Árboles.

En un árbol vamos a encontrar eficiencia en tiempo para lograr las cosas, no tanto en espacio. Un árbol es un grafo, es decir:

$$G(V, E): \{V \text{ es un conjunto de vértices, } E \text{ es un conjunto de aristas}\}$$

$$G(V, E): \{\ \{v1, v2, \ldots\ , vn\}, \{(v1, v2), (v2, v3), \ldots\ , (vi, vj)\}\ \}$$

No precisamente los vértices tienen que ser esos, es un ejemplo para hacer "light" la definición formal.

Un árbol es un grafo **conexo** [si existe un camino entre cualquier pareja de vértices (un camino es una secuencia de aristas o un conjunto de vértices unidos por aristas)] y **acíclico** [que no exista un camino de un vértice para llegar a sí mismo y sin repetir aristas], con un vértice especial llamado **raíz**. Por la raíz podemos acceder a cualquier otro vértice en la estructura, gracias a la propiedad conexa; este nodo es como la cabeza de la estructura, por el cual podemos acceder a la misma.

A los nodos que están inmediatamente 'para abajo' en el camino les llamamos **hijos**, y a todos los que están 'debajo' de un nodo les llamamos descendientes; a los que están inmediatamente 'para arriba' en el camino les llamamos **predecesores**, y a los otros antecesores. Una **hoja** es un nodo que no tiene hijos, es el último nodo que va de la raíz hasta el término de un camino; a los nodos que están en el camino se les llama intermedios. También se puede decir que un árbol tiene varios sub-árboles dentro de él.

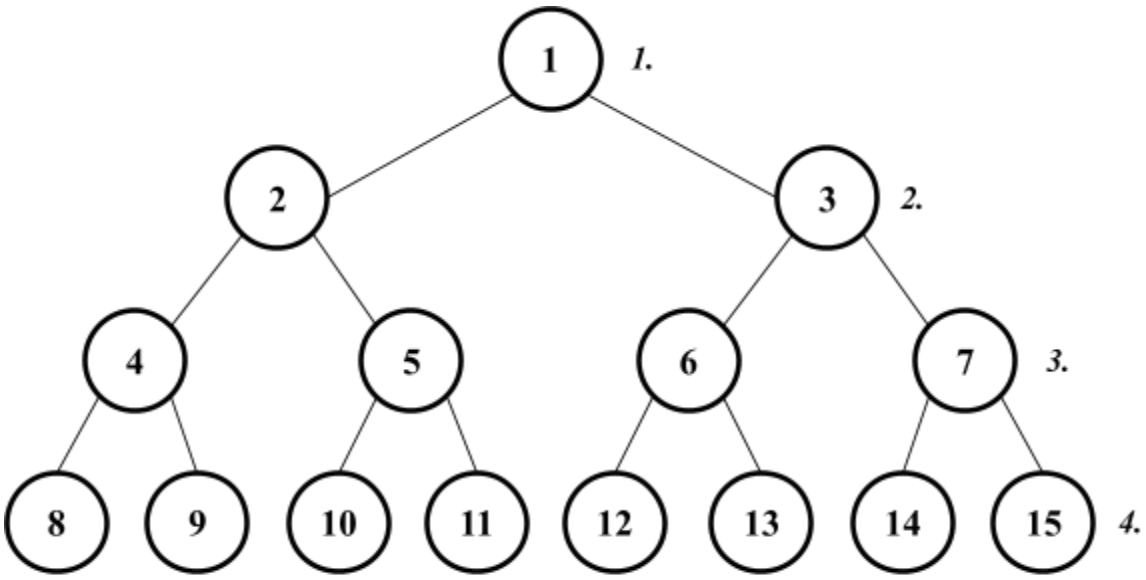
Una lista ligada especialmente es un árbol de orden 1. ¿Qué es un **orden 'n'**? El orden máximo de un árbol es el número de hijos máximos [n] que tiene un nodo (ya sea la raíz, o cualquier nodo de algún nivel), es decir, un nodo dentro de la estructura tiene *a lo más* 'n' nodos.

> Esto lo vamos a poder implementar en java por medio de nodos o por medio de arreglos. El problema de utilizar arreglos es que podemos llegar a desperdiciar demasiado espacio, pero vamos poco a poco.

Si tomamos en cuenta que un árbol que está **lleno** tiene 'n' niveles con 'm' elementos, además de que el **nivel 1** es la raíz, ¿cuántos niveles tiene un árbol binario que está lleno? La altura de un árbol que está lleno es de $log_2(m + 1)$. Ahora bien, ¿cuántos elementos tiene un árbol binario que está lleno?

La cantidad de elementos que tiene un árbol que está lleno es de $2^n - 1$.

Ejemplo de un árbol lleno y sus elementos:



Niveles: $\log_2(15+1) = 4$ **Elementos**: $2^4 - 1 = 15$

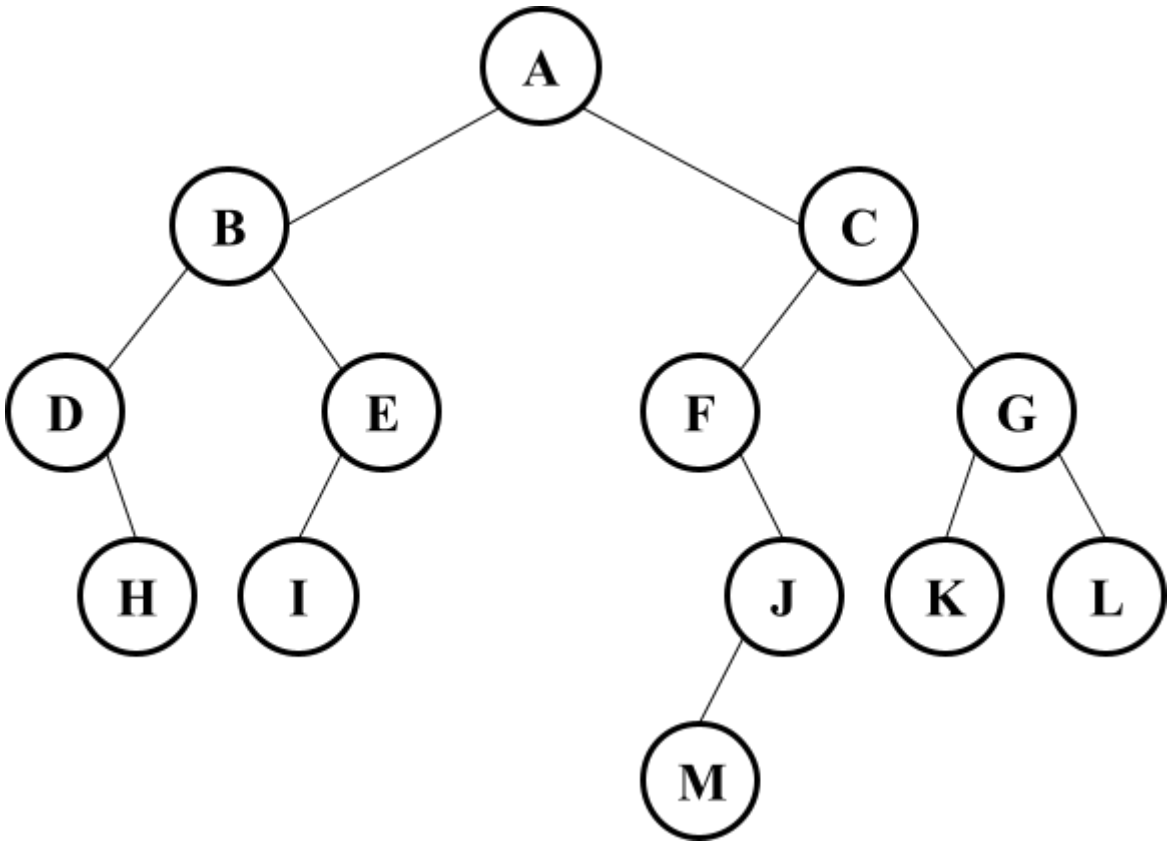
Conceptos rápidos previos.

