****

**Trabajo de Investigación**

**CIT2000 Estructuras de Datos**

**Semestre 2019-01**

Profesor: Mauricio Hidalgo

Ayudante: Martín Saavedra

Sección: CIT2000-03

Alumno: Ze Hui Fu Zeng

Índice

[Lista enlazada simple 3](#_Toc12742813)

[**Inserción** 3](#_Toc12742814)

[**Búsqueda** 4](#_Toc12742815)

[**Eliminación** 4](#_Toc12742816)

[Árboles AVL 5](#_Toc12742817)

[**Inserción** 7](#_Toc12742818)

[**Búsqueda** 7](#_Toc12742819)

[**Eliminación** 7](#_Toc12742820)

[Árbol B 8](#_Toc12742821)

[**Búsqueda** 8](#_Toc12742822)

[**Inserción** 9](#_Toc12742823)

[**Eliminación** 9](#_Toc12742824)

[Heap máximo 11](#_Toc12742825)

[**Inserción** 12](#_Toc12742826)

[**Eliminación** 12](#_Toc12742827)

[**Búsqueda** 12](#_Toc12742828)

[Árbol digital 13](#_Toc12742829)

[**Búsqueda** 13](#_Toc12742830)

[**Inserción** 13](#_Toc12742831)

[**Eliminación** 13](#_Toc12742832)

[Tabla de rendimiento 14](#_Toc12742833)

[Resultados 14](#_Toc12742834)

# **Lista enlazada simple**

Las listas enlazadas son estructuras de datos semejantes a los arreglos salvo que el acceso a un elemento no se hace mediante un índice sino mediante un puntero.

La asignación de memoria es hecha durante la ejecución.

En una lista los elementos son contiguos en lo que concierne al enlazado.

En cambio, mientras que en un arreglo los elementos están contiguos en la memoria, en una lista los elementos están dispersos. El enlace entre los elementos se hace mediante un puntero. En realidad, en la memoria la representación es aleatoria en función del espacio asignado.

El puntero siguiente del último elemento tiene que apuntar hacia None (el fin de la lista).

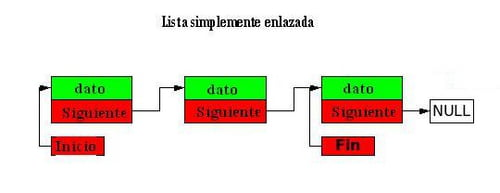
Para acceder a un elemento, la lista es recorrida comenzando por el inicio, el puntero Siguiente permite el cambio hacia el próximo elemento.

El desplazamiento se hace en una sola dirección, del primer al último elemento.

Si deseas desplazarte en las dos direcciones (hacia delante y hacia atrás) deberás utilizar las listas doblemente enlazadas.

Creación de nodo: Los nodos de la lista tendrán un elemento y un puntero que indica el siguiente nodo, también tendrá la función de retornar su elemento.

Clase lista simple: La lista tendrá dos nodos centinela para poder identificar al primero y último de la lista. Sus respectivas funciones para retornar el primero, último y vacío.



Lista simple

## **Inserción**

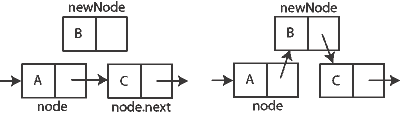
Para añadir un elemento a la lista se presentan dos casos inserción al inicio de la lista, la inserción al final de la lista y la inserción entre la lista.

Siempre hay que verificar si la lista está vacía, ya que si esta lo está el primero y el último serían el mismo nodo que se quiere insertar.

Añadir nodo al final (Matemática asociada O(n)): Se revisa si la lista está vacía, si esta lo está se procede a igualar el primero y último al nodo nuevo, de lo contrario se igualará el siguiente del último al nodo nuevo y el último será el nodo nuevo.

Añadir nodo al inicio (Matemática asociada O(1)): Se revisa si la lista está vacía, si esta lo está se procede a igualar el primero y último al nodo nuevo, de lo contrario se igualará el nodo nuevo siguiente a el primero y el primero será el nodo nuevo.

Añadir nodo intermedio (Matemática asociada O(n2)): Se revisa si la lista está vacía, si esta lo está se procede a igualar el primero y último al nodo nuevo, de lo contrario se crean dos nodos temporales (t1 y t2) uno con el primero y el segundo con el siguiente del primero, se recorre la lista con los dos temporales hasta llegar a la posición deseada. Luego se iguala el siguiente de t1 a el nodo nuevo y el siguiente del nodo nuevo a t2.



Añadir nodo

## **Búsqueda**

(Matemática asociada O(n))

Se iguala a un temporal el primero de la lista y se hace un ciclo que hasta que no encuentre lo solicitado o que el temporal no sea None, mientras el temporal no sea igual a lo solicitado, el temporal se igualará al temporal siguiente.

## **Eliminación**

Para eliminar un nodo se presentan tres casos, eliminar al principio, al final de la lista o un nodo en específico.

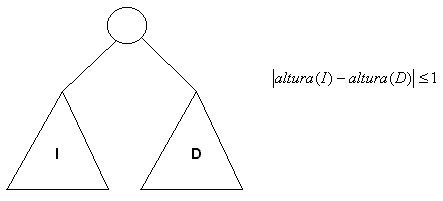
Eliminación del primero (Matemática asociada O(1)): Primero se verifica si la lista está vacía, si lo está se advierte del caso. Si la lista tiene sólo un nodo, se iguala el primero y el último a None. Si tiene más de un nodo, se crea un temporal igualado al primer nodo y se iguala el primero al siguiente primero, luego el temporal se iguala a None.

Eliminación del último (Matemática asociada O(n)): Primero se verifica si la lista está vacía, si lo está se advierte del caso. Si la lista tiene sólo un nodo, se iguala el primero y el último a None. Si tiene más de un nodo, se crea un temporal (t1) igualado al primer nodo y un ciclo que no para hasta eliminar el último nodo, dentro del ciclo si el t1 siguiente no es el último, el t1 será igual al t1 siguiente, de lo contrario se crea otro temporal (t2) que se igualará al último, posteriormente el último será igual a t1 y t2 será None. Luego el ciclo termina.

