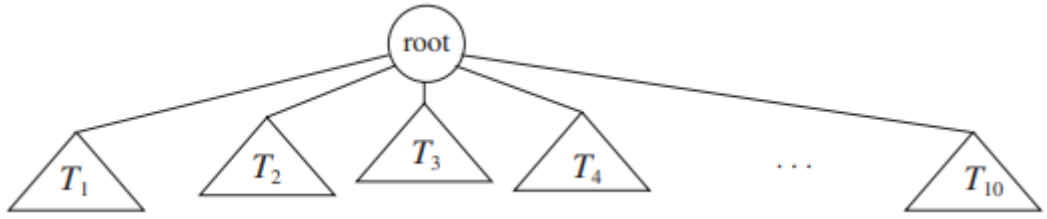
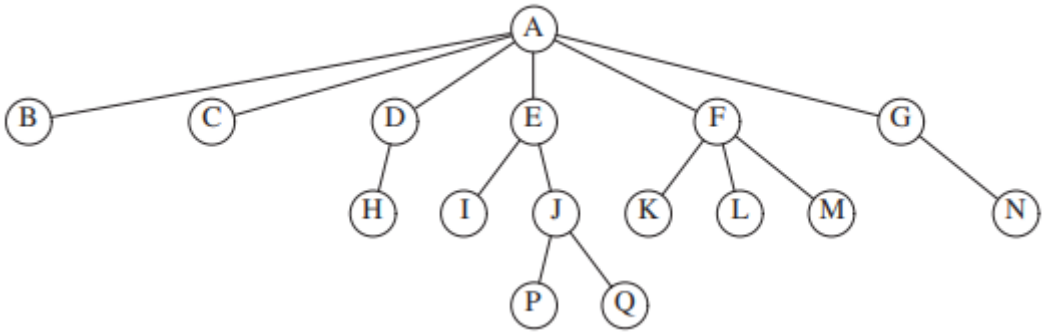
Resumen cuarto capitulo

#Arboles

Para grandes cantidades de entrada, el tiempo de acceso lineal de las listas enlazadas y doblemente enlazadas es demasiado, en este resumen se vera una estructura de datos nueva para la cual el tiempo de ejecución promedio es logarítmica. a la par de dos modificaciones que pueden convenir en casos muy selectos.
La estructura de datos antes mencionada se conoce como árbol de búsqueda binaria. El árbol de búsqueda binaria es la base para la implementación de dos clases de colecciones de biblioteca, set y mapa. Los arboles en general son abstracciones muy útiles en informática, por lo que discutiremos su uso en otras aplicaciones

4.1 Preliminares.

- Un árbol se puede definir de varias maneras. Una forma natural de definir un árbol es de forma recursiva. Un árbol es una colección de nodos. La colección puede estar vacía; de lo contrario, un árbol consta de un nodo distinguido, r, llamado raíz, y cero o más (sub)árboles no vacíos.
- Se dice que la raíz de cada subárbol es hija de r, y r es el padre de la raíz de cada subárbol.
- 

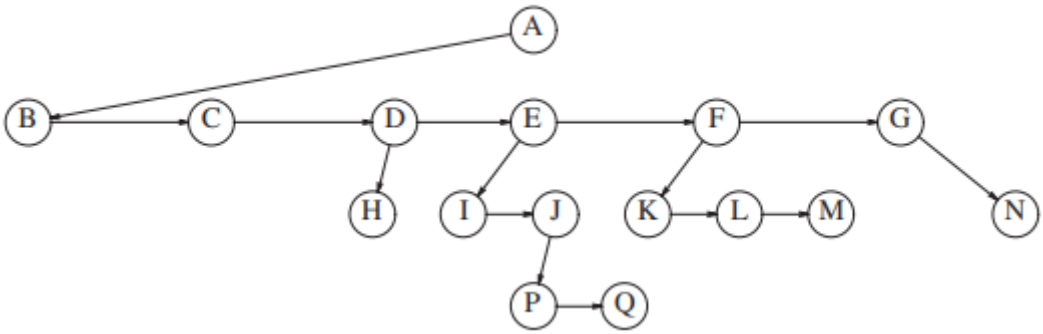
- De la definición recursiva, encontramos que un árbol es una colección de N nodos, uno de los cuales es la raíz, y N − 1 aristas. Que hay N − 1 aristas se desprende del hecho de que cada arista conecta algún nodo con su padre, y cada nodo, excepto la raíz, tiene un padre.
- 

Para cualquier nodo ni, la profundidad de ni es la longitud del camino único desde la raíz hasta ni. Por lo tanto, la raíz está en la profundidad 0. La altura de ni es la longitud del camino más largo desde ni hasta una hoja. Por tanto, todas las hojas están a la altura 0. La altura de un árbol es igual a la altura de la raíz.

4.1.1 implementación de los arboles.

Una forma de implementar un árbol seria tener en cada nodo, además de sus datos un en lace a cada uno de ellos. Sin embargo, dado que el numero de hijos por nodo puede variar muchos y no se sabe de antemano, podría ser inviable hacer que los hijos tengan enlaces directos en los datos.