Visitar un nodo: es procesar un nodo [imprimir el elemento del nodo, por ejemplo].

Recorrer un nodo: es simplemente pasar por un nodo.

Métodos importantes.

Árbol ejemplo.



Pre-order.

1. Visitamos el nodo en el que nos encontramos.
2. Recorremos a su hijo izquierdo.
3. Recorremos a su hijo derecho.

   Impresión ejemplo: A, B, D, H, E, I, C, F, J, M, G, K, L

In-order.

1. Recorremos a su hijo izquierdo.
2. Visitamos el nodo en el que nos encontramos.
3. Recorremos a su hijo derecho.

   Impresión ejemplo: D, H, B, I, E, A, F, M, J, C, K, G, L

Post-order.

1. Recorremos a su hijo izquierdo.
2. Recorremos a su hijo derecho.
3. Visitamos el nodo en el que nos encontramos.
   Impresión ejemplo: H, D, I, E, B, M, J, F, K, L, G, C, A

APUNTE NO. 7 Miércoles 17 de febrero

¿Podemos hacer estos métodos de manera **iterativa**?
Cuando ocupamos un método recursivo estamos aprovechando el Stack del sistema, es decir, la pila del sistema; ella es la que se acuerda de los elementos. Así que sí, podemos hacerlo de manera iterativa, pero tenemos que implementar una pila externa en el algoritmo.

APUNTE NO. 8 Viernes 19 de febrero

Árboles binarios de búsqueda.

Son árboles binarios que almacenan elementos comparables y que tienen la siguiente característica: para cada nodo C, los elementos almacenados en su subárbol izquierdo son menores o iguales que C, y los elementos almacenados en su subárbol derecho son mayores que C.

***Inserción** en árboles binarios de búsqueda*.
Debido a la característica misma de los árboles binarios de búsqueda, si queremos insertar un elemento S en el árbol, se debe de hacer lo siguiente: a cada paso del árbol [un nodo C] debemos comparar si S es menor o igual a lo almacenado en el nodo C para saber si el elemento S debe pertenecer al subárbol izquierdo del nodo o al derecho; si este nodo tiene hijo del lado en el cual debe ir elemento, realizamos la comparación una vez más y así sucesivamente. El elemento S se va a insertar cuando el hijo del nodo C del lado en el cual debe ir el elemento no exista [esto significa que se "terminó la rama" y da paso a la inserción de la hoja].

***Borrar** en árboles binarios de búsqueda*.

Borrar a un hijo significa que su padre ya no apunte a él, no que el elemento de ese hijo sea null. Además, hay casos para borrar en un árbol:

- Cuando el elemento **no existe**. En este caso no se hace nada porque no existe el elemento a borrar en el árbol.
- Cuando el elemento es una **hoja**. Lo que se hace es buscar al elemento y simplemente eliminar la referencia de su padre a él. Si es la raíz, solamente hay que determinar que la nueva raíz va a ser.
- Cuando el nodo tiene **un solo hijo**. En este caso primeramente se busca al nodo y luego al padre del nodo se le asigna el hijo del nodo. Hay que cuidar el caso cuando no tiene padre, pero ¿cuándo es posible eso? Cuando es la raíz. Si tenemos este caso especial, lo único que se debe de hacer es determinar que su hijo va a ser la nueva raíz.
- Cuando el nodo tiene **dos hijos**. Esto es un poco más complicado, pero de que se logra, se logra. Lo que se debe hacer es sustituir al nodo que se desea borrar por el sucesor in-order y eliminar el nodo en donde estaba el sucesor. Un recorrido in-orden nos va a dar de chico a grande, es decir, ordenado [claro, si el árbol ya está en orden]. El sucesor in-order de un elemento es el elemento más chico de su subárbol derecho. Para llegar al sucesor in-order, dado un nodo C, es irnos **uno** a la derecha y de ahí, **todos** a la izquierda.

  El sucesor in-order va a caer en uno de los casos anteriores: o tiene un hijo o no tiene ninguno. Si tiene un hijo, entonces forzosamente va a ser un hijo derecho; por lo tanto, el papá del sucesor in-order tiene que apuntar a ese hijo como su hijo izquierdo para no eliminar al subárbol completo.

  Hay un caso especial: cuando el sucesor in-order es el hijo derecho del nodo que se desea borrar. Lo único diferente aquí es que el papá del sucesor in-order tiene que apuntar a ese hijo como su hijo derecho para no eliminar al subárbol completo.

Por último, el método de borrado de un elemento en un árbol binario de búsqueda regresa un nodo que corresponde a los siguientes casos: si no existía el elemento regresa null, si el elemento borrado tenía un hijo o ningún hijo regresa a su padre, y si el elemento tenía dos hijos, regresa al padre del sucesor in-order.

Árbol AVL.

[Adelson-Velsky & Landis]

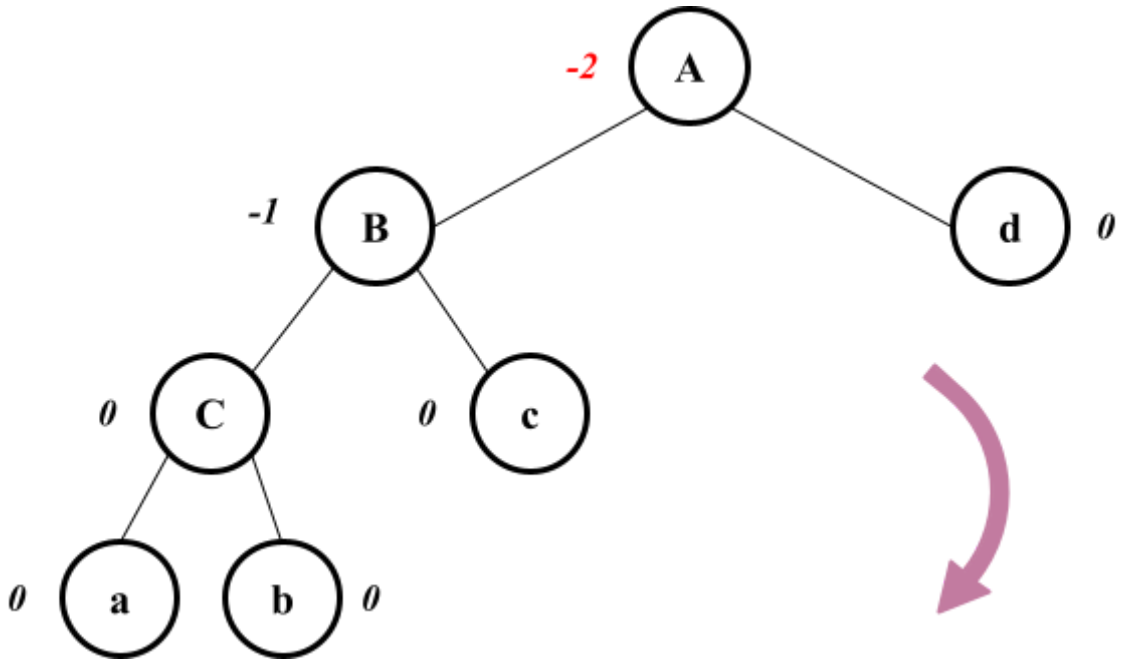
Queremos mantener la altura de nuestros árboles en log(n), pues en el peor de los casos en un árbol binario nos tardamos n. La idea es que para cada nodo más o menos queremos que tengan la misma cantidad de nodos en su lado izquierdo y su lado derecho. La idea en general es mantener un árbol "balanceado", y este es uno de los métodos. Un árbol solamente tiene sentido cuando está balanceado, porque en el peor de los casos es tan malo como una lista. La pregunta ahora es ¿qué es un árbol AVL?