Eliminación de cualquier nodo (Matemática asociada O(n2)): Primero se verifica si la lista está vacía, si lo está se advierte del caso. Si la lista tiene sólo un nodo, se verifica si este es el solicitado, si lo es se elimina, de lo contrario se le advierte. Si tiene más de un nodo se crea un temporal (t1) igualado a el primero para recorrer la lista con un ciclo que no parará hasta encontrarlo y eliminarlo, se elimina de siguiente forma, se crea un temporal (t2) igualado al t1 siguiente (el que se quiere eliminar), t1 siguiente se iguala a t2 siguiente y t2 igual a None. Luego termina el ciclo.

# **Árboles AVL**

Los árboles AVL son árboles binarios que están siempre balanceados de tal modo que para todos los nodos, la altura de la rama izquierda más larga no difiere en más de una unidad de la altura de la rama derecha más larga o viceversa, por lo tanto la altura de un nodo perteneciente a un árbol AVL siempre diferirá de 1, 0 o -1.

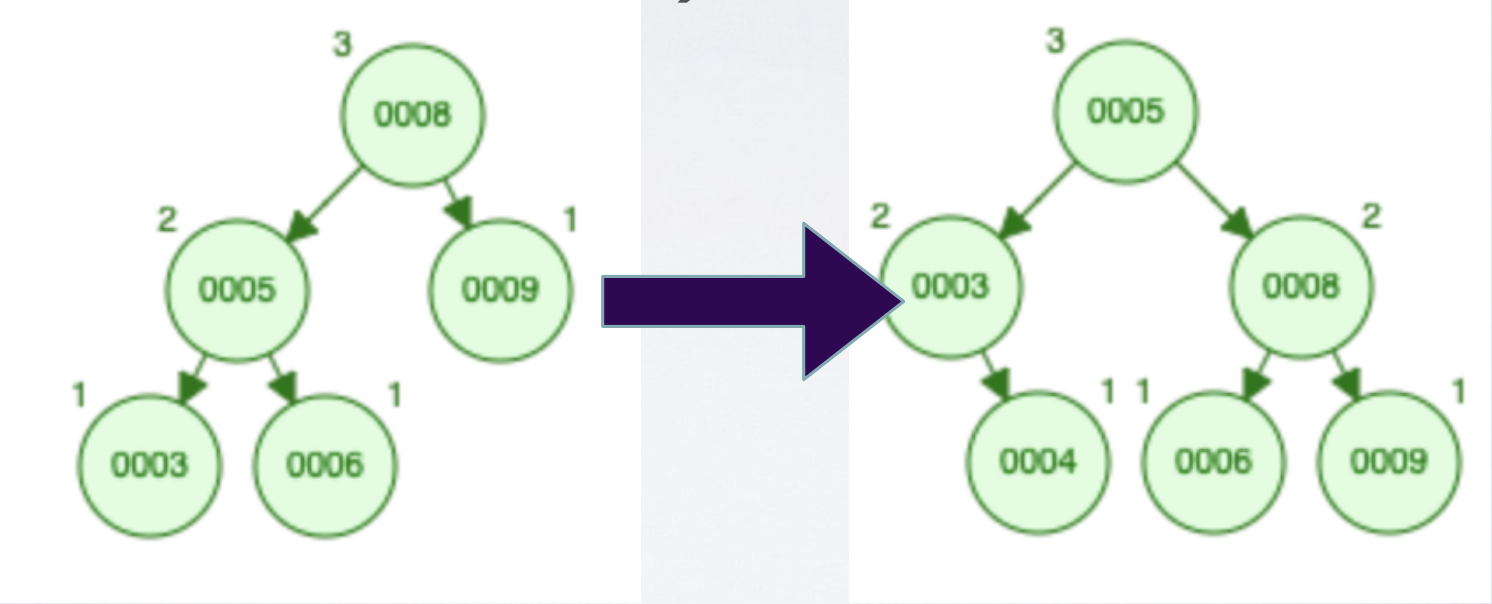


Árboles AVL

Este árbol se balancea por medio de rotaciones simples o doble, estas rotaciones pueden ser hacia la izquierda o hacia la derecha.

Rotaciones simples: Esto ocurre cuando hay un desbalance después de insertar un nodo externo.

* Derecha (inserción de nodo externo a la izquierda): Se usa como pivote el padre desbalanceado más próximo al nodo insertado, el hijo izquierdo se convierte en el padre, el padre se convierte en su hijo derecho y el hijo derecho del hijo izquierdo se convierte en el hijo izquierdo del padre.



Al insertar 4

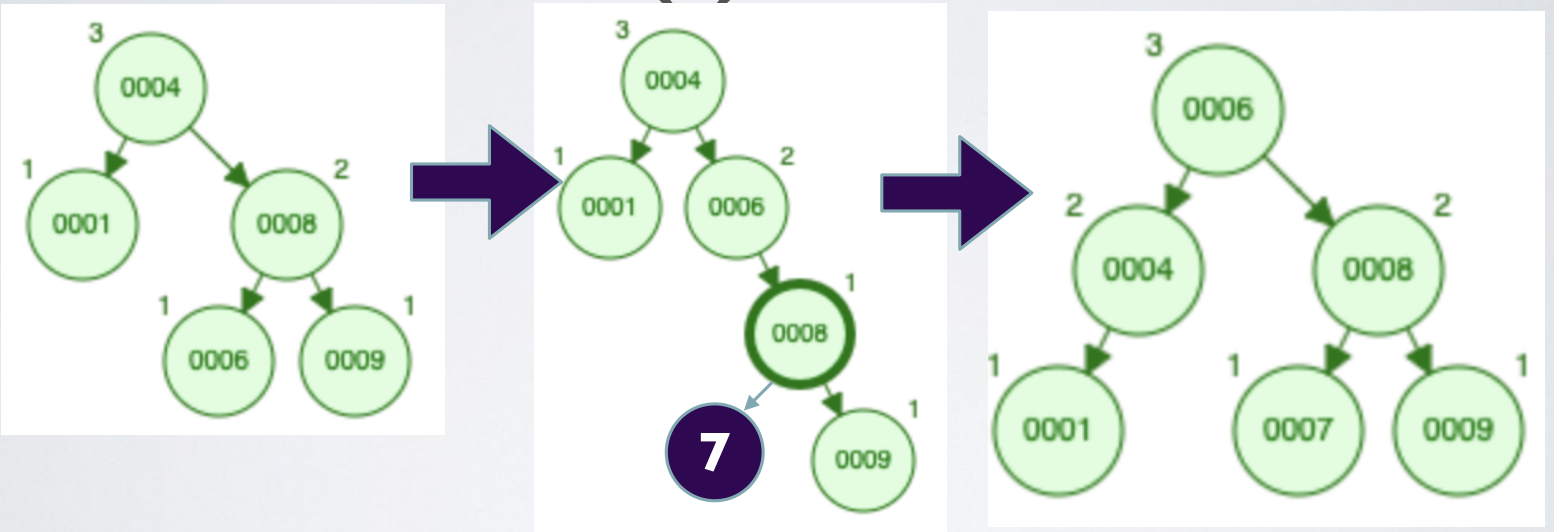
* Izquierda (inserción de nodo externo a la derecha): Se usa como pivote padre desbalanceado más próximo al nodo insertado, el hijo derecho se convierte en el padre, el padre se convierte en su hijo izquierdo y el hijo izquierdo del hijo derecho se convierte en el hijo izquierdo del padre.