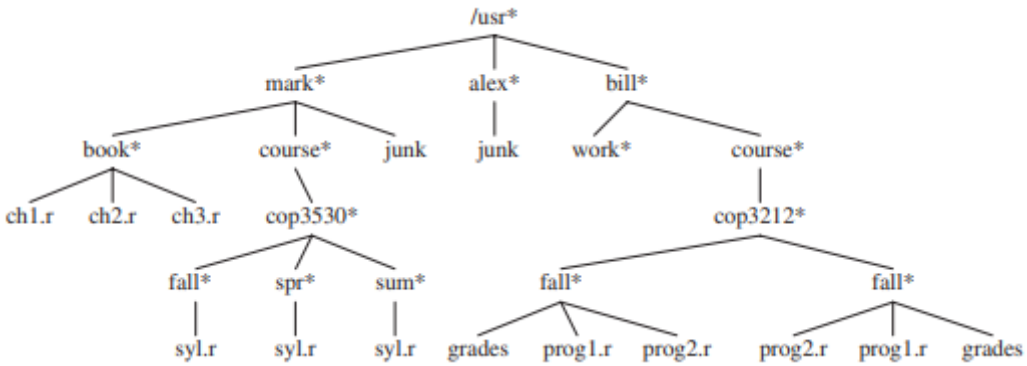
```
struct TreeNode
{
Object element;
TreeNode *firstChild;
TreeNode *nextSibling;
}
```



La solución es simple: mantener los hijos de cada nodo en una lista vinculada de nodos de árbol.
en la imagen anterior, muestra cómo se podría representar un árbol en esta implementación. Horizontal
Las flechas que apuntan hacia abajo son enlaces de primer hijo. Las flechas que van de izquierda a derecha son los siguientes enlaces entre hermanos. Los enlaces nulos no se dibujan porque hay demasiados.

4.1.2 Tree Traversals with an Application.

Hay muchas aplicaciones para los árboles. Uno de los usos populares es la estructura de directorios en muchos sistemas operativos comunes, incluidos UNIX y DOS.



Este sistema de archivos jerárquico es muy popular porque permite a los usuarios organizar sus datos de forma lógica. Además, dos archivos en directorios diferentes pueden compartir el mismo nombre, porque deben tener rutas diferentes desde la raíz y, por lo tanto, tener nombres de ruta diferentes. Un directorio en el sistema de archivos UNIX es simplemente un archivo con una lista de todos sus hijos, por lo que los directorios están estructurados casi exactamente de acuerdo.

```
void FileSystem::listAll( int depth = 0 ) const
{
printName( depth ); // Print the name of the object
if( isDirectory( ) )
for each file c in this directory (for each child)
c.listAll( depth + 1 );
}
```

Con la declaración de tipo anterior.1 De hecho, en algunas versiones de UNIX, si el comando normal para imprimir un archivo se aplica a un directorio, entonces los nombres de los archivos en el directorio se pueden ver en la salida.

Supongamos que queremos enumerar los nombres de todos los archivos en el directorio. Nuestro formato de salida será que los archivos con profundidad di tendrán sus nombres sangrados por tabulaciones di.

La función recursiva listAll debe iniciarse con una profundidad de 0 para indicar que no hay sangría para la raíz. Esta profundidad es una variable de contabilidad interna y no es un parámetro que deba esperarse que conozca una rutina de llamada. Por lo tanto, se proporciona el valor predeterminado de 0 para la profundidad.

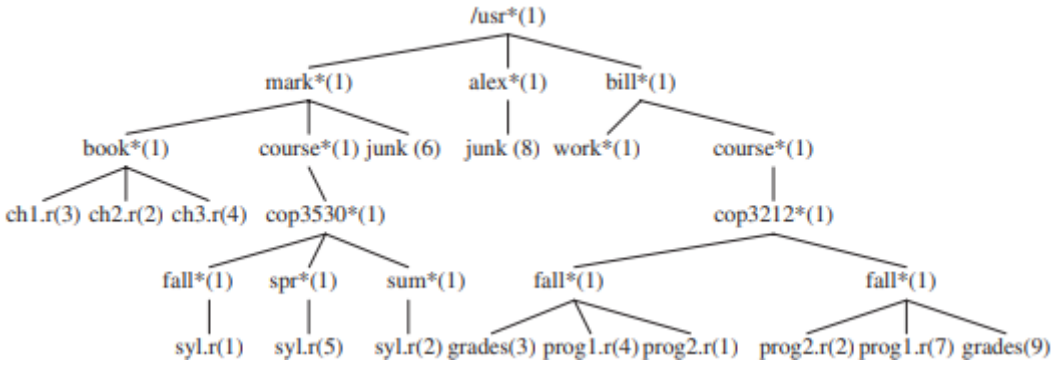
La lógica del algoritmo es sencilla de seguir. El nombre del objeto de archivo se imprime con el número adecuado de pestañas. Si la entrada es un directorio, procesamos todos los elementos

secundarios de forma recursiva, uno por uno. Estos niños están un nivel más abajo y, por lo tanto, es necesario sangrar un espacio adicional.

Otro método común para atravesar un árbol es el recorrido posterior al orden. En un recorrido posterior al orden, el trabajo en un nodo se realiza después de que se evalúen (post) sus hijos.

Dado que los directorios son en sí mismos archivos, también tienen tamaños. Supongamos que queremos calcular el número total de bloques utilizados por todos los archivos del árbol. La forma más natural de hacer esto sería encontrar el número de bloques contenidos en los subdirectorios /usr/mark (30), /usr/alex (9) y /usr/bill (32). El número total de bloques es entonces el total de los subdirectorios (71) más el bloque utilizado por /usr, para un total de 72. El tamaño del método de pseudocódigo en la Figura 4.9 implementa esta estrategia.

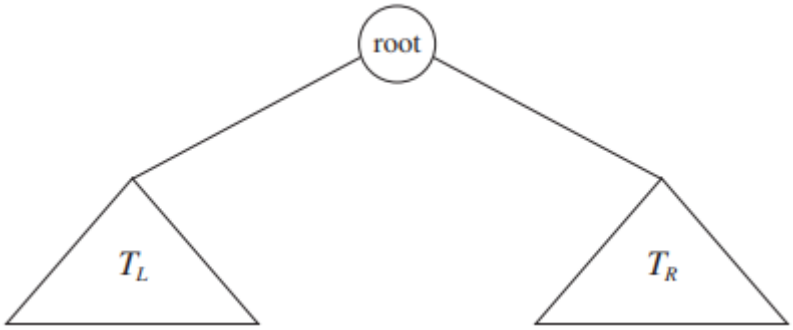
Si el objeto actual no es un directorio, entonces el tamaño simplemente devuelve el número de bloques que utiliza en el objeto actual. De lo contrario, la cantidad de bloques utilizados por el directorio se suma a la cantidad de bloques (recursivamente) encontrados en todos los hijos.




```
int FileSystem::size( ) const
{
int totalSize = sizeOfThisFile( );
if( isDirectory( ) )
for each file c in this directory (for each child)
totalSize += c.size( );
return totalSize;
}
```

4.2 Binary Trees

Un árbol binario es un árbol en el que ningún nodo puede tener más de dos hijos.



