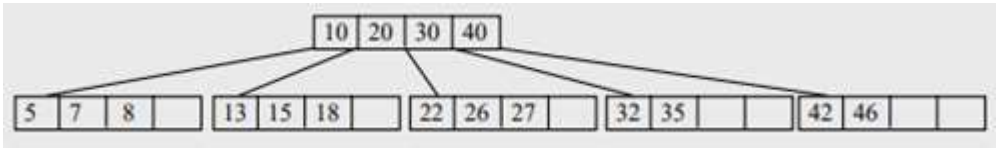
Insertamos el 5:



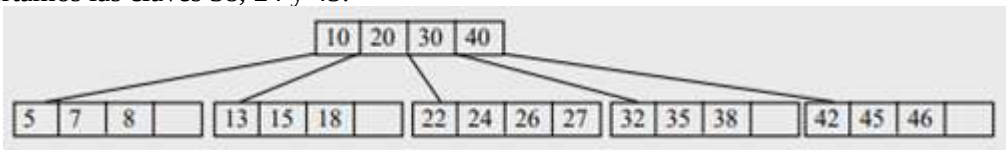
Insertamos las claves 42, 13, 46, 27 y 8:



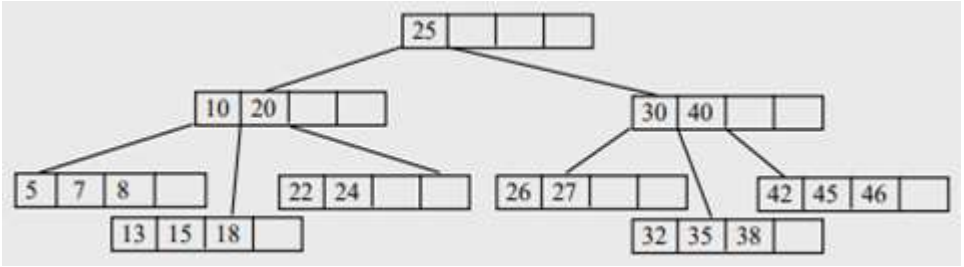
Insertamos el 32:



Insertamos las claves 38, 24 y 45:



Por último, insertamos la clave 25:



Algoritmo de Eliminación

La idea para realizar el borrado de una clave es similar a la inserción teniendo en cuenta que ahora, en lugar de divisiones, realizamos uniones. Existe un problema añadido, las claves a borrar pueden aparecer en cualquier lugar del árbol y por consiguiente no coincide con el caso de la inserción en la que siempre comenzamos desde una hoja y propagamos hacia arriba. La solución a esto es inmediata pues cuando borramos una clave que está en un nodo interior, lo primero que realizamos es un intercambio de este valor con el inmediato sucesor en el árbol, es decir, el hijo más a la izquierda del hijo derecho de esa clave.

Las operaciones a realizar para poder llevar a cabo el borrado son:

- Redistribución: la utilizaremos en el caso en que al borrar una clave el nodo se queda con un número menor que el mínimo y uno de los hermanos adyacentes tiene al menos uno más que ese mínimo, es decir, redistribuyendo podemos solucionar el problema.
- Unión: la utilizaremos en el caso de que no sea posible la redistribución y por tanto sólo será posible unir los nodos junto con la clave que los separa y se encuentra en el padre.

En definitiva, el algoritmo nos queda como sigue:

1. El elemento a borrar se encuentra en una página hoja, entonces se suprime.
2. La clave a borrar no se encuentra en una página hoja, entonces debe sustituirse por la clave que se encuentra más a la izquierda en el subárbol derecho o por la clave que se encuentra más a la derecha en el subárbol izquierdo.

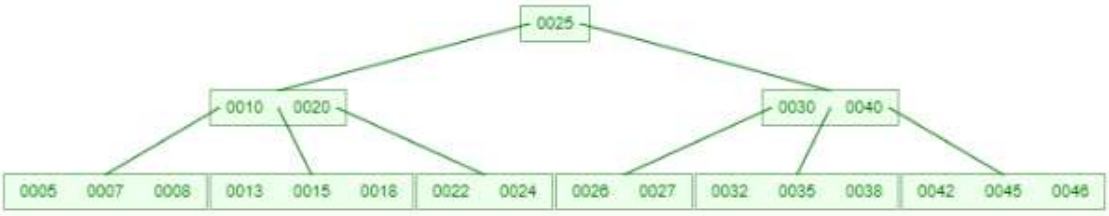
Debe verificarse el valor de n después de la eliminación:

a) Si $n \geq \frac{m}{2} - 1$ entonces se trasladan las claves hacia la izquierda y termina la operación de borrado.