Al insertar 9

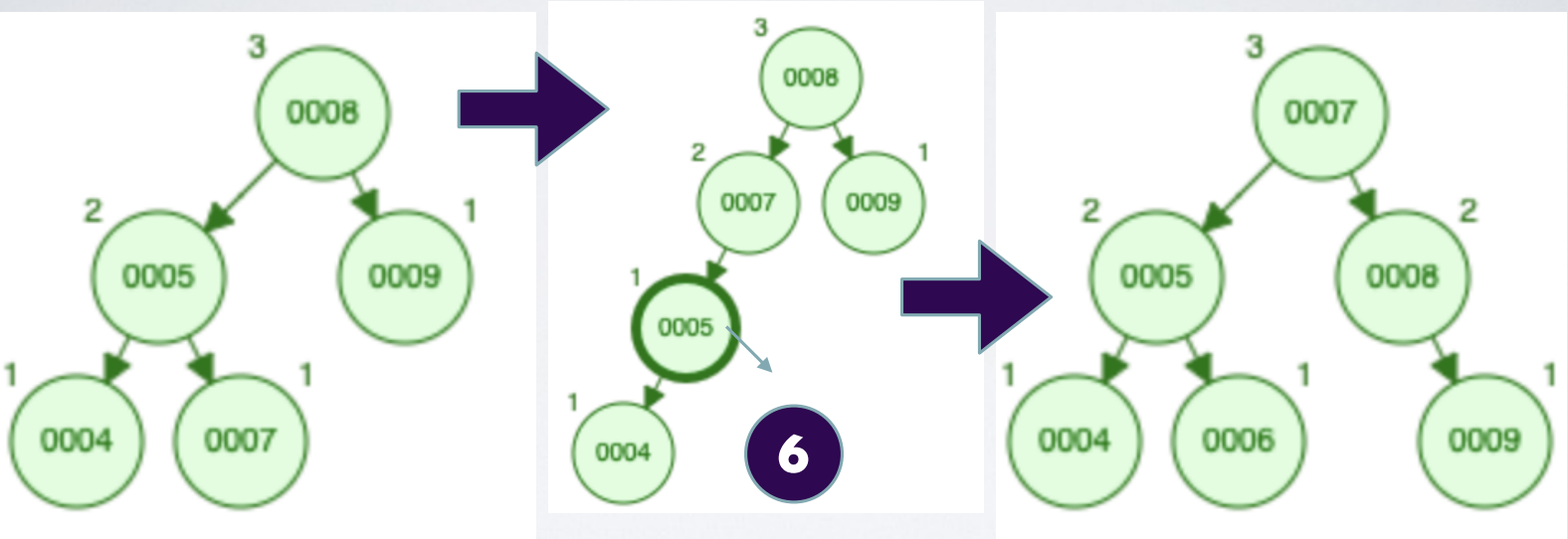
Rotaciones dobles: Esto ocurre cuando hay un desbalance después de insertar un nodo interno.

* Derecha (inserción de nodo interno a la derecha): Ocurre cuando se inserta un nodo que desbalancea en el subárbol derecho. Se debe hacer una rotación simple hacia la derecha usando como pivote al padre del nodo que se insertó, luego una rotación simple hacia la izquierda usando como pivote al padre desbalanceado más cercano al nodo insertado.



Al insertar 7

* Izquierda (inserción de nodo interno a la derecha): Ocurre cuando se inserta un nodo que desbalancea en el subárbol izquierdo. Se debe hacer una rotación simple hacia la izquierda usando como pivote al padre del nodo que se insertó, luego una rotación simple hacia la derecha usando como pivote al padre desbalanceado más cercano al nodo insertado.



Al insertar 6

## **Inserción**

(Matemática asociada O(log n))

Los árboles AVL tiene el mismo modo de inserción que los árboles binarios, o sea que un nodo puede tener como máximo dos hijos (derecho e izquierdo). Si el árbol está vacío, el nodo a insertar será la raíz del árbol. Si el árbol tiene más de un nodo se tendrá que compara la raíz del árbol o de los subárboles si es mayor o menor que su raíz, si ésta es menor será su hijo izquierdo, de lo contrario será su hijo derecho.

## **Búsqueda**

(Matemática asociada O(log n))

Se compara cada raíz del árbol o subárbol con el nodo a buscar, comparando si es igual, menor o mayor a la raíz del árbol o subárbol, si este es igual se ha encontrado, si es menor baja y compara con el subárbol izquierdo, si es mayor baja y compara con el subárbol derecho, recursivamente hasta encontrar lo solicitado o que el próximo subárbol sea None.

## **Eliminación**

(Matemática asociada O(log n))

Para la eliminación existen 3 casos:

* Si el nodo a eliminar es una hoja (subárbol con hijos None) se elimina.
* Si el nodo a eliminar solamente tiene un hijo, este se reemplaza por el nodo.
* Si el nodo a eliminar tiene dos hijos, se tendrá que elegir entre los dos, si se elige el izquierdo el nodo a eliminar se reemplazará por el nodo más grande de ese subárbol, si se elige el derecho el nodo a eliminar se reemplazará por el nodo más chico de ese subárbol.

Para todos estos casos se tendrá que verificar si el árbol se ha desbalanceado, si se da el caso se tendrá que balancear con las rotaciones mencionadas.