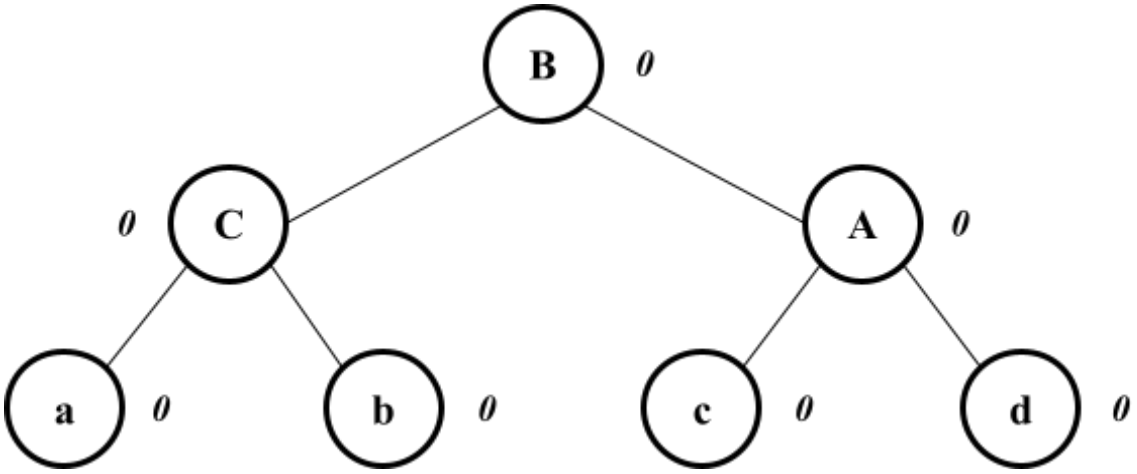
Son árboles binarios de búsqueda en los que para cada nodo, la diferencia absoluta en la altura entre su subárbol izquierdo y su subárbol derecho es igual a 1. La estrategia es utilizar operadores de rotación para mantener esta condición. Además, vamos a almacenar en cada nodo la diferencia de altura entre el subárbol derecho y el izquierdo. A esto le llamamos factor de equilibrio [este puede tomar valores {-1, 0, 1}, pues el signo nos dice de qué lado está más "pesado" el árbol].

Rotaciones.

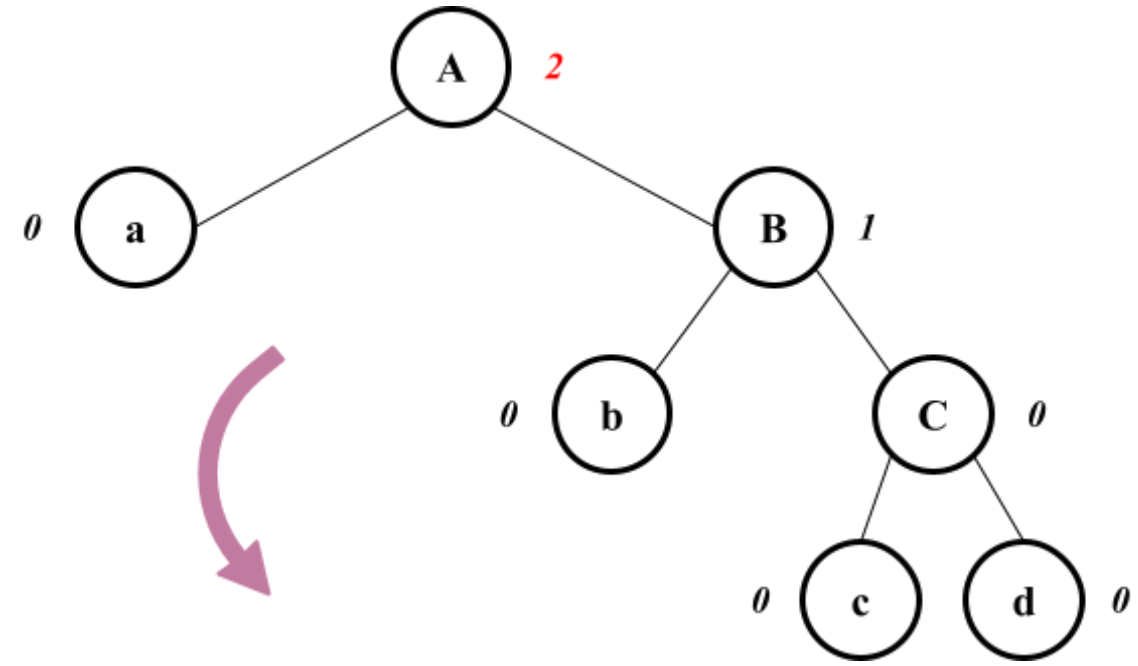
Son operaciones que invocamos cuando encontramos un problema. Para cada caso tenemos un factor de equilibrio de -2 para el nodo.

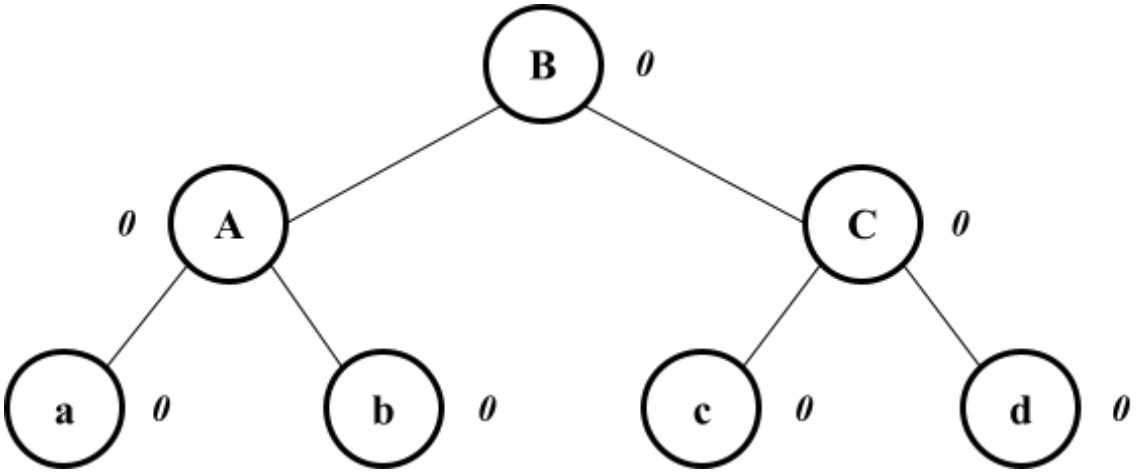
I. Izquierda-izquierda.