Una propiedad de un árbol binario que a veces es importante es que la profundidad de un árbol binario promedio es considerablemente menor que N. Un análisis muestra que la profundidad promedio es O(√N), y eso para un tipo especial de árbol binario, a saber En el árbol de búsqueda binario, el valor promedio de la profundidad es O (logN).

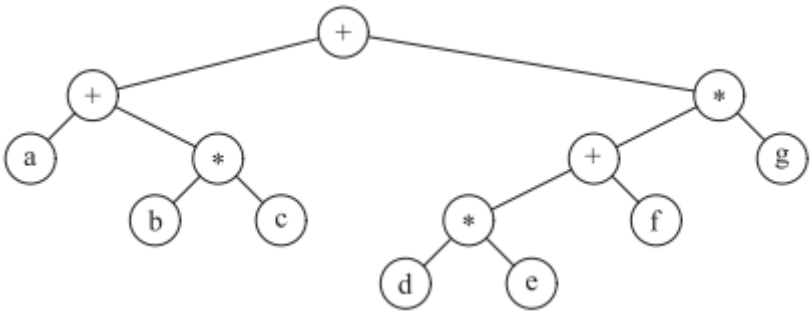
4.2.1 Implementación.

Debido a que un nodo de árbol binario tiene como máximo dos hijos, podemos mantener enlaces directos a ellos. La declaración de nodos de árbol es similar en estructura a la de las listas doblemente enlazadas, en el sentido de que un nodo es una estructura que consta de la información del elemento más dos punteros (izquierdo y derecho) a otros nodos

Los árboles binarios tienen muchos usos importantes no asociados con la búsqueda. Uno de los usos principales de los árboles binarios es el área del diseño de compiladores.

```
struct BinaryNode
{
Object element;
BinaryNode *left;
BinaryNode *right;
};
```

4.2.2 An example: Expression trees.



Las hojas de un árbol de expresión son operandos, como constantes o nombres de variables, y los otros nodos contienen operadores.

Este árbol en particular resulta ser binario, porque todos los operadores son binarios y, aunque este es el caso más simple, es posible que los nodos tengan más de dos hijos. También es posible que un nodo tenga un solo hijo, como es el caso del operador menos unario. Podemos evaluar un árbol de expresión, T, aplicando el operador en la raíz a los valores.

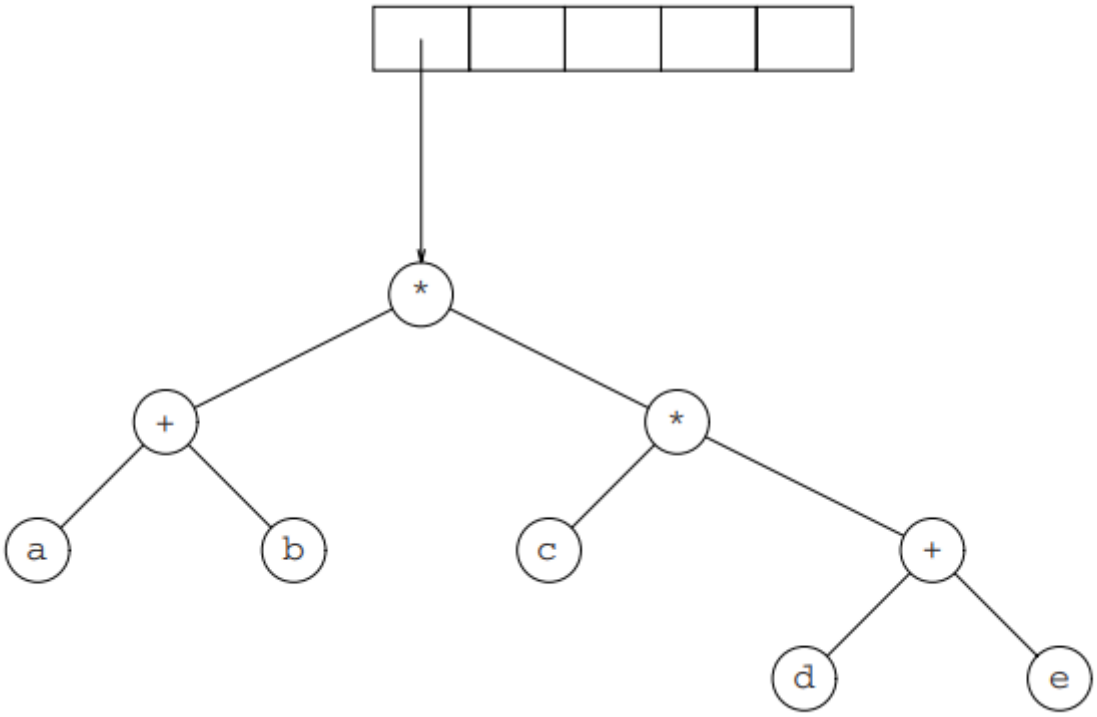
Podemos producir una expresión infija, generando recursivamente una expresión izquierda entre paréntesis, luego imprimiendo el operador en la raíz y finalmente produciendo recursivamente una expresión derecha entre paréntesis. Esta estrategia general (izquierda, nodo, derecha) se conoce como recorrido en orden.