# **Árbol B**

Los árboles B son árboles de búsqueda de auto-equilibrio. En la mayoría de los otros árboles de búsqueda de equilibrio automático (como AVL), se supone que todo está en la memoria principal. Para comprender el uso de B-Trees, debemos pensar en la gran cantidad de datos que no caben en la memoria principal. Cuando el número de llaves es alto, los datos se leen del disco en forma de bloques.

La idea principal de usar B-Trees es reducir la cantidad de accesos al disco. La mayoría de las operaciones del árbol (búsqueda, inserción, eliminación, Etc.) requieren O (h) accesos de disco donde h es la altura del árbol.

La altura de los B-Trees se mantiene baja al colocar la cantidad máxima de llaves posibles en un nodo B-Tree. En general, un tamaño de nodo B-Tree se mantiene igual al tamaño de bloque de disco. Como h es bajo para B-Tree, los accesos totales al disco para la mayoría de las operaciones se reducen significativamente en comparación con los árboles de búsqueda binaria equilibrados como AVL.

Propiedades

1) Todas las hojas están en el mismo nivel.

2) Un árbol B se define por el término grado mínimo 't'. El valor de t depende del tamaño del bloque de disco.

3) Cada nodo, excepto la raíz, debe contener al menos t-1 llaves. La raíz puede contener un mínimo de 1 llave.

4) Todos los nodos (incluida la raíz) pueden contener como máximo 2t - 1 llaves.

5) El número de hijos de un nodo es igual al número de llaves en él más 1.

6) Todas las llaves de un nodo se ordenan en orden creciente. El elemento secundario entre dos llaves k1 y k2 contiene todas las teclas en el rango de k1 y k2.

7)Los árboles B crecen y se encogen desde la raíz.

8) Al igual que otros árboles de búsqueda binaria equilibrados, la complejidad del tiempo para buscar, insertar y eliminar es **O (Log n)**.

## **Búsqueda**

Es similar a la búsqueda en el árbol binario de búsqueda.

Partimos de la raíz y recursivamente descendemos. Para cada nodo no hoja visitado, si el nodo tiene la llave, simplemente retornamos el nodo. De lo contrario, recurrimos al hijo apropiado (el hijo justo antes de la primera llave mayor del valor solicitado) del nodo. Si alcanzamos un nodo hoja y no encontramos lo solicitado en el nodo, retornamos None.

## **Inserción**

Siempre se inserta una nueva llave en el nodo de hoja. Al igual que árboles binarios, comenzamos desde la raíz y descendemos (derecha si es mayor o izquierda si es menor) hasta llegar a un nodo hoja. Una vez que llegamos a un nodo hoja, insertamos la llave en ese nodo. A diferencia de los árboles binarios, tenemos un rango predefinido en el número de llaves que puede contener un nodo. Entonces, antes de insertar una llave en el nodo, verificaremos si está lleno o no, si este lo está, lo dividimos para crear espacio.

Se divide de la siguiente forma, en la figura 1 el hijo Y de X está lleno, por lo que se divide en dos nodos Y y Z. Note que la operación splitChild sube una llave y esta es la razón por la que los B-Trees crecen hacia arriba, a diferencia de los árboles binarios que crecen hacia abajo.