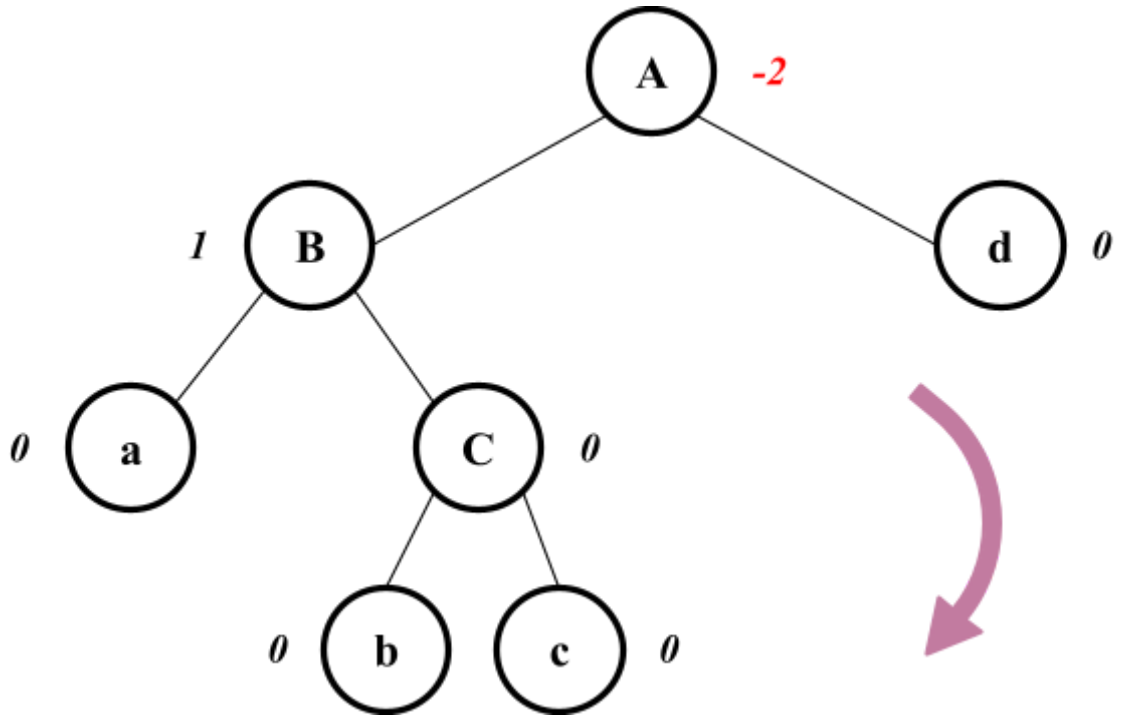


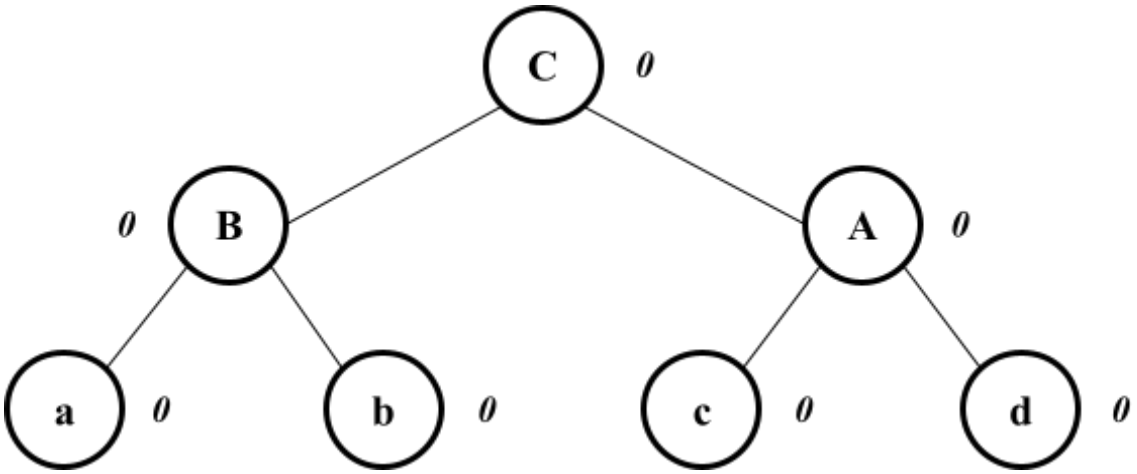
II. Derecha-derecha.



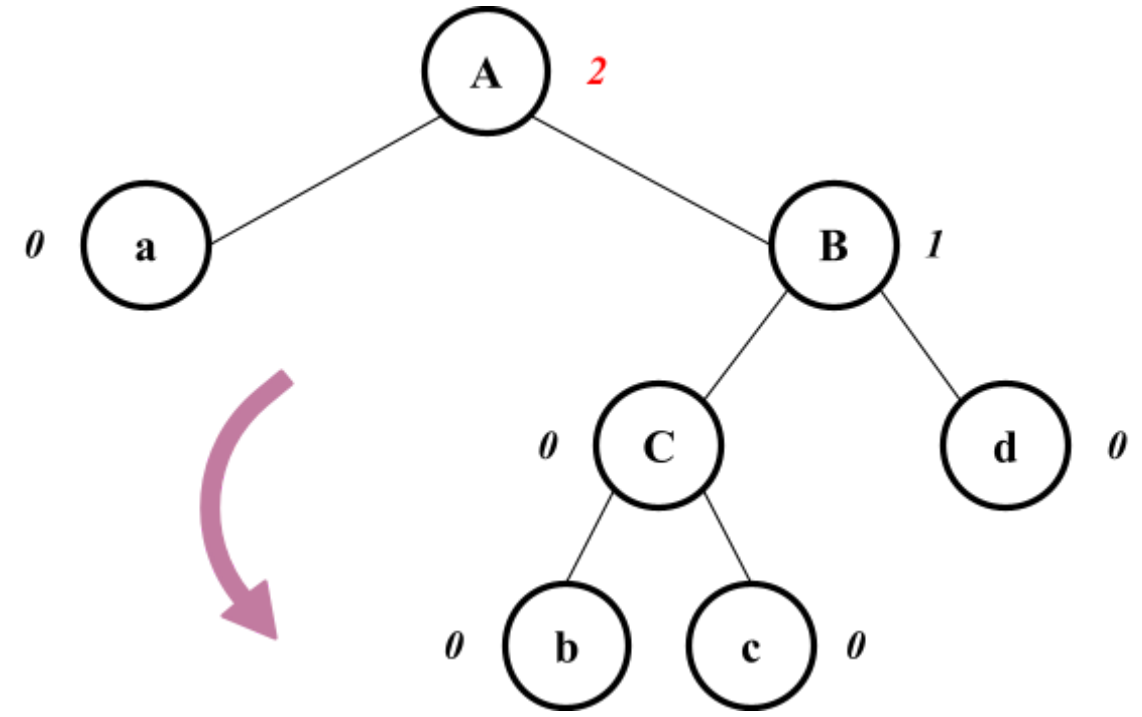


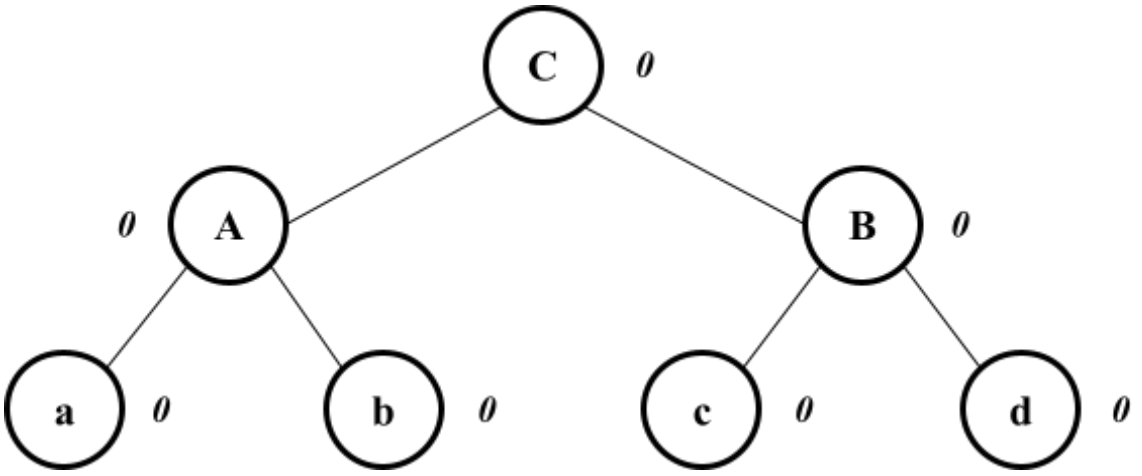
III. Izquierda-derecha.





IV. Derecha-izquierda.





APUNTE NO. 11 Lunes 1 de marzo

¿Cómo **insertar** en un árbol **AVL**?

La inserción del elemento en el árbol AVL se hace de la misma que la inserción en el árbol binario de búsqueda. Recordemos cómo se hace: por cada nodo se hacen comparaciones para saber si el elemento

a insertar es menor o es mayor que el elemento del nodo. Si es menor, nos movemos a su rama izquierda; si es mayor, nos movemos a su rama derecha. Este proceso es continuo hasta que nos encontramos con un nodo nulo, es decir, nos encontramos con un lugar vacío para lograr la inserción. Si esto es así, entonces ¿qué es lo diferente? La actualización de los factores de equilibrio.