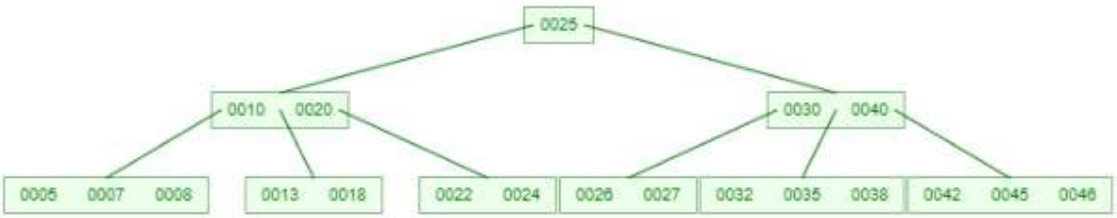
b) En caso contrario, se exploran las hojas hermanas adyacentes. Si en alguna de ellas, $n > \frac{m}{2} - 1$, uno de los elementos sube al nodo padre para que descienda de éste otra clave al nodo que se quiere restaurar.

c) Si en las hojas hermanas contiguas $n = \frac{m}{2} - 1$, se fusiona con una de sus hermanas adyacentes, incluyendo en el nuevo nodo el elemento del padre situado entre ambas páginas. Esta fusión puede dejar al padre con un número de elementos por debajo del mínimo, entonces el proceso de propaga hacia la raíz. Si se utiliza el último elemento de la raíz, la altura del árbol disminuye en una unidad.

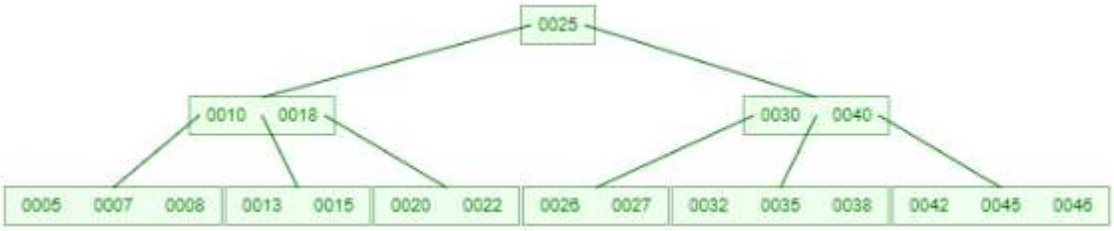
Ejemplos: Consideremos el siguiente árbol B de orden 5 y tomemos como criterios (1) si el nodo tiene dos hijos hay que sustituir por el mayor de la izquierda, (2) consultar el hermano de la derecha.:



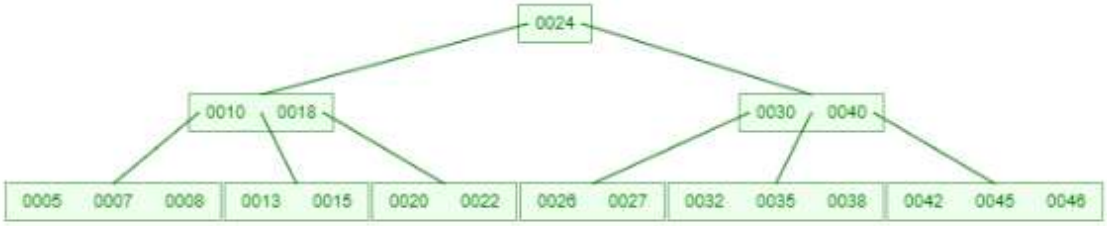
Eliminación de la clave 15 (Caso 1-a):



Eliminación de la clave 24 (Caso 1-b):



Eliminación de la clave 25 (Caso 2-b):



Eliminación de la clave 24 del resultado anterior (Caso 2-c):

En el link https://www.cs.usfca.edu/~galles/visualization/BTree.html podemos ver cómo insertar y eliminar claves en un árbol B.

Árboles B+

Los árboles B+ surgen como una variante de los árboles B convirtiéndose en la técnica más utilizada para la organización de archivos indizados. La principal característica de estos árboles es que todas las claves se encuentran en las hojas (a diferencia de los árboles B, en que las claves podrían están en páginas intermedias) y por lo tanto cualquier camino desde la raíz hasta alguna de las claves tienen la misma longitud. En la raíz y en las paginas internas se encuentran almacenados índices o claves para llegar a un dato.