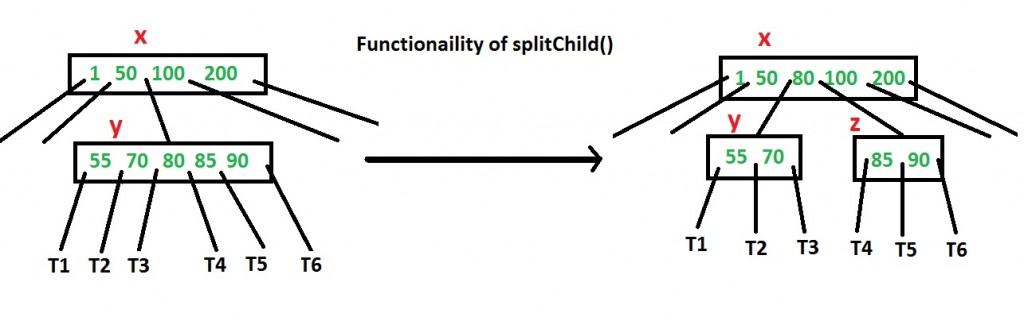


Figura 1

## **Eliminación**

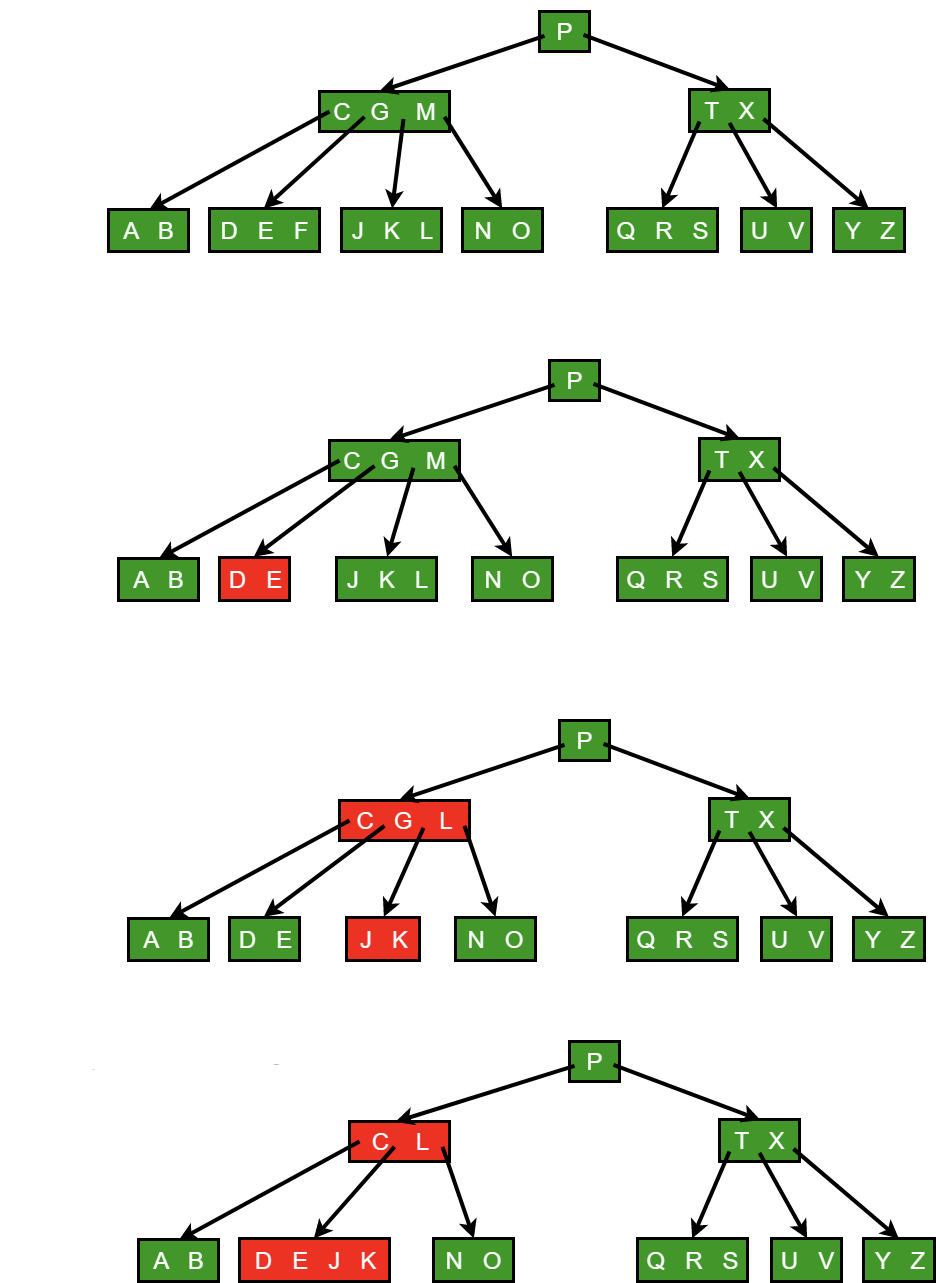
Podremos eliminar una llave de cualquier nodo, sea hoja o nodo interno, cuando eliminemos un nodo interno, tendremos que reorganizar los nodos hijos.

Debemos asegurarnos de que la eliminación no viole las propiedades del árbol-B, para esto nos tendremos que asegurar que el nodo no sea demasiado pequeño durante la eliminación.

En la eliminación se podría hacer una copia de seguridad si un nodo (que no sea la raíz) a lo largo de la ruta por donde se debe eliminar la llave tiene el número mínimo de llaves.

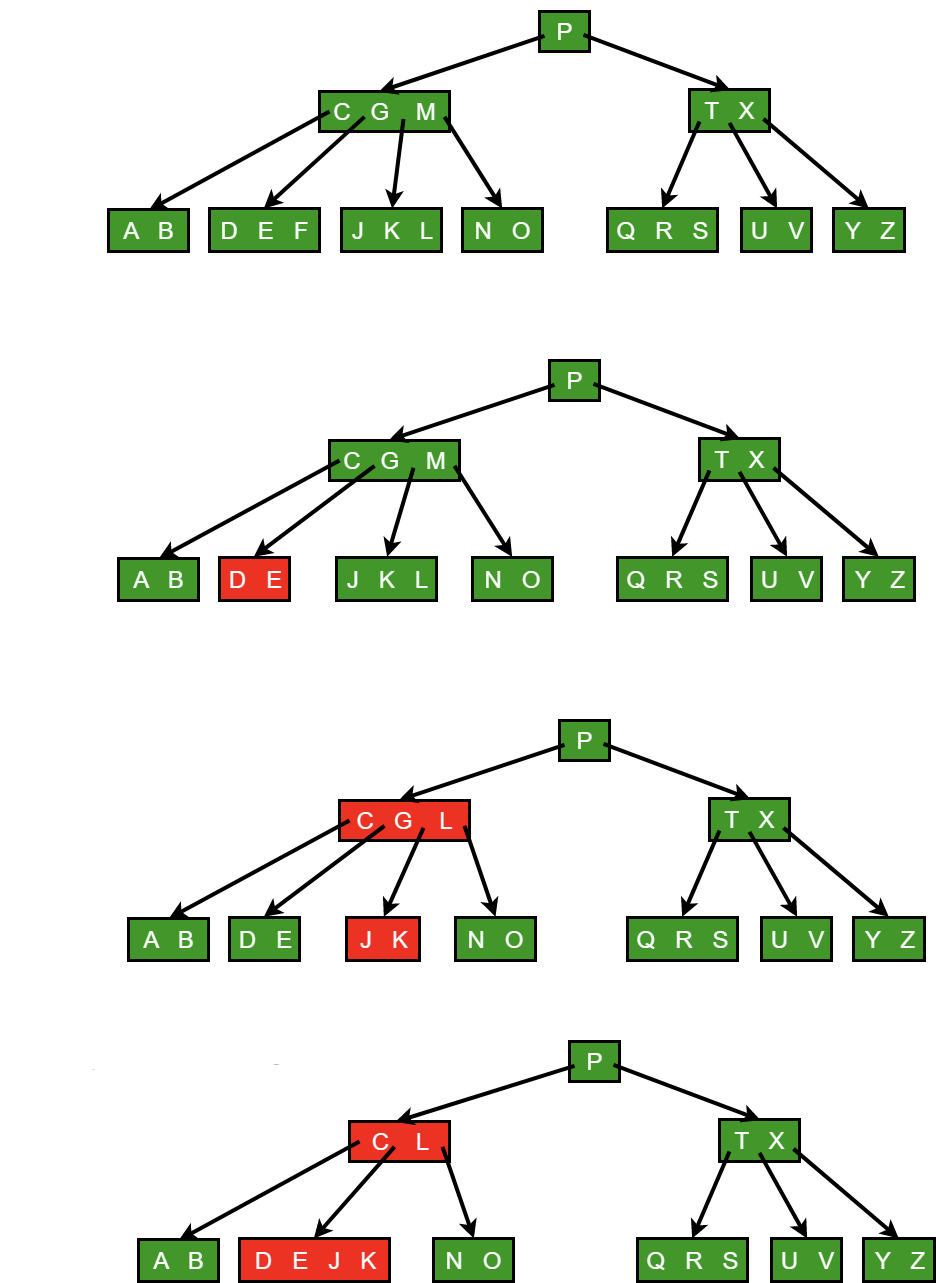
El procedimiento de eliminación de la llave K perteneciente del subárbol con raíz X. Este proceso asegura que siempre que se llame a sí mismo de forma recursiva al nodo X, el número de llaves en X será mayor o igual al mínimo de llaves.