Para actualizar los factores de equilibrio siempre se debe de tener en cuenta la "rama" por la cual nos estamos moviendo. Para hacer las actualizaciones de los factores de equilibrio se tiene que recorrer el árbol desde la posición del elemento insertado hacia arriba, es decir, en dirección hacia la raíz. En este recorrido ascendente es que actualizamos los contadores de los papás: si el nodo en el que nos encontramos [en la primera vuelta es el nodo que se insertó] es el hijo izquierdo de su papá, restamos una unidad al factor de equilibrio de este último; si es el hijo derecho, entonces sumamos una unidad al factor de equilibrio del padre. Nos vamos moviendo hacia arriba por medio de los padres. Al ir actualizando nos podremos encontrar con la problemática de que el factor de equilibrio resultante sea -2 o +2; si este caso se presenta, va a ser necesario hacer una rotación para balancear adecuadamente al subárbol. ¿Cuándo nos vamos a detener? Hay tres casos: primero, si al actualizar el factor de equilibrio este cambia de [-1 a 0] o de [1 a 0], es decir, cuando el factor de equilibrio se vuelve 0. Como segundo caso a parar es cuando se hace una rotación, pues recordemos cuál es la función de una rotación: mantener balanceado al árbol; por lo tanto, al hacer una rotación los factores de equilibrio se ajustan adecuadamente por su función de balanceo. Como último caso es que se termina la actualización cuando se llega a la raíz.

APUNTE NO. 12 Miércoles 3 de marzo

¿Cómo **borrar** en un árbol **AVL**?

El borrado de un elemento en un árbol AVL inicialmente se hace también de la misma manera que el borrado de un elemento en un árbol binario de búsqueda. Para recordar un poco, solo vamos a hacer mención de los cuatro casos posibles: no existe el elemento en el árbol, el elemento es una hoja, el elemento tiene un hijo o tiene dos hijos. Cada caso se resuelve de diferente manera, pero logra cumplir la función. El borrado del elemento como tal se hace por el llamado al método {borra} que pertenece al borrado de elemento en árbol binario de búsqueda [y que sí se puede hacer porque recordemos que un árbol AVL es un árbol binario de búsqueda], por lo que regresa null, al padre del elemento borrado o al padre del sucesor in-order. Lo diferente del borrado en esta estructura, así como en la inserción, es la actualización de los factores de equilibrio.

Al igual que la inserción de un elemento en un árbol AVL, en el borrado se tiene que tener siempre en cuenta la "rama" por la que nos estamos moviendo. Para hacer las actualizaciones del factor de equilibrio se empieza por la actualización del padre del elemento borrado hacia la raíz. Al principio se hace una actualización especial pues el método {borra} regresa al padre del elemento borrado, pero el recorrido se hace de manera normal después. Si el nodo en el que nos encontramos [en la primera vuelta es aquel que borramos] es el hijo izquierdo, sumamos una unidad al factor de equilibrio del papá; si es el hijo derecho, se resta una unidad al factor de equilibrio. El recorrido ascendente también debe de hacerse por medio de los padres. Si llega a haber alguna complicación [que sea -2 o +2] con el factor de equilibrio del padre, se hace el llamado al método de rotación para que efectúe la rotación adecuada y se mantenga el árbol equilibrado. Ahora bien, entonces ¿cuándo paramos? Al igual que en la inserción, existen tres casos: cuando se hace una rotación [porque su función es mantener balanceado al subárbol], cuando se llega a la raíz o cuando el factor de equilibrio pasa de [0 a -1] o de [0 a 1].

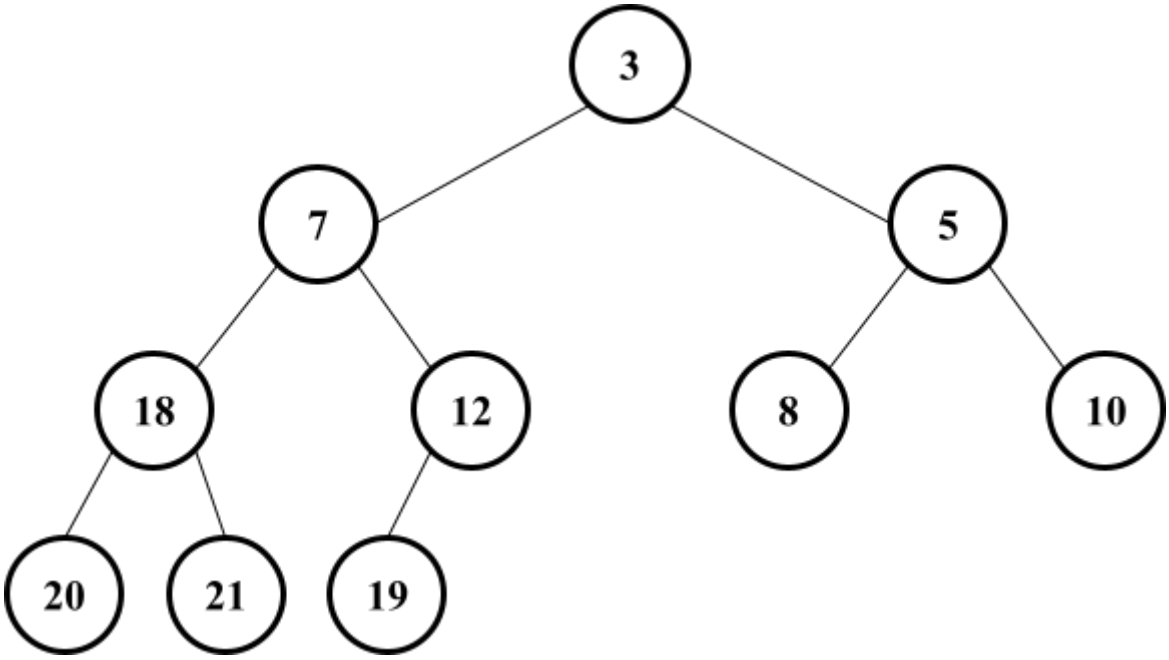
APUNTE NO. 14 Miércoles 17 de marzo

Heaps.

Los Heaps son árboles binarios, pero no son de búsqueda. ¿En qué radica su diferencia? En que son **árboles completos**, es decir, son árboles en los que todos sus niveles están llenos excepto posiblemente el último. Además, todas las hojas están recargadas hacia la izquierda y no hay nodos que tengan hijo derecho sin tener un hijo izquierdo. Por último, hay dos tipos de heaps: **min-heap** y **max-heap**.

Min-heap: para cada nodo del árbol, el elemento que está guardado en ese nodo es menor o igual al de todos sus descendientes. En otras palabras, para cada subárbol el elemento mínimo está en la raíz. Si nos fijamos bien, el árbol se va llenando por nivel de izquierda a derecha hasta que esté completamente lleno. Para n elementos, la estructura del heap siempre es la misma.