Una estrategia transversal alternativa es imprimir recursivamente el subárbol izquierdo, el subárbol derecho y luego el operador. Esta estrategia transversal se conoce generalmente como recorrido posterior al orden.

Una tercera estrategia transversal es imprimir primero el operador y luego imprimir recursivamente los subárboles izquierdo y derecho.

Constructing an Expression Tree.

Ahora damos un algoritmo para convertir una expresión postfija en un árbol de expresión. Como ya tenemos un algoritmo para convertir infijo en postfijo, podemos generar árboles de expresión a partir de los dos tipos comunes de entrada. Leemos nuestra expresión un símbolo a la vez. Si el símbolo es un operando, creamos un árbol de un nodo y colocamos un puntero en una pila. Si el símbolo es un operador, sacamos (punteros) a dos árboles T1 y T2 de la pila (T1 se saca primero) y formamos un nuevo árbol cuya raíz es el operador y cuyos hijos izquierdo y derecho apuntan a T2 y T1, respectivamente.

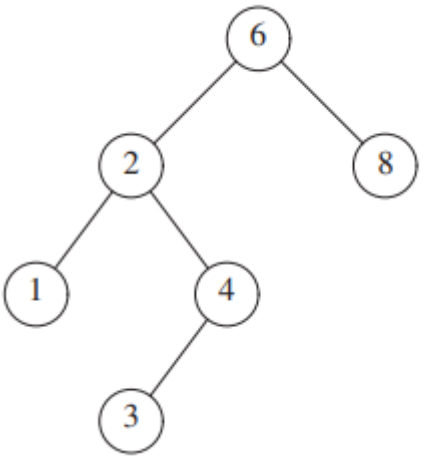


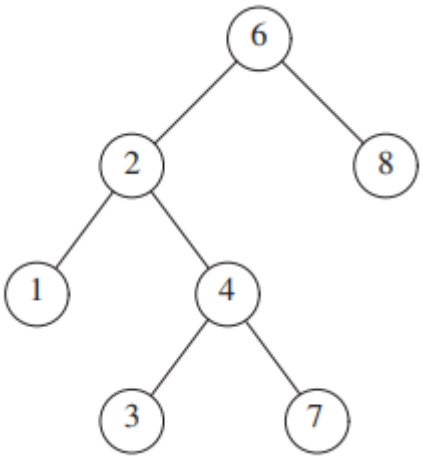
4.3 The Search Tree ADT—Binary Search Trees.

Una aplicación importante de los árboles binarios es su uso en la búsqueda. Supongamos que cada nodo del árbol almacena un elemento. En nuestros ejemplos, asumiremos, por simplicidad, que se trata de números enteros.

La propiedad que convierte un árbol binario en un árbol de búsqueda binaria es que para cada nodo, X, en el árbol, los valores de todos los elementos en su subárbol izquierdo son menores que el elemento en X, y los valores de todos los elementos en su subárbol derecho es mayor que el elemento en X. Esto implica que todos los elementos del árbol se pueden ordenar de alguna manera consistente.

Ahora damos breves descripciones de las operaciones que normalmente se realizan en árboles de búsqueda binarios. Tenga en cuenta que debido a la definición recursiva de árboles, es común escribir estas rutinas de forma recursiva. Debido a que la profundidad promedio de un árbol de búsqueda binario resulta ser O (logN), generalmente no necesitamos preocuparnos por quedarnos sin espacio en la pila.






```
template <typename Comparable>
class BinarySearchTree
{
public:
BinarySearchTree( );
BinarySearchTree( const BinarySearchTree & rhs );
BinarySearchTree( BinarySearchTree && rhs );
~BinarySearchTree( );
const Comparable & findMin( ) const;
const Comparable & findMax( ) const;
bool contains( const Comparable & x ) const;
bool isEmpty( ) const;
void printTree( ostream & out = cout ) const;

void makeEmpty( );
void insert( const Comparable & x );
void insert( Comparable && x );
void remove( const Comparable & x );
BinarySearchTree & operator=( const BinarySearchTree & rhs );
BinarySearchTree & operator=( BinarySearchTree && rhs );

private:
struct BinaryNode
{
Comparable element;
BinaryNode *left;
BinaryNode *right;

BinaryNode( const Comparable & theElement, BinaryNode *lt, BinaryNode *rt )
: element{ theElement }, left{ lt }, right{ rt } { }

BinaryNode( Comparable && theElement, BinaryNode *lt, BinaryNode *rt )
```

```
: element{ std::move( theElement ) }, left{ lt }, right{ rt } { }
};

BinaryNode *root;

void insert( const Comparable & x, BinaryNode * & t );
void insert( Comparable && x, BinaryNode * & t );
void remove( const Comparable & x, BinaryNode * & t );
BinaryNode * findMin( BinaryNode *t ) const;
BinaryNode * findMax( BinaryNode *t ) const;
bool contains( const Comparable & x, BinaryNode *t ) const;
void makeEmpty( BinaryNode * & t );
void printTree( BinaryNode *t, ostream & out ) const;
BinaryNode * clone( BinaryNode *t ) const;
};
```

Varias de las funciones miembro privadas utilizan la técnica de pasar una variable de puntero mediante llamada por referencia. Esto permite que las funciones miembro públicas pasen un puntero a la raíz a las funciones miembro recursivas privadas. Las funciones recursivas pueden luego cambiar el valor de la raíz para que la raíz apunte a otro nodo. Describiremos la técnica con más detalle cuando examinemos el código para insertar.
Ahora podemos describir algunos de los métodos privados.

```
bool contains( const Comparable & x ) const
{
return contains( x, root );
 }

void insert( const Comparable & x )
{
insert( x, root );
}

void remove( const Comparable & x )
{
remove( x, root );
}
```

4.3.1 contains.