Definición

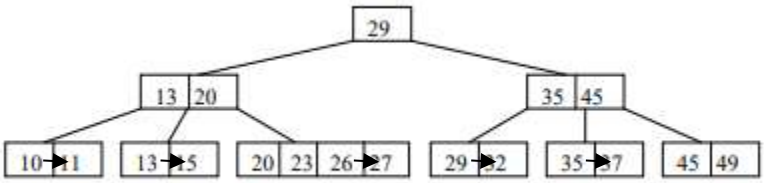
Un árbol B+ de orden m es un árbol que satisface las siguientes propiedades:

a) Cada página, excepto la raíz, tiene entre $\frac{m}{2}$ y m descendientes.

b) Cada página, excepto la raíz, contiene entre $\frac{m}{2} - 1$ y $m - 1$ elementos.

c) La página raíz, o es una hoja o tiene al menos 2 hijos.
d) Las páginas hojas están todas al mismo nivel.
e) Todas las claves se encuentran en las páginas hojas.
f) Las claves de las páginas raíz e interiores se utilizan como índices.
g) Las hojas están enlazadas.

Aquí se muestra un ejemplo de árbol B+ de orden 5:



Algoritmo de Búsqueda

En este caso, la búsqueda no debe detenerse cuando se encuentre la clave en la página raíz o en una página interior, sino que debe proseguir en la página apuntada por la rama derecha de dicha clave.

Algoritmo de Inserción

Su diferencia con el proceso de inserción en árboles B consiste en que cuando se inserta una nueva clave en una página llena, ésta se divide también en otras dos, pero ahora la primera contendrá $\frac{m}{2}$ claves y la segunda $\frac{m}{2} + 1$, y lo que subirá a la página antecesora será una copia de la clave central.
Los pasos a seguir para una inserción son los siguientes:

1. Se ubica en la página raíz.
2. Se evalúa si es una página hoja.

2.1 Si la respuesta es afirmativa, se evalúa si no sobrepasa los límites de datos.

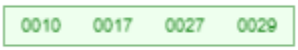
2.1.1 Si la respuesta es afirmativa, entonces se procede a insertar el nuevo valor en lugar del correspondiente.

2.1.2 Si la respuesta es negativa, se divide la página en dos, se sube una copia de la clave central a la página padre, si la página padre se encuentra llena se debe partir igual y así el mismo proceso hasta donde sea necesario. Si este proceso llega hasta la raíz, la altura del árbol aumenta en uno.

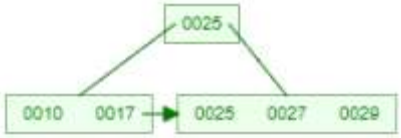
2.2 Si no es hoja, se compara el elemento a insertar con cada uno de los valores almacenados para encontrar la página descendiente donde proseguir la búsqueda. Se regresa al paso 1.

Ejemplo: Insertar las siguientes claves a un árbol B+ de orden 5: 10 – 27 – 29 – 17 – 25 – 21 – 15 – 31 – 13 – 51 – 20 – 24 – 48 – 19 – 60 – 35 – 66.

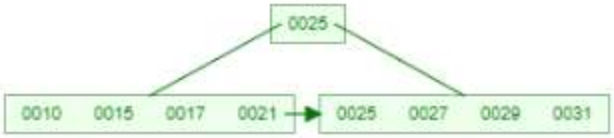
Insertamos las primeras cuatro claves:



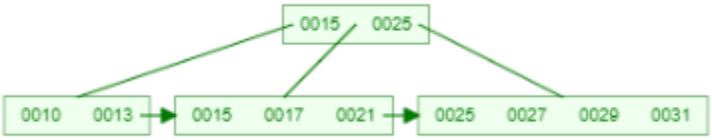
Insertamos la clave 25:



Insertamos las claves 21, 15 y 31:

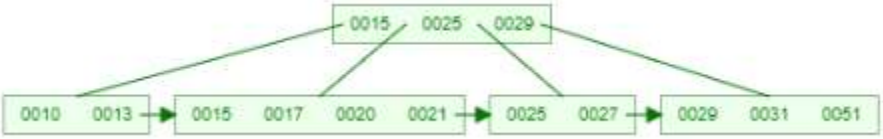


Insertamos la clave 13:



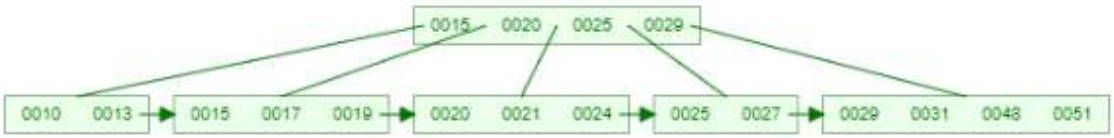
Insertamos la clave 51:

Insertamos la clave 20:

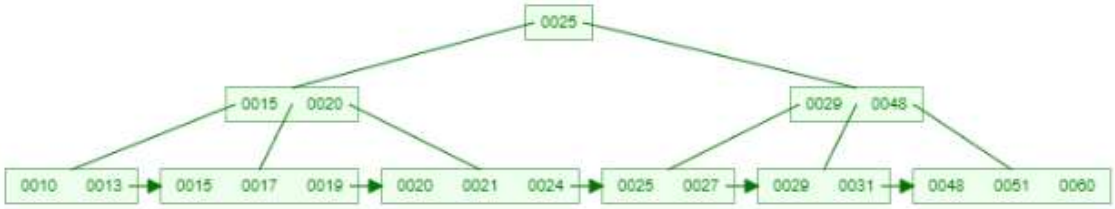


Insertamos la clave 24:

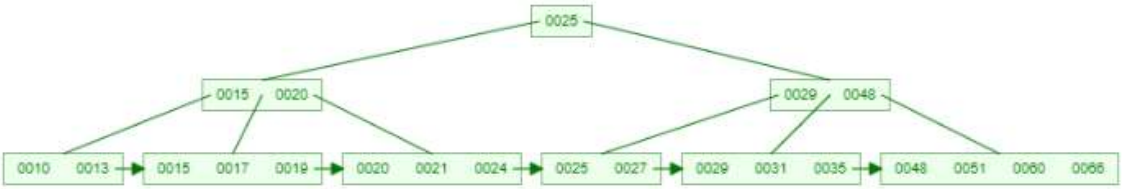
Insertamos las claves 48 y 19:



Insertamos la clave 60:



Finalmente, insertamos las claves 35 y 66:

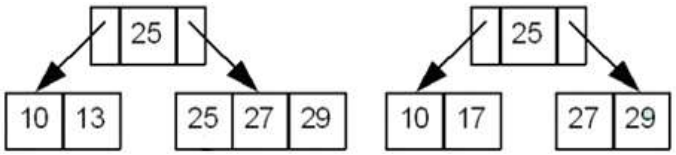


Algoritmo de Eliminación

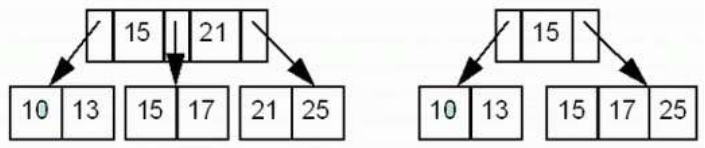
La operación de eliminación en árboles B+ es más simple que en árboles B. Esto ocurre porque las claves a eliminar siempre se encuentran en las páginas hojas.
La operación de borrado debe considerar:

1) Si al eliminar la clave (siempre en una hoja) el número de claves es mayor o igual a $\frac{m}{2}$ el proceso ha terminado. Las claves de las páginas raíz o internas no se modifican, aunque sean una copia de la eliminada, pues siguen constituyendo un separador válido entre las claves de las páginas descendientes.
2) Si al eliminar la clave el número de ellas en la página es menor que $\frac{m}{2}$ será necesaria una fusión y redistribución de las mismas tanto en las páginas hojas como en el índice.

Ejemplos: Caso 1: Eliminamos la clave 25:



Caso 2: Eliminamos la clave 21:



En el link https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html podemos ver cómo insertar y eliminar claves en un árbol B.

Referencias

BST (Binary Search Tree)
[BST-1] http://www.algolist.net/Data_structures/Binary_search_tree/Removal
[BST-2] http://geeksquiz.com/binary-search-tree-set-2-delete/
[BST-3] http://www.stoimen.com/blog/2012/06/22/computer-algorithms-binary-search-tree-data-structure/

SBBST (Self Balancing Binary Search Trees)
[SBBST-1] https://www.cpp.edu/~ftang/courses/CS241/notes/self%20balance%20bst.htm
[SBBST-2] http://www.stoimen.com/blog/2012/07/03/computer-algorithms-balancing-a-binary-search-tree/

AVL
[AVL-1] https://en.wikipedia.org/wiki/AVL_tree