Ejemplo de **min-heap** de **10 elementos**:

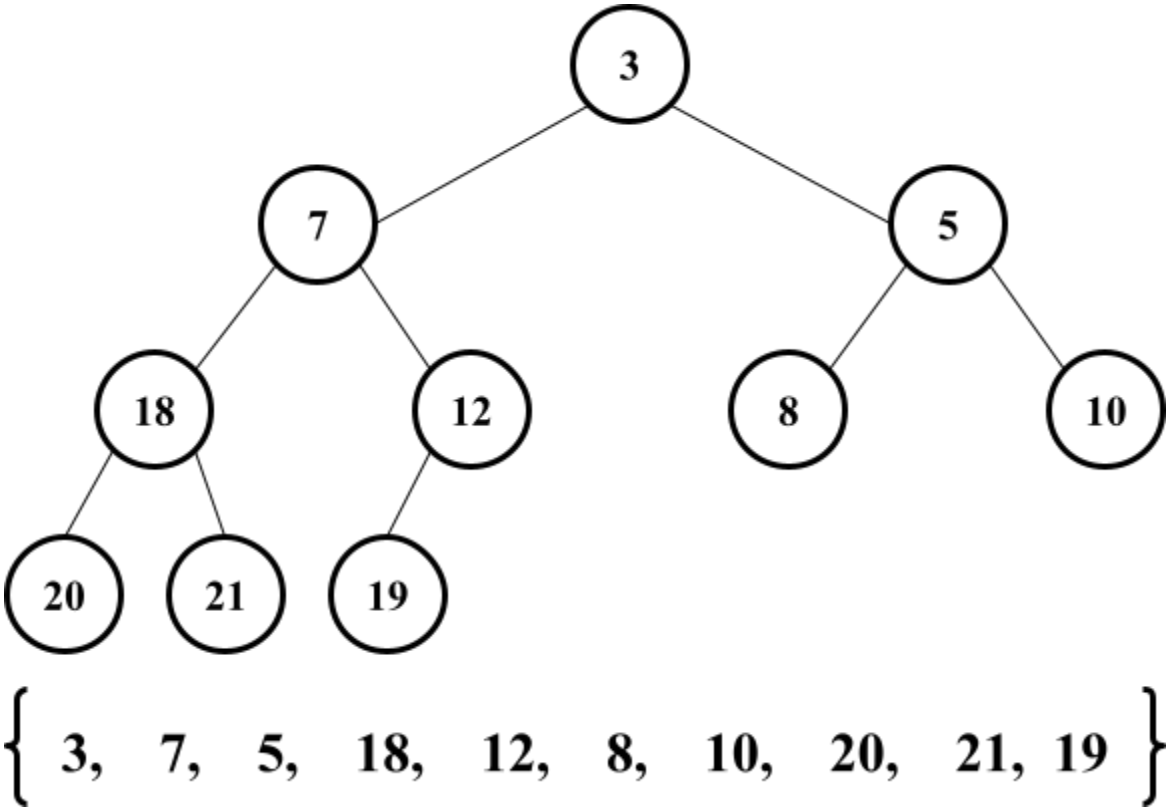


¿Cuáles son las ***operaciones***? Las operaciones de un heap son: insertar, buscar mínimo y borrar mínimo. En el caso del buscar mínimo es muy sencillo, pues siempre el elemento más chico está en la raíz; por lo tanto, el resultado es regresar la raíz y el algoritmo es de complejidad de ***O(1)***.
Ahora bien, ¿cuál es la **altura** de un heap? Sin detalles de implementación, se podría esperar que la altura sea de log(n). ¿Por qué? Porque se sabe que en la estructura todos los niveles están llenos excepto posiblemente por el último.

¿Cómo **implementar** los **heaps**? Si bien se puede implementar por medio de nodos, para fines de este curso se va a ver la implementación en un **arreglo**. Esto es para cambiar un poco el rumbo del asunto y para conocer otras formas de ver este tipo de estructuras. Recordemos rápidamente cuál era el problema de la implementación de un árbol en un arreglo: ya que no existen nodos, se tiene que determinar la forma, la lógica o la política en la que los elementos van a ir insertados en el arreglo para representar la raíz, el hijo izquierdo, el hijo derecho, etcétera. Si el árbol no está lleno, habrá mucho desperdicio de espacio en el arreglo en cualquier lógica que se haya determinado. La única manera de hacer el uso eficiente de todos los espacios es que **el árbol esté lleno**. Es por ello que podemos ocupar esta implementación en los heaps. Además, todos los heaps de n elementos se ven exactamente igual en

cuanto a la estructura o “acomodo visual” de los nodos, lo único que cambia es el contenido de cada nodo.

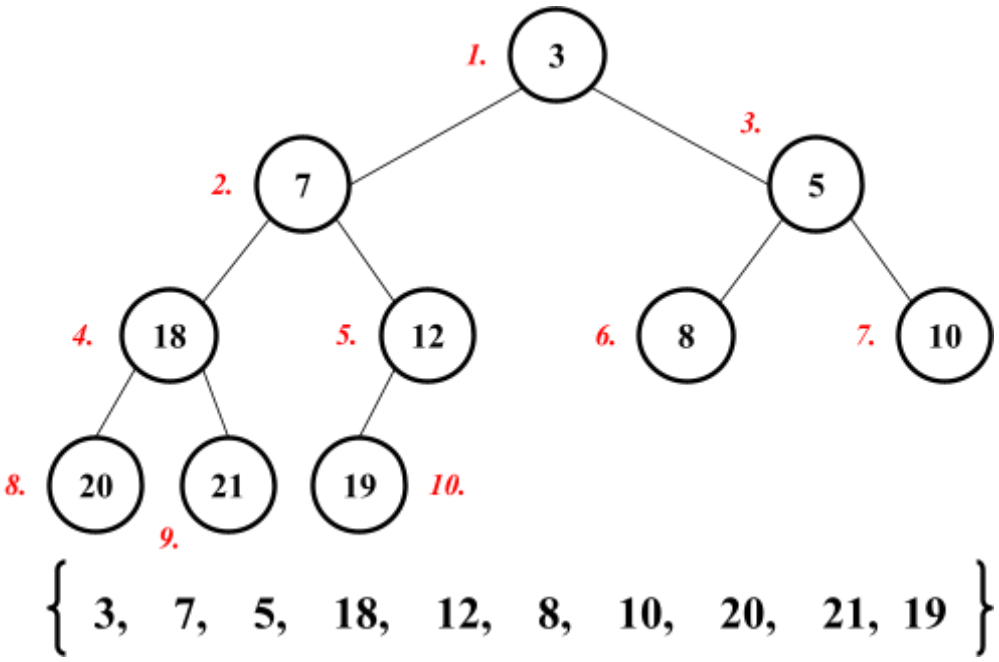
Ejemplo de **implementación** de **arreglos** para un heap:



Si nos fijamos bien, simplemente se está insertando en el arreglo por nivel de izquierda a derecha. Si tuviéramos que determinar por escrito cómo es la lógica de esta implementación en un arreglo, podríamos decir que es de la siguiente manera:

$$[\, r,\ i,\ d,\ ii,\ id,\ di,\ dd,\ iii,\ iid,\ idi,\ idd,\ dii,\ did,\ ddi,\ ddd\ ,\ \ldots]$$

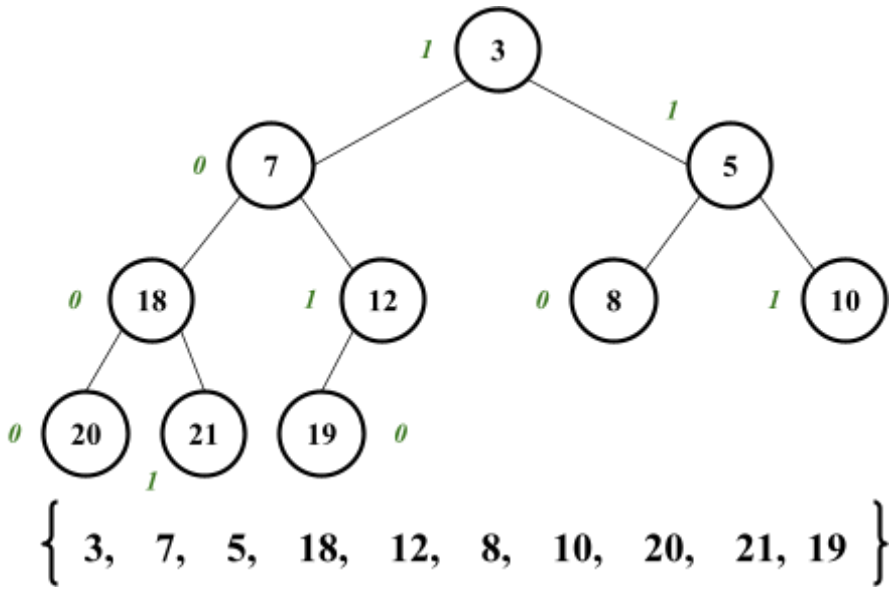
siendo (r) la representación de la raíz, (i) la representación del hijo izquierdo y (d) la representación de derecho. Esta es una propuesta para mantener una política de acomodo de los elementos del heap en un arreglo. Además, esta lógica propuesta nos brinda algunas ventajas. Veamos lo que sigue:



Esta es el orden de llenado del arreglo para un min-heap. Existe una pequeña fórmula para saber dónde encontrar a los hijos de un nodo en el arreglo y es relativamente sencilla y lógica, pero medio *tricky*. Sabemos que, por definición, los hijos de un nodo están en el siguiente nivel; también sabemos que para cada nivel, el número de elementos se duplica. Entonces la fórmula para encontrar a los hijos del nodo en un arreglo es multiplicando la posición en la que se encuentra por dos para el hijo izquierdo y multiplicar por dos más la suma de una unidad para el hijo derecho. En otras palabras tenemos que: **posHijoIzq = posActual * 2** , mientras que **posHijoDer = (posActual * 2) + 1 = posHijoIzq + 1**. Este es un patrón claro.

