# Árboles 2-3.

Estructuras de Datos Avanzadas

Jesús Jauregui 180234.
Diego Pérez 181059
Rodrigo Gil 182473

En las ciencias de la computación, los árboles-2-3 son estructuras de datos de árbol que se encuentran comúnmente en las implementaciones de bases de datos y sistemas de archivos. Los árboles 2-3 mantienen los datos ordenados, y las inserciones y eliminaciones se realizan en tiempo logarítmico amortizado.

## Definición.

Los árboles 2-3 son un tipo de árbol balanceado por altura (*height balanced*). Se define como un árbol en donde todos los nodos no-terminales tienen 2 o 3 descendientes y todos los nodos hoja tienen la misma longitud (*path length*) o distancia desde la raíz.

Fueron introducidos con el objetivo de mejorar el tiempo de acceso en estructuras de datos manejados en memoria secundaria, donde el número de consultas a un registro influye de manera determinante en el tiempo de respuesta de la operación.

Según Donald Ervin Knuth, uno de los expertos más reconocidos en ciencias de la computación por su fructífera investigación dentro del análisis de algoritmos y compiladores, los árboles 2-3 son, en pocas palabras, un árbol B de orden 3. (Knuth, 2009)

## Propiedades.

Un árbol 2-3 permite que un nodo tenga dos o tres hijos. Esta característica le permite conservar el balanceo tras insertar o borrar elementos, por lo que el algoritmo de búsqueda es casi tan rápido como en un árbol de búsqueda de altura mínima. Por otro lado, es mucho más fácil de mantenerlo.

La estructura de árbol 2-3 exige que el crecimiento no se haga a nivel de las hojas (aunque la inserción sigue siendo en las hojas), sino a nivel de la raíz, ya que todas las hojas se deben mantener siempre en el mismo nivel. El proceso global de inserción comienza por localizar la hoja en la cual se debe agregar el elemento.

De forma recursiva se pueden definir como:

A es un árbol 2-3 de altura h si:

A es un árbol vacío (un árbol 2-3 de altura 0), o

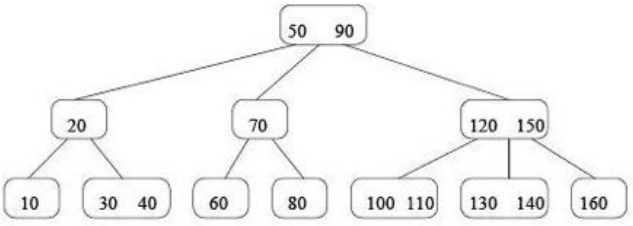
A es de la forma (r, I, D), donde r es un nodo e I y D son árboles 2-3 de altura $h - 1$, o

A es de la forma (r, I, C, D), donde r es un nodo e I, C y D son árboles 2-3 de altura h-1.

Para usar estos árboles de forma eficiente en las búsquedas, hay que introducir un orden entre los elementos por lo que un árbol A es un árbol 2-3 de búsqueda de altura h si:
Todos los elementos de I son menores que r y todos los elementos de D son mayores que r.

A es de la forma (r1, r2,I, C, D), donde r1 y r2, I, Ac y D son árboles 2-3 de búsqueda de altura h-1 y todos los elementos de I son menores que r1, todos los elementos de C son mayores que r1 y menores que r2 y todos los elementos de D son mayores que r2.

Esta definición implica que el número de hijos de un nodo es siempre uno más que el número de elementos que contiene ese nodo. En el caso de las hojas se permiten uno o dos elementos en el nodo. Desde ahora nos referiremos a los árboles 2-3 de búsqueda simplemente como árboles 2-3.




La especificación del tipo árbol 2-3 es muy similar a la de otros árboles con una diferencia que es la definición de tres operaciones generadoras en lugar de dos. ArbolVacìo es la operación que genera un árbol sin elementos, como en los otros tipos y hay una operación, *ConsArbol*, que genera un árbol 2-3 cuya raíz tiene un solo elemento y dos hijos y otra, *Cons3Arbol*, que genera un árbol 2-3 cuya raíz tiene dos elementos y tres hijos. Estas dos últimas operaciones son los generadores que se mantienen ocultos al usuario.

## Búsqueda.

El algoritmo de búsqueda es recursivo y es idéntico al de los árboles de búsqueda binaria, a excepción de que si el nodo tiene 2 valores, y el valor buscado se encuentra entre esos dos valores, buscaremos recursivamente en el *subárbol central* del *nodo X*

## Inserción.

A la hora de insertar un nuevo dato en un árbol 2-3 se hace de forma que se mantenga el equilibrio en el árbol. La capacidad de tener uno o dos elementos en cada nodo ayuda a conseguirlo. Aunque el árbol solo puede tener un nodo con 2 valores y 3 hijos o 1 valor y dos hijos, es muy común que la implementación del árbol se haga con la capacidad de hasta 3 valores y 4 hijos, los cuales son sólo momentáneos(como variables auxiliares) para facilitar los algoritmos de insercion y eliminacion.