Esta operación requiere devolver verdadero si hay un nodo en el árbol T que tiene el elemento X, o falso si no existe dicho nodo. La estructura del árbol hace que esto sea sencillo. Si T está vacío, entonces podemos devolver falso. De lo contrario, si el elemento almacenado en T es X, podemos devolver verdadero. De lo contrario, hacemos una llamada recursiva a un subárbol de T, ya sea izquierdo o derecho, dependiendo de la relación de X con el elemento almacenado en T.

```
bool contains( const Comparable & x, BinaryNode *t ) const
{
if( t == nullptr )
return false;
else if( x < t->element )
return contains( x, t->left );
else if( t->element < x )
return contains( x, t->right );
else
return true;
}
```

Observe el orden de las pruebas. Es fundamental que primero se realice la prueba para un árbol vacío, ya que de lo contrario generaríamos un error de tiempo de ejecución al intentar acceder a un miembro de datos a través de un puntero nullptr. Las pruebas restantes se organizan con el caso menos probable al final. También tenga en cuenta que ambas llamadas recursivas son en realidad recursiones de cola y se pueden eliminar fácilmente con un bucle while. El uso de la recursividad de cola se justifica aquí porque la simplicidad de la expresión algorítmica compensa la disminución de la velocidad y se espera que la cantidad de espacio de pila utilizado sea sólo O(log N).

4.3.2 findMin and findMax

Estas rutinas privadas devuelven un puntero al nodo que contiene los elementos más pequeños y más grandes del árbol, respectivamente. Para realizar un findMin, comience en la raíz y vaya a la izquierda siempre que quede un hijo. El punto de parada es el elemento más pequeño. La rutina findMax es la misma, excepto que la bifurcación es hacia el hijo correcto.

Observe cómo manejamos cuidadosamente el caso degenerado de un árbol vacío. Aunque siempre es importante hacer esto, es especialmente crucial en programas recursivos. También observe que es seguro cambiar t en findMax, ya que solo estamos trabajando con una copia de un puntero.

```
template <typename Object, typename Comparator=less<Object>>
class BinarySearchTree
{
 public:

 private:

BinaryNode *root;
Comparator isLessThan;

bool contains( const Object & x, BinaryNode *t ) const
{
if( t == nullptr )
return false;
else if( isLessThan( x, t->element ) )
return contains( x, t->left );
else if( isLessThan( t->element, x ) )
return contains( x, t->right );
else
return true; // Match
}
};
```

4.3.3 Insert

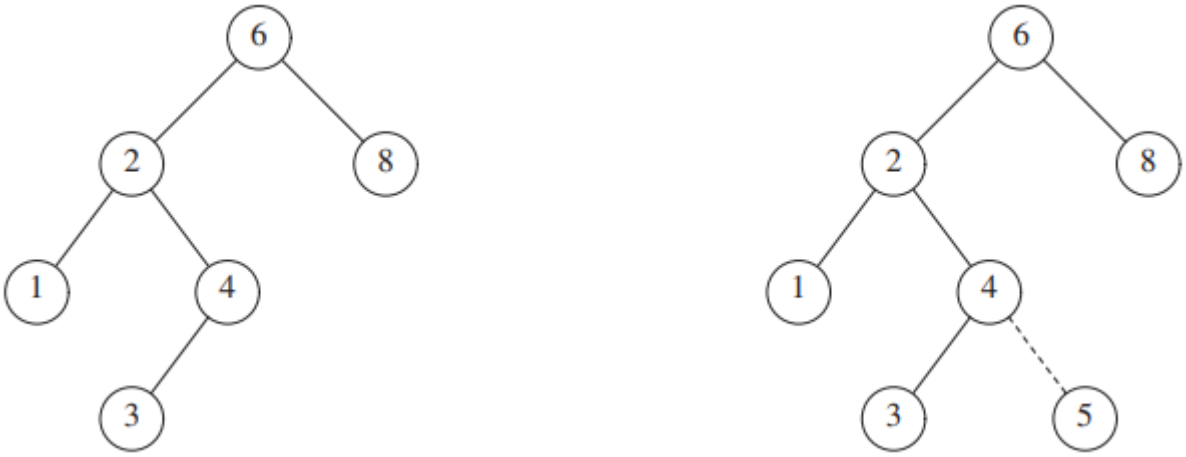
La rutina de inserción es conceptualmente simple. Para insertar X en el árbol T, avance por el árbol como lo haría con un contiene. Si se encuentra X, no haga nada. De lo contrario, inserte una X en el último punto del camino recorrido. En el nodo con el elemento 4, debemos ir a la derecha, pero no hay ningún subárbol, por lo que 5 no está en el árbol y este es el lugar correcto para colocar 5.

Los duplicados se pueden manejar manteniendo un campo adicional en el registro del nodo que indique la frecuencia de aparición. Esto añade algo de espacio extra a todo el árbol, pero es mejor que poner duplicados en el árbol.

```
BinaryNode * findMin( BinaryNode *t ) const
{
if( t == nullptr )
return nullptr;
if( t->left == nullptr )
return t;
```

```
return findMin( t->left );
}
```

```
BinaryNode * findMax( BinaryNode *t ) const
{
if( t != nullptr )
while( t->right != nullptr )
t = t->right;
return t;
}
```



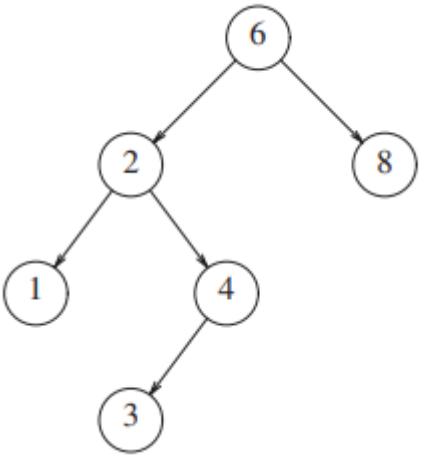
Esta estrategia no funciona si la clave que guía al operador < es solo parte de una estructura más grande. Si ese es el caso, entonces podemos mantener todas las estructuras que tienen la misma clave en una estructura de datos auxiliar, como una lista u otro árbol de búsqueda.