Pero aquí viene una parte confusa, pero bonita e interesante para checar los min-heaps. Dado que no tenemos apuntadores, ¿cómo podemos saber dónde están guardados los hijos?, ¿cómo navegamos el árbol? [pues estamos acostumbrados a navegar por las aristas o las "ramas" del árbol]. Permítase ponerle a las aristas izquierdas un **0** y a las derechas un **1**. ¿Por qué así? Porque a los programadores nos gusta pensar en binario. Además, veamos una tabla de las conversiones a binario:

Decimal	Binary
1	0
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010



Si nos damos cuenta, el recorrido se puede hacer por la "adición" del valor binario al que corresponde el nodo [determinado por el hecho de que sea hijo izquierdo o hijo derecho]. Sabemos que si quisiéramos el elemento 5 del heap en notación *hijo* sería: **{ raiz izq der }**. Por el valor binario antes mencionado, tendríamos entonces que: **{ 1 0 1 }**. Cool, ¿no? Entonces si guardamos n elementos, ¿cuántos bits se necesitan para guardar esos elementos en forma binaria? La respuesta es sencilla: **log(n)**. Porque por cada nivel existe una cantidad determinada de bits.

¿Cómo podemos guardar y manipular números binarios en la computadora?, ¿por qué se debe de poner **&&** como **and** en Java? Checar la clase del 17 de marzo para responder estas preguntas.

¡Bonus! Encriptación One-time pad y encriptación con llave pública y privada (números primos).

APUNTE NO. 15 *Lunes 24 de marzo*

¿Cómo **insertar** en un min-heap?

En principio, simplemente se inserta el elemento en la siguiente posición libre en el nivel correspondiente [esto es obligatorio por las propiedades de la estructura]; de esta manera se cumple que sea un árbol completo "recargado" a la izquierda. Hay que recordar la otra propiedad de un min-heap, a saber: que para cada nodo del árbol, el elemento que está guardado en ese nodo es menor o igual al de todos sus descendientes. Entonces, si el elemento insertado no cumple la condición de ser mayor que su padre, se hace un swap de los elementos. Si se realiza esta comparación, también se debe de hacer la

comparación del papá del elemento con el papá del papá, pues tampoco puede estar cumpliendo la condición y así sucesivamente. Por lo tanto, mientras que el elemento en el que nos encontramos sea más chico que su padre, se hace un swap de los mismos. ¿Cuándo terminamos? Cuando se cumple la propiedad o cuando se llega a la raíz. Este movimiento hacia los antecesores puede hacerse matemáticamente hablando [pos = pos/2] o por medio de bits [pos = pos << 1].

Insertar 1 elemento en el heap se tarda *log(n)*, e insertar n elementos en el heap se va a tardar *nlog(n)* porque es una suma *n* de *log(n)*. No tardamos *log(n)* en insertar un elemento porque en el peor de los casos lo tenemos que burbujear hasta la raíz y recordemos que este árbol va a estar completo en todos sus niveles exceptuando el último; por lo tanto, el árbol va a estar relativamente (o mayormente) balanceado.

APUNTE NO. 16 Lunes 22 de marzo

¿Cómo **borrar** en un min-heap?

Como primer paso tenemos algo extremadamente sencillo: como sabemos que el elemento a borrar es el mínimo del árbol y se cumple la propiedad de que para cada subárbol la raíz es el elemento más chico, entonces simplemente tenemos que eliminar (y regresar) al elemento que está en la primera posición del arreglo. ¿Qué vamos a poner en la raíz? Al elemento que está en la última posición de la estructura [del arreglo en este caso] y después decrementamos el contador para eliminar formalmente al elemento. Una vez que hicimos el primer *swap* (que corresponde a la raíz), se procede a la "heapificación". Esta última tiene que ver con la propiedad de que para cada nodo del árbol, el elemento que está guardado en ese nodo es menor o igual al de todos sus descendientes. Entonces ¿cómo se lleva a cabo? Empezamos por la raíz y nos movemos de una manera descendente. Primeramente se va a obtener la posición del elemento más chico entre los hijos de 'actual'; luego se va a comparar al elemento más chico entre los hijos con 'actual' para saber si se necesita hacer un swap necesario. Si el elemento más chico de los dos hijos es menor al elemento de 'actual', se hace el intercambio; de otro modo, no se tiene que hacer nada. Este proceso se sigue por el hijo que intercambiamos, de manera descendente. ¿Cuándo dejamos de bajar en la estructura? Cuando no es necesario hacer un swap o cuando ya llegamos al final. Solamente hay que tener cuidado sobre qué se debe de hacer cuando 'actual' solamente tiene un hijo [el izquierdo, por las propiedades de un heap].

Igual para borrar *n* datos de la estructura nos tardaremos *nlog(n)* por la misma lógica mencionada anteriormente.

Heap Sort.

Si insertamos *n* datos que se encuentran en un arreglo de entrada en un min-heap [nos tardamos *nlog(n)*], y luego borramos esos mismos *n* datos de la estructura [que también nos tardamos *nlog(n)*] y los colocamos en un arreglo de salida, ¿qué característica va a tener ese arreglo? Los datos en el arreglo de salida van a estar posicionados de **manera ordenada**. En otras palabras, lo que acabamos de deducir es que con un heap podemos insertar y borrar con un tiempo de *nlog(n)*, lo que metimos fueron datos desordenados y estos salieron ordenados. Es por ello que existe un algoritmo de ordenamiento llamado Heap Sort que se tarda *nlog(n)*.

APUNTE NO. 17 Lunes 5 de abril

Árboles B.

En los árboles se tiene el problema (o es un poco el problema) de que se desperdicie memoria al tener nodos que contengan solo a un elemento en su interior. Imaginemos que se tienen muchos datos de tal forma que no quepan todos los datos en la memoria principal [RAM], tendríamos que ocupar el disco secundario. Este segundo disco es el que conocemos como "Disco Duro". ¿Cuál es el problema con esto? Pensemos en un disco de vinilo: para acceder a un cacho de una canción para samplear, lo que se tendría que hacer primero es mover la aguja hasta la "circunferencia interior" donde se encuentre, y luego esperar a que de vuelta para encontrar ese cacho. Así es más o menos como funciona un disco duro. Ahora imaginemos que en cada "slot" (si es que podemos llamarle así a una locación de memoria en el disco) solo guarda un elemento en su interior. Para poder movernos en el árbol, la aguja tendría que primero encontrar la circunferencia inferior y luego esperar a que gire para encontrar cada uno de los elementos que están en el árbol. Como podemos ver, esto es completamente ineficiente y muy tardado. Además es un desperdicio porque no solo lee un elemento, sino que la aguja lee todo un bloque completo para acceder a ese elemento. Si lo pasamos a un ejemplo con la nube es lo mismo, porque se tendría que mandar un mensaje que pase por diferentes lugares [virtuales] para llegar a alguna sección de Silicon Valley [ejemplo] y que esta nos devuelva un mensaje diciendo que tenemos un elemento en ese nodo y se haga el recorrido de cada nodo en el árbol de la misma manera. Queremos minimizar esto, ¿es buena la motivación?