## Pseudo código de inserción en un árbol 2-3.

*haremos uso de la función “split” , la cual es como sigue:

el llamado a *split* se hace cuando hacemos un *push* al nodo (se inserta un valor) y ahora el nodo tiene 3 valores, por lo cual necesitamos dividirlo.

Sea X el nodo donde ejecutamos *split*, tenemos los siguientes casos:

caso 1: X es un nodo hoja

pasa el valor de en medio al padre de este mediante un *push* y divide el nodo actual en dos nodos, los cuales serán hijos izquierdos o hijos derechos del padre, si el padre tiene 3 valores ahora, ejecuta *split* en este de manera recursiva

caso 2: X es un nodo interno con 4 hijos y 3 valores

subcaso a) X es la raíz:

crea un nuevo nodo que será la raíz con el valor de en medio, después, crea dos nodos que serán los hijos de la raíz, el hijo izquierdo tendrá como hijos a los dos hijos izquierdos del nodo original y tendrá como valor el valor izquierdo de este y similarmente, el hijo derecho que se creó con el valor máximo del nodo original será el padre de los hijos derecho del nodo original, formando así de un nodo con 3 valores y 4 hijos, un nodo (la raíz) con 2 hijos, los cuales tienen 2 hijos cada uno.

Si el árbol está vacío

subcaso b) X es un nodo interno

El algoritmo es igual que el anterior, pero en lugar de crear una nueva raíz, este valor se pasa al padre con un *push*, y los hijos nuevos que se forman, se le añaden al padre, ya sea como hijos izquierdo y central ,como central y derecho o como central izquierdo y central derecho, esto depende de la orientación de X (nótese que en el caso de que sean hijos central izquierdo y central derecho se haría una llamada recursiva de *split* al padre).

con el uso de *split* puede resumirse el código de inserción como el siguiente:

sea X la raíz y V el valor a insertar
si X es igual a Null/None:
    crea un nuevo nodo con el valor V y este es la raíz
de lo contrario:
    llama a X.*insertarT*

*InsertarT*:
si X es un nodo hoja:
    *push* V en X

de lo contrario
    si X tiene 3 hijos:
        llama recursivamente a *insertarT* con un hijo de X, ya sea el izquierdo , central o derecho (esto depende de la comparación de V con los dos valores de X, si V es menor que el valor minimo de X, sera el hijo izquierdo, de lo contrario si es menor que el valor máximo de X, mandalo con el hijo central , de lo contrario, se llamará recursivamente *insertarT* con el hijo derecho de X)

si X tiene 3 hijos:
    llama a *split* en X

## Eliminación.

El algoritmo para eliminación es comúnmente descrito como *lo contrario a la inserción*, y tiene cierto grado de verdad, sin embargo es igualmente considerado más complejo que el de inserción.

## Pseudocodigo de eliminación.

Llama a *find*
Sea X el nodo que contiene el valor a eliminar V
borra V
si X es un nodo hoja

si X tiene ahora 0 valores

llama a *merge* en X

de lo contrario

haz un *push* en el nodo X con el valor mínimo del subárbol derecho o central (dependiendo de si se borro el valor izquierdo o derecho de X)

llama recursivamente a la eliminación en ese valor (hacerlo con la raíz del subárbol de donde se sacó dicho valor, pues de lo contrario, debido a que el mismo valor puede ser encontrado en el árbol completo varias veces y puede generar errores).

## Pseudocodigo de merge.

Sea X el nodo en el que se llamó merge y P el padre de X

si X es la raíz(es decir P =None), el hijo de X es la nueva raíz(ya que merge sólo se llama cuando X tiene 0 valores, tiene igualmente un solo hijo)

si P tiene 3 hijos y el hermano inmediato de X tiene 2 valores

(ya que P tiene 3 hijos, P tiene dos valores) Roba un valor del padre, dependiendo de la orientación de X y del hijo que tiene 2 valores , puede ser el valor izquierdo o el valor derecho, después el padre roba un valor del hijo con dos valores, el hijo que donó un valor tiene que donar igualmente un hijo a X (por la definición del árbol, X tendrá un solo hijo y el hijo donador tendrá 3, para cumplir con la definición del árbol 2-3 los dos nodos tendrán 1 valor y dos hijos para cuando termine la operación de esta manera)

si P tiene 2 hijos y el otro hijo de P tiene 2 valores

se repite lo mismo que en el caso anterior

de lo contrario (el otro hijo de P tiene sólo un valor)