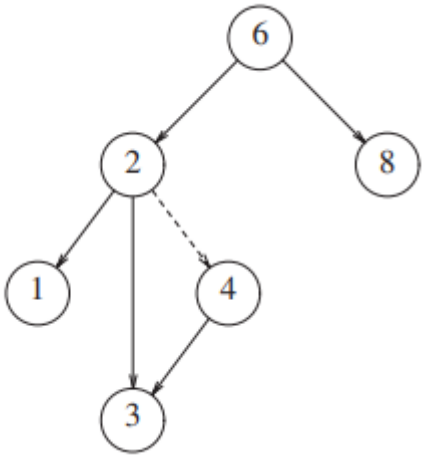
4.3.4 Remove

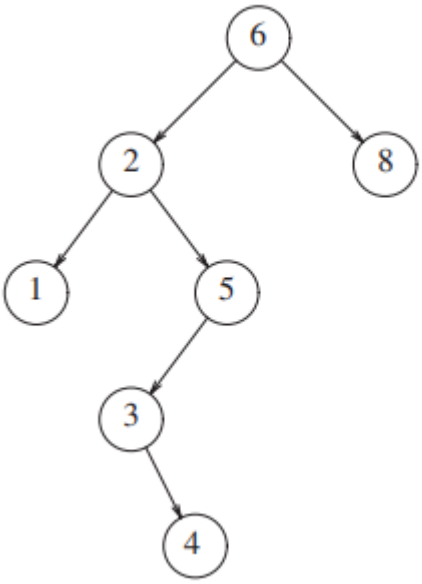
Como suele ocurrir con muchas estructuras de datos, la operación más difícil es la eliminación. Una vez que hemos encontrado el nodo a eliminar, debemos considerar varias posibilidades.

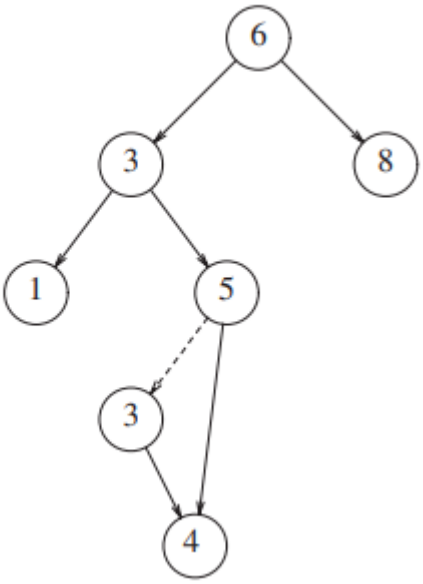
Si el nodo es una hoja, se puede eliminar inmediatamente. Si el nodo tiene un hijo, el nodo se puede eliminar después de que su padre ajuste un enlace para omitir el nodo.

El complicado caso trata de un nodo con dos hijos. La estrategia general es reemplazar los datos de este nodo con los datos más pequeños del subárbol derecho y eliminar recursivamente ese nodo. Debido a que el nodo más pequeño en el subárbol derecho no puede tener un hijo izquierdo, la segunda eliminación es fácil.










```
void remove( const Comparable & x, BinaryNode *&t)
{
        if( t == nullptr )
                return;
        if( x < t->element )
                remove( x, t->left );
        else if( t->element < x )
                remove( x, t->right );
        else if( t->left != nullptr && t->right != nullptr )
        {
                t->element = findMin( t->right )->element;
                remove( t->element, t->right );
        }
        else
        {
```

```
                BinaryNode *oldNode = t;
                t = ( t->left != nullptr ) ? t->left : t->right;
                delete oldNode;
        }
}
```

4.3.5 Destructor and copy constructor.

Como siempre, el destructor llama a makeEmpty. Simplemente llama a la versión recursiva privada. El constructor de copias, sigue el procedimiento habitual, primero inicializando root en nullptr y luego haciendo una copia de rhs. Usamos una función recursiva, llamada clonar.

4.3.6 Average-case Analysis

esperamos que todas las operaciones descritas en esta sección, excepto hacer vacío y copiar, tomen un tiempo O(logN), porque en tiempo constante descendemos un nivel en el árbol, operando así en un árbol que ahora es aproximadamente la mitad de grande. De hecho, el tiempo de ejecución de todas las operaciones (excepto hacer vacío y copiar) es O(d), donde d es la profundidad del nodo que contiene el elemento al que se accede.

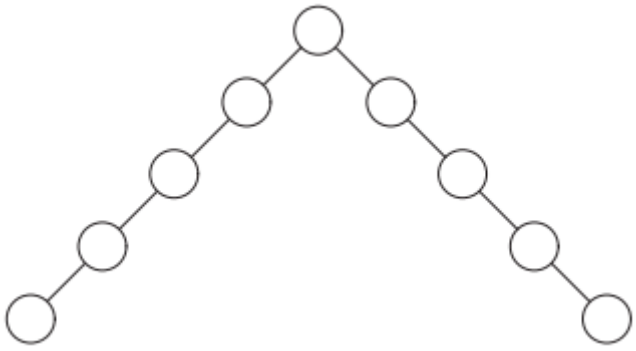
```
1 /**
2 * Destructor for the tree
3 */
4 ~BinarySearchTree( )
5 {
6 makeEmpty( );
7 }
8 /**
9 * Internal method to make subtree empty.
10 */
11 void makeEmpty( BinaryNode * & t )
12 {
13 if( t != nullptr )
14 {
15 makeEmpty( t->left );
16 makeEmpty( t->right );
17 delete t;
18 }
19 t = nullptr;
20 }
```

```
1 /**
2 * Copy constructor
```

```
3 */
4 BinarySearchTree( const BinarySearchTree & rhs ) : root{ nullptr }
5 {
6 root = clone( rhs.root );
7 }
8
9 /**
10 * Internal method to clone subtree.
11 */
12 BinaryNode * clone( BinaryNode *t ) const
13 {
14 if( t == nullptr )
15 return nullptr;
16 else
17 return new BinaryNode{ t->element, clone( t->left ), clone( t->right ) };
18 }
```