Casos de eliminación:



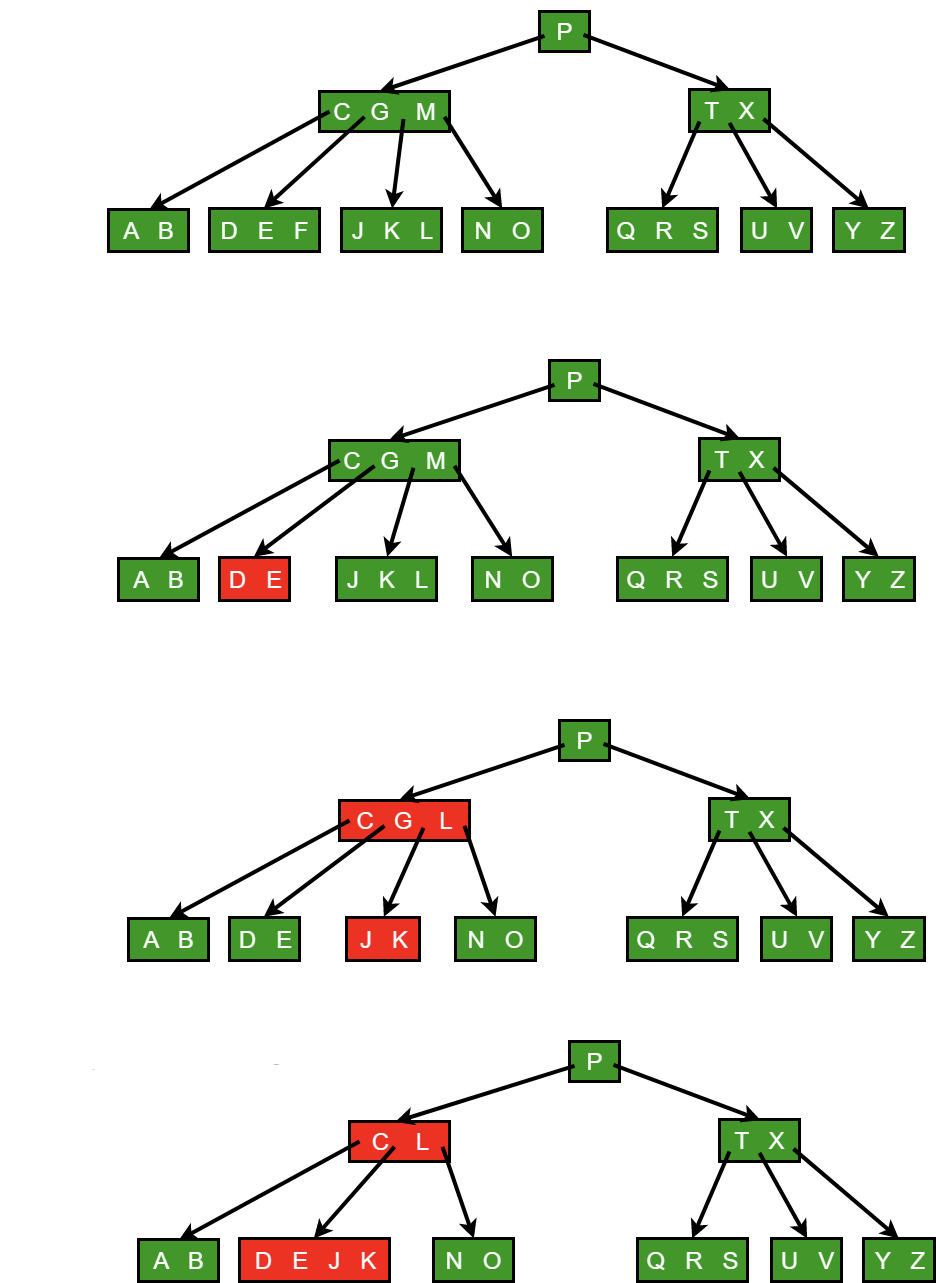
Árbol inicial

1. Si la llave K está en el nodo X y X es una hoja, borrar la llave K de X.



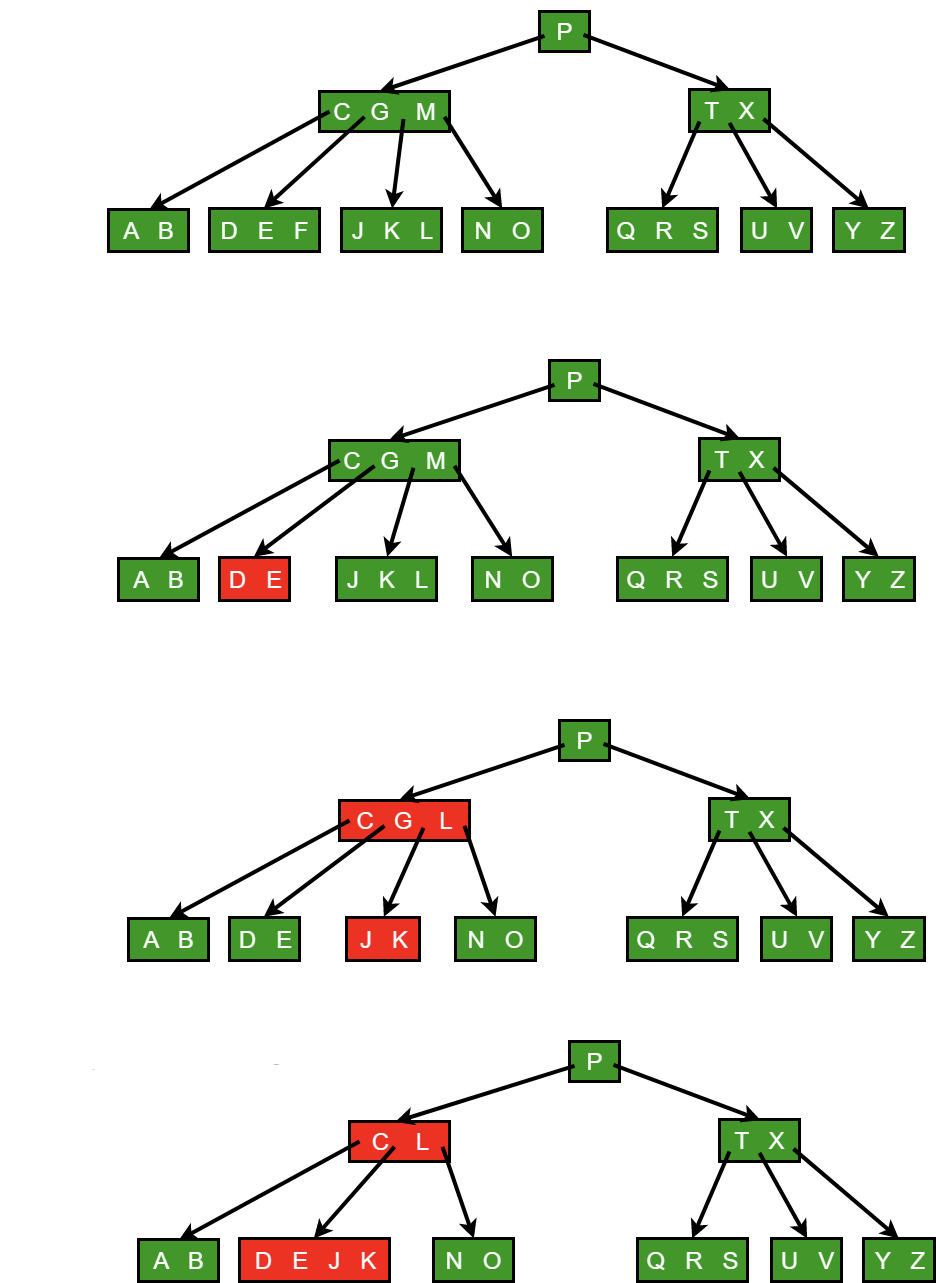
Eliminar F, caso 1

1. Si la llave K está en el nodo X y X en un nodo interno, hacer lo siguiente
2. Si el hijo Y que precede K en el nodo X y este tiene al menos t llaves, entonces se encuentra un predecesor K0 de K en la raíz del subárbol en Y. Eliminar K0 y reemplazar K por K0 en X.



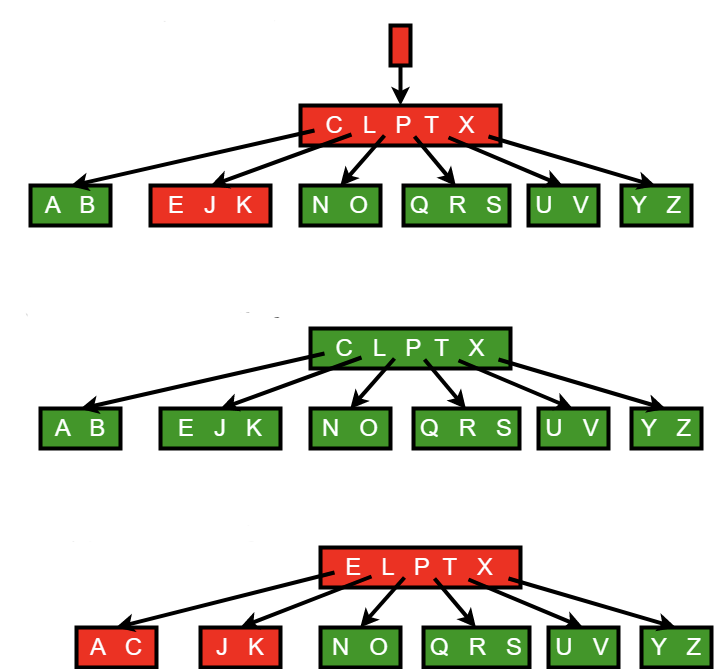
Eliminar M, caso 2a

1. Si Y tiene menos de t llaves, entonces, se verifica que que el otro hijo Z de K en el nodo X. Si Z tiene al menos t llaves, se deberá encontrar un sucesor K0 de K en la raíz del subárbol en Z. Se elimina K0, y se reemplaza k por K0 en X.
2. Si Y y Z tienen t-1 llaves, se deberá combinar K y toda Z en Y, de modo que X pierda K y el puntero a Z, por lo que Y tendrá 2t-1 llaves. Luego se libera Z y se borra K de Y.