X se roba el único valor de P y el valor del otro hijo de P igualmente, X y su hermano se convierten en un solo nodo, por lo que P se convierte en un nodo con un solo hijo con dos valores (X) . Se deberá llamar recursivamente a *merge* en P (porque este tiene 0 valores)

si P tiene 3 hijos pero ninguno de ellos tiene dos valores:

P tiene 3 hijos y por lo tanto dos valores, X( que por definición tiene un hijo) se roba un valor del padre, el cual depende de la orientación de este.Posteriormente X se fusiona con su hermano más próximo (en el caso de ser X el hijo central puede fusionarse con cualquiera de los dos) al hacer esto X ahora tiene dos valores ( el que se robo del padre y el que era del hermano) y 3 hijos 1 suyo y otros 2 que eran del hermano de X.

## Motivación.

• Optimizar el tiempo de acceso en una estructura de datos manejadas en memoria secundaria en las cuales el número de consultas a un registro influye de manera determinante en el tiempo de respuesta de la operación.

• Acceso a la información en $O(\log_3 N)$

Baja complejidad en los algoritmos de actualización.

Son un tipo de árbol balanceado por altura.

• Todos los nodos pueden tener hasta 2 elementos.

Un nodo interno puede tener 2 ó 3 hijos, dependiendo de cuántos elementos posea el nodo:

Si hay 1 elemento en el nodo, debe tener 2 hijos.
Si hay 2 elementos en el nodo, debe tener 3 hijos.

## Análisis de complejidad.

La altura h de un árbol 2-3 en el peor caso es cuando es un árbol binario completo:
n=1+2+4+...+2(h-1) =(2h -1)/(2-1)=2h -1 nodos, es decir h <= log2 (n+1)

• En el mejor caso es un árbol ternario, entonces: n=1+3+9+...+3(h-1) =(3h -1)/(3-1)=(3h -1)/2nodos,
es decir h >= $\log_3(2n+1)$
Luego la altura está entre $\log_2(n)$ y $\log_3(n)$

La principal ventaja con los árboles 2-3 es que son de naturaleza equilibrada en comparación con los árboles de búsqueda binario cuya altura en el peor de los casos puede ser O (n). Debido a esto, en el peor de los casos, la complejidad temporal de operaciones como la búsqueda, inserción y eliminación es la altura de un árbol 2-3.

## Principal comparación con los árboles binarios de búsqueda.

El peor caso para búsqueda, inserción y eliminación en un árbol binario de búsqueda es de complejidad O(n) mientras que para un árbol 2-3 la complejidad en el peor de los casos es O(logN). Como consecuencia, la complejidad promedio para la búsqueda, inserción y eliminación de datos en un árbol 2-3 es O(logN).

Siendo N el número de datos almacenados en un árbol.

## Referencias.


2–3 tree. (2019, Septiembre 13). Recuperado de https://en.wikipedia.org/wiki/2–3_tree.

Agrawal, S., Kumar, A., & Cicekli, I. (2017). 2-3 and 2-3-4 Trees. Recuperado de http://www.cse.iitd.ac.in/~mausam/courses/col106/autumn2017/lectures/10-234trees.pdf.

Departamento de Lenguajes y Sistemas Informáticos. (2019, Mayo 21). Árboles 2-3. Recuperado de https://rua.ua.es/dspace/bitstream/10045/4411/21/ped-06_07-tema3_3.pdf.

Jain, R. (2019, Mayo 2). 2-3 Trees: (Search and Insert). Recuperado de https://www.geeksforgeeks.org/2-3-trees-search-and-insert/.

Solar, M., & Figueroa, L. (2010, Enero 3). Estructura de Datos. Recuperado de http://www.cime.cl/archivos/ILI134/1624_ED-Cap54A2-312010.pdf.

Universidad de Oviedo. (n.d.). ÁRBOLES DE ORDEN N Y GENERALES. Recuperado de http://www6.uniovi.es/usr/cesar/Uned/EDA/Apuntes/TAD_apUM_06.pdf.

University of Wisconsin-Madison. (n.d.). 2-3 Trees. Recuperado de http://pages.cs.wisc.edu/~vernon/cs367/notes/10.23TREE.html.

Árbol 2-3. (2019, Julio 14). Recuperado de https://es.wikipedia.org/wiki/Árbol_2-3.

Knuth, D. E. (2009). *Clasificación Y Búsqueda* (Vol. 3). Barcelona: Reverte. ISBN 13: 9788429126648


## Apéndice.

- Código del equipo (incompleto): https://pastebin.com/vBd7VJbZ
- Código funcional: http://code.activestate.com/recipes/577898-2-3-tree/
- Visualizador de árboles B: https://www.cs.usfca.edu/~galles/visualization/BTree.html