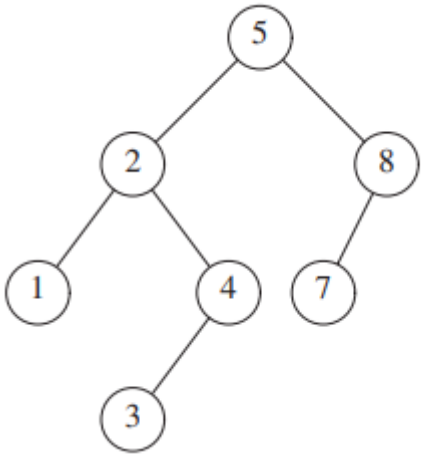
4.4 AVL Trees

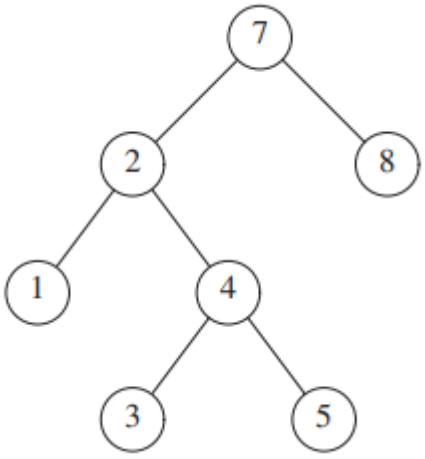
Un árbol AVL es un árbol de búsqueda binario con una condición de equilibrio. La condición de equilibrio debe ser fácil de mantener y garantiza que la profundidad del árbol sea O (logN). La idea más sencilla es exigir que los subárboles izquierdo y derecho tengan la misma altura.



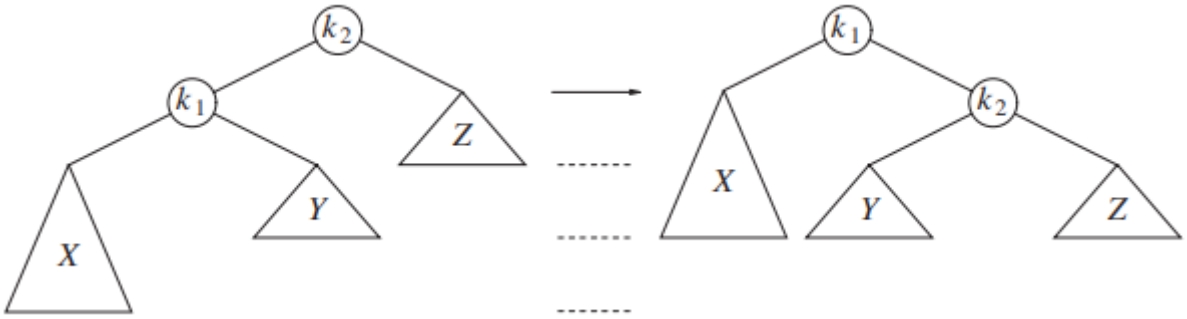
Su condición de equilibrio insistiría en que cada nodo debe tener subárboles izquierdo y derecho de la misma altura. Si la altura de un subárbol vacío se define como −1 (como es habitual), entonces sólo los árboles perfectamente equilibrados de 2k − 1 nodos satisfarían este criterio. Por lo tanto, aunque esto garantiza árboles de poca profundidad, la condición de equilibrio es demasiado rígida para ser útil y necesita relajarse.

Por lo tanto, todas las operaciones del árbol se pueden realizar en tiempo O(logN), excepto posiblemente la inserción y eliminación. Cuando realizamos una inserción, necesitamos actualizar toda la información de equilibrio de los nodos en la ruta de regreso a la raíz, pero la razón por la que la inserción es potencialmente difícil es que insertar un nodo podría violar la propiedad del árbol AVL.