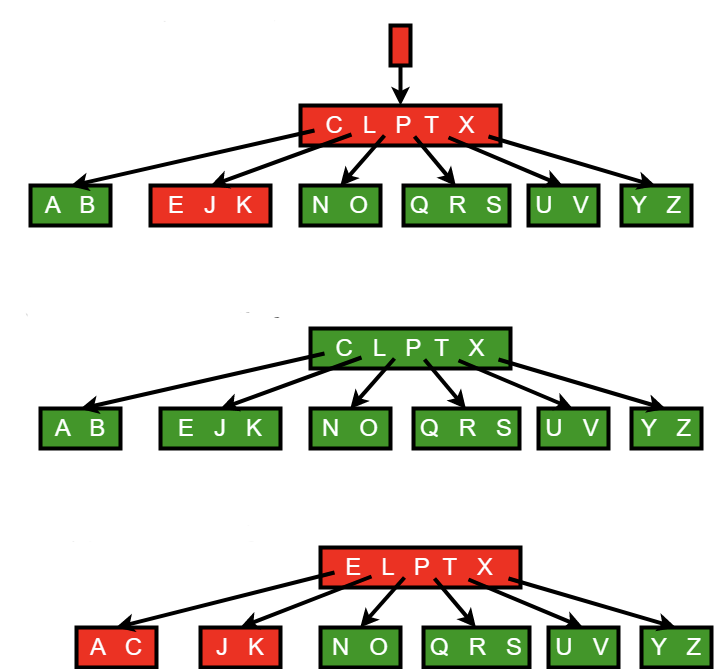


Eliminar G, caso 2c

1. Si la llave K no está presente en el nodo interno X, se determina la raíz XC(i) del subárbol apropiado que debe contener K. Si XC(i) tiene t-1 llaves, se llevan a cabo los pasos 3a o 3b según sea necesario para garantizar que se desciende a un nodo que al menos tenga t llaves. Luego termina recursando en el hijo apropiado de X.
2. Si XC(i) tiene solo t-1 llaves, pero tiene un hermano inmediato con al menos t llaves, dar a XC(i) una llave adicional moviendo una llave de X a XC(i), moviendo una llave de los hermanos inmediatos izquierdo o derecho de XC(i) a X y mover el puntero del hijo apropiado del hermano a XC(i).
3. Si XC(i) y los dos hermanos inmediatos del mismo tienen t-1 llaves, se fusionan XC(i) con un hermano, lo que implica mover una llave de X a el nuevo nodo fusionado, para convertirse en la lleve del medio de ese nodo.