Lo que quisiéramos ahora es maximizar el número de cosas que están guardadas en un nodo para que en una sola lectura se pueda hacer una navegación más eficiente de un árbol y no tener que hacer 20 mil preguntas. Un ejemplo de estos árboles son los **árboles 2-3**: árboles de búsqueda que ya no son

binarios. El punto de todo esto es optimizar. Se puede decir que los árboles B son una generalización de los árboles de búsqueda para que no solo sean binarios. La idea de estos árboles es guardar muchos elementos en un solo nodo para que la lectura en cada acceso a la memoria secundaria sea mejor [menos desperdicio, más eficiente, más rápido, etcétera]; es decir, para que podamos leer más datos "de un jalón" en cada lectura de la memoria secundaria; en otras palabras, es aprovechar la maximización la cantidad de información útil que se está leyendo; si no ha quedado claro, se buscan optimizar los accesos a memoria secundaria. Porque otra vez: la memoria secundaria es mucho más lenta que la primaria [esta es mucho más cara]. La altura va a seguir siendo *log(n)*, pero la constante multiplicada por esa función va a ser mucho menor y la base del logaritmo va a ser diferente.

Árboles 2-3.

Tienen dos tipos de nodos:

1. **2-nodo**: almacenan un elemento C1. Los datos de su subárbol izquierdo son menores a C1 y los de su subárbol derecho son mayores a C1. Además, tienen cero o dos hijos forzosamente;
2. **3-nodo**: almacenan 2 elementos C1 y C2 con C1 < C2. Los elementos menores a C1 están en su subárbol izquierdo, los elementos que están entre C1 y C2 están en un subárbol medio y los elementos que son mayores a C2 están en su subárbol derecho. Además, tienen cero o tres hijos forzosamente.

- **Todas** las hojas están en el mismo nivel en el árbol.

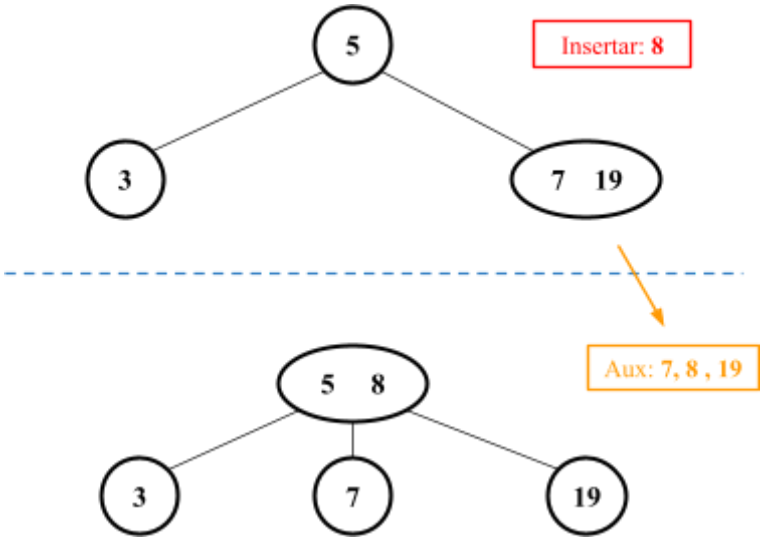
¿Cuáles son las **operaciones**? Las operaciones de un árbol 2-3 son insertar, borrar y buscar. Las operaciones son exactamente las mismas que las de un árbol binario de búsqueda.

Insertar en un árbol 2-3

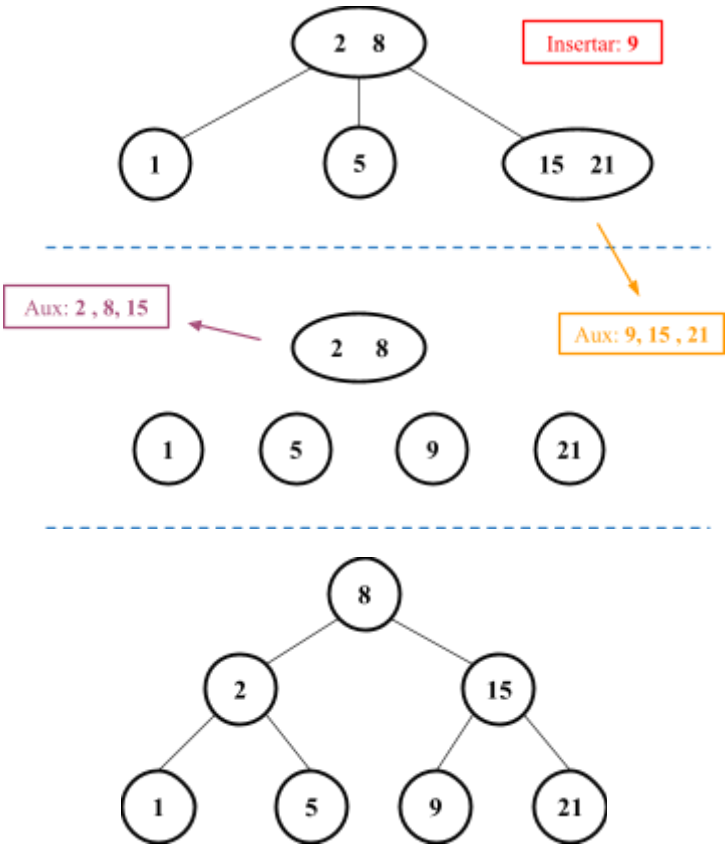
1. Se recorre el árbol hasta llegar a una hoja.
2. Si el nodo hoja es un 2-nodo, convertimos en 3-nodo e insertamos en ese nodo.
3. Si el nodo es un 3-nodo:
    a. Se toman el mínimo y el máximo entre los elementos C1, C2 del nodo hoja y el elemento a insertar y se convierten ambos elementos en 2-nodo.
    b. Se toma el elemento medio entre los 3 elementos y se inserta en el papá.
        I. Si el papá era un 2-nodo, al insertar el elemento medio, el nodo se convierte en un 3-nodo y el elemento mínimo se queda como hijo medio de este nodo. Ahora el nodo tiene 3 hijos.
        II. Si el papá era un 3-nodo, se sigue el mismo procedimiento que el punto número 3.

III. Si el nodo no tiene papá (es decir, si es la raíz), se crea una nueva raíz con el elemento medio.

Ejemplo de inserción en el caso **3.b.I**:



Ejemplo de inserción en el caso **3.b.II** y **3.b.III**:



Borrar en un árbol 2-3.

1. Se encuentra al elemento.
2. Si el elemento está en un nodo hoja, se debe de hacer lo siguiente:
    a. Si la hoja es un 3-nodo, se elimina al elemento y se convierte el nodo a un 2-nodo.
    b. Si la hoja es un 2-nodo, es balancea/reorganiza el árbol.
3. Si el elemento está en un nodo intermedio, se sustituye por el sucesor in-order y se toma en consideración el paso 2. ¿Por qué? Porque el sucesor in-order siempre va a estar en una hoja. ¿A qué se debe esto? Si el sucesor in-order tuviera hijo izquierdo, entonces ese no sería el sucesor in-order y como la condición que se debe cumplir es que tenga 0 o 2 nodos para los 2-nodo o 0 o 3 nodos para los 3-nodo, entonces no puede haber un nodo que tenga hijo derecho o medio sin tener hijo izquierdo.

Balanceo.

Lo que tenemos que tener en cuenta al momento de reorganizar el árbol es que el árbol resultante tiene que tener la estructura adecuada para que sea un Árbol 2-3. Si queremos recordarlo, regrese a la sección de la clase anterior. Los movimientos que se hacen pueden deducirse de manera intuitiva, lo único que hay que mantener son las propiedades del árbol.