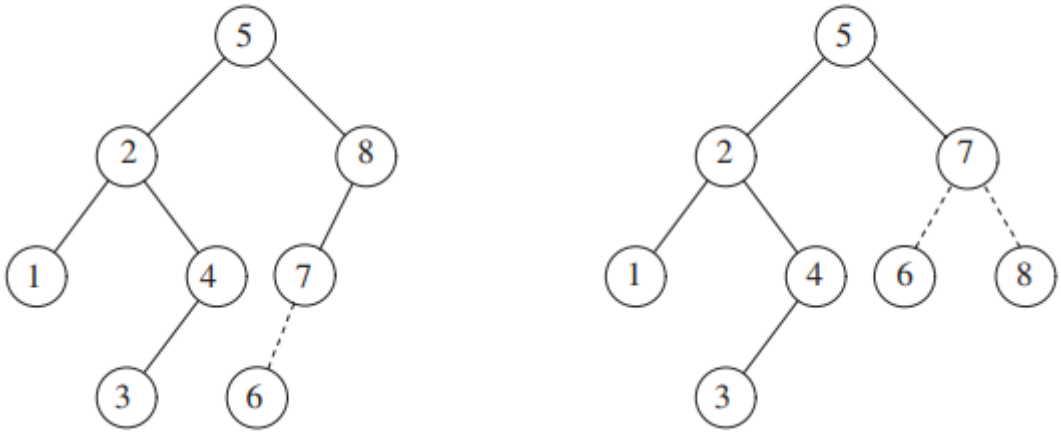




4.4.1 single rotation

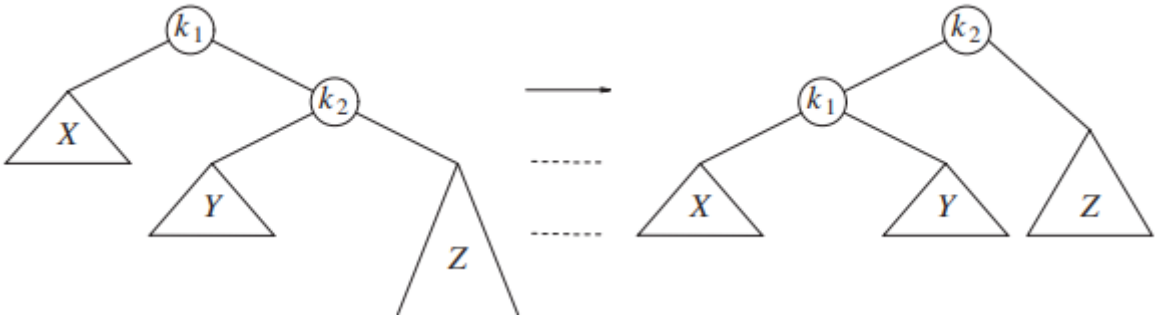


El nodo k2 viola la propiedad de equilibrio AVL porque su subárbol izquierdo es dos niveles más profundo que su subárbol derecho (las líneas discontinuas en el medio del diagrama marcan los niveles). La situación descrita es el único escenario posible del caso 1 que permite a k2 satisfacer la propiedad AVL antes de una inserción pero violarla después. El subárbol X ha crecido a un nivel adicional, lo que hace que sea exactamente dos niveles más profundo que Z. Y no puede estar en el mismo nivel que el nuevo X porque entonces k2 habría estado desequilibrado antes de la inserción, e Y no puede estar en el mismo nivel que Z porque entonces k1 sería el primer nodo en el camino hacia la raíz que violaba la condición de equilibrio AVL.
Para reequilibrar idealmente el árbol, nos gustaría mover X hacia arriba un nivel y Z hacia abajo. Tenga en cuenta que esto es en realidad más de lo que requeriría la propiedad AVL. Para hacer esto, reagrupamos los nodos de rango en un árbol equivalente.



4.4.2 Double Rotation.

El algoritmo descrito anteriormente tiene un problema, no funciona para los casos 2 o 3. El problema es que el subárbol Y es demasiado profundo y una sola rotación no lo hace menos profundo. La doble rotación que resuelve el problema



El hecho de que al subárbol se le haya insertado un elemento garantiza que no está vacío. Por tanto, podemos suponer que tiene una raíz y dos subárboles. En consecuencia, el El árbol puede verse como cuatro subárboles conectados por tres nodos. Como sugiere el diagrama, exactamente uno del árbol B o C es dos niveles más profundo que D (a menos que todos estén vacíos), pero no podemos
estar seguro de cuál. Resulta que no importa; tanto B como C se dibujan en 11 2 niveles por debajo de D.

```
1 struct AvlNode
2 {
3 Comparable element;
4 AvlNode *left;
5 AvlNode *right;
6 int height;
7
8 AvlNode( const Comparable & ele, AvlNode *lt, AvlNode *rt, int h = 0 )
```

```
9 : element{ ele }, left{ lt }, right{ rt }, height{ h } { }
10
11 AvlNode( Comparable && ele, AvlNode *lt, AvlNode *rt, int h = 0 )
12 : element{ std::move( ele ) }, left{ lt }, right{ rt }, height{ h } { }
13 };
```

```
1 /**
2 * Return the height of node t or -1 if nullptr.
3 */
4 int height( AvlNode *t ) const
5 {
6 return t == nullptr ? -1 : t->height;
7 }
```

```
1 /**
2 * Internal method to insert into a subtree.
3 * x is the item to insert.
4 * t is the node that roots the subtree.
5 * Set the new root of the subtree.
6 */
7 void insert( const Comparable & x, AvlNode *&t)
8 {
9 if( t == nullptr )
10 t = new AvlNode{ x, nullptr, nullptr };
11 else if( x < t->element )
12 insert( x, t->left );
13 else if( t->element < x )
14 insert( x, t->right );
15
16 balance( t );
17 }
18
19 static const int ALLOWED_IMBALANCE = 1;
20
21 // Assume t is balanced or within one of being balanced
22 void balance( AvlNode *&t)
23 {
24 if( t == nullptr )
25 return;
26
27 if( height( t->left ) - height( t->right ) > ALLOWED_IMBALANCE )
28 if( height( t->left->left ) >= height( t->left->right ) )
29 rotateWithLeftChild( t );
30 else
31 doubleWithLeftChild( t );
```

```
32 else
33 if( height( t->right ) - height( t->left ) > ALLOWED_IMBALANCE )
34 if( height( t->right->right ) >= height( t->right->left ) )
35 rotateWithRightChild( t );
36 else
37 doubleWithRightChild( t );
38
39 t->height = max( height( t->left ), height( t->right ) ) + 1;
40 }
```

#Conclusión

Es un tema fácil de entender en teoría pero en código se vuelve revoltoso y complicado, aun así al tener varias formas de hacer los arboles puedes aprender a hacer el que mas te sirva y eso me gusta la idea de poder acomodarlos a tus necesidades

#opinion_critica

mientras mas temas vemos mas fácil es de entender el libro en este caso no se necesitaron horas y horas para leer un solo párrafo y eso me hace sentir muy bien

#Referencias

En este caso me quede solo con la información del libro.