Eliminar D, caso 3b

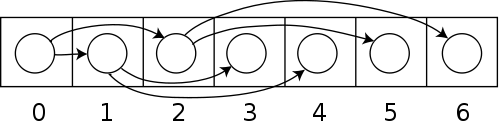


Eliminar B, caso 3a

# **Heap máximo**

Un árbol binario cumple la condición de heap máximo si todos los nodos padres de todos los subárboles tienen un valor mayor que los dos hijos, además es un árbol binario casi completo. Un árbol binario es completo cuando todos los niveles están llenos, a excepción del último que se llena de la izquierda hacia la derecha.

El heap se puede representar por vector colocando los elementos por niveles de izquierda a derecha.



Heap en vector

Siempre se puede calcular la posición de los hijos o del padre a partir de la posición de un nodo en el arreglo.

* El nodo raíz se almacena en la posición 0 del arreglo.
* Los hijos de un nodo almacenado en la posición K se almacenan en las posiciones 2K+1 y 2K+2 respectivamente.
* El padre de un nodo que está en la posición K está almacenado en la posición

[(K-1)/2], donde [X] es la parte entera de X.

## **Inserción**

(Matemática asociada O(log n))

Se inserta el nodo en la posición libre correspondiente, luego se compara con su padre. Si el hijo es menor que el padre, entonces el elemento es insertado correctamente, si ocurre lo contrario se deberá sustituir el hijo por el padre, al hacer la sustitución se deberá verificar recursivamente si su padre es mayor que él y hacer las sustituciones si es necesario.

## **Eliminación**

(Matemática asociada O(log n))

Al eliminar el elemento máximo de la raíz, se deberá subir el último elemento insertado y este quedará en None, después se tendrá que verificar si sus hijos son mayores que el, si es así, se sustituye el padre por el mayor de sus hijos, recursivamente hasta que se cumpla la propiedad de heap máximo.

## **Búsqueda**

(Matemática asociada O(n))

Si se quiere encontrar el valor máximo será la raíz, si quiere encontrar un valor X se tendrá que buscar por cada uno de los nodos correspondientes al árbol

# **Árbol digital**

Un árbol digital es un árbol binario en donde la posición de inserción de un elemento ya no depende de su valor, sino de su representación binaria. Los elementos en un árbol digital se almacenan sólo en sus hojas, pero no necesariamente todas las hojas contienen elementos (ver Figura 2).

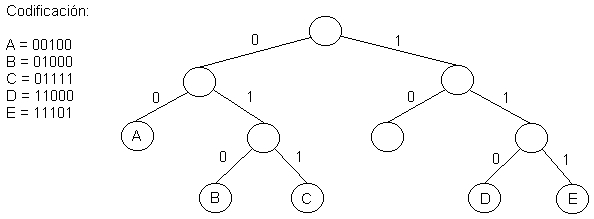


Figura 2

Los elementos de un conjunto se pueden representar como una secuencia de bits

X = b0b1b2....bk

## **Búsqueda**

(Matemática asociada O(M), M es el largo del bit)

Para buscar un elemento X en un árbol digital se procede de la siguiente manera:

* Se examinan los bits bi del elemento X, partiendo desde b0 en adelante.
* Si bi = 0 se avanza por la rama izquierda y se examina el siguiente bit (bi+1).
* Si bi = 1 se avanza por la rama derecha y se examina el siguiente bit (bi+1).
* El proceso termina cuando se llega a una hoja, único lugar posible en donde puede estar insertado X y se retorna.

## **Inserción**

(Matemática asociada O(M\*log n), M es el largo del bit)

Se quiere insertar X en el árbol digital. Se realiza una búsqueda infructuosa hasta llegar a una hoja. Si la hoja está vacía, se almacena X en dicha hoja. En caso contrario, se divide la hoja utilizando el siguiente bit del elemento y se repite el procedimiento, si es necesario, hasta que quede sólo un elemento por hoja.

## **Eliminación**

(Matemática asociada O(M\*log n), M es el largo del bit)

Se quiere eliminar el elemento X. Se elimina el elemento de la hoja, y por lo tanto ésta queda vacía. Si la hoja vacía es hermana de otra hoja no vacía, entonces ambas se fusionan y se repite el procedimiento mientras sea posible.

# **Tabla de rendimiento**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Búsqueda** | **Inserción** | **Eliminación** |
| **Lista enlazada simple** | Big O(n) | Big O(1) - Al principio  Big O(n) - Al final  Big O(n2) - En medio | Big O(1) - Al principio  Big O(n) - Al final  Big O(n2) - En medio |
| **AVL** | Big O(log n) | Big O(log n) | Big O(log n) |
| **Árboles B** | Big O(log n) | Big O(log n) | Big O(log n) |
| **Heap máximo** | Big O(n) | Big O(log n) | Big O(log n) |
| **Árboles digitales** | Big O(M), M es el largo de los bits | Big O(M\*log n) | Big O(M\*log n) |

# **Resultados**

Si se requiere una estructura de datos que almacene una cantidad pequeña de datos, se recomendaría usar lista simple enlazada, ya que su implementación es fácil, es más simple de entender y utilizar. Por otro lado, si se quiere almacenar grandes cantidades de datos, se recomienda usar AVL o Árbol B, ya que una vez implementados, la manipulación de estos es mucho más rápida y eficiente que la lista. Si los datos se repiten constantemente se recomienda usar Árboles digitales, ya que con esta estructura se ahorraría mucho espacio y su búsqueda sería rápidas. Si los datos se quieren clasificar según alguna prioridad se recomendaría usar Heap máximo, ya que siempre la prioridad más alta estaría en la raíz, por lo que sería muy rápido acceder a ella.