Algoritmos y estructuras de datos

1. Introducción a la construcción de software
2. Algoritmos no recursivos
3. Algoritmos recursivos
4. Listas, pilas y colas
5. Ordenación
6. Tablas hash
7. **Árboles**
8. Grafos

www.u-tad.com

Grado en Ingeniería en Desarrollo de Contenidos Digitales

Curso 2015/2016

Prof. Dr. Carlos Grima Izquierdo
www.carlosgrima.com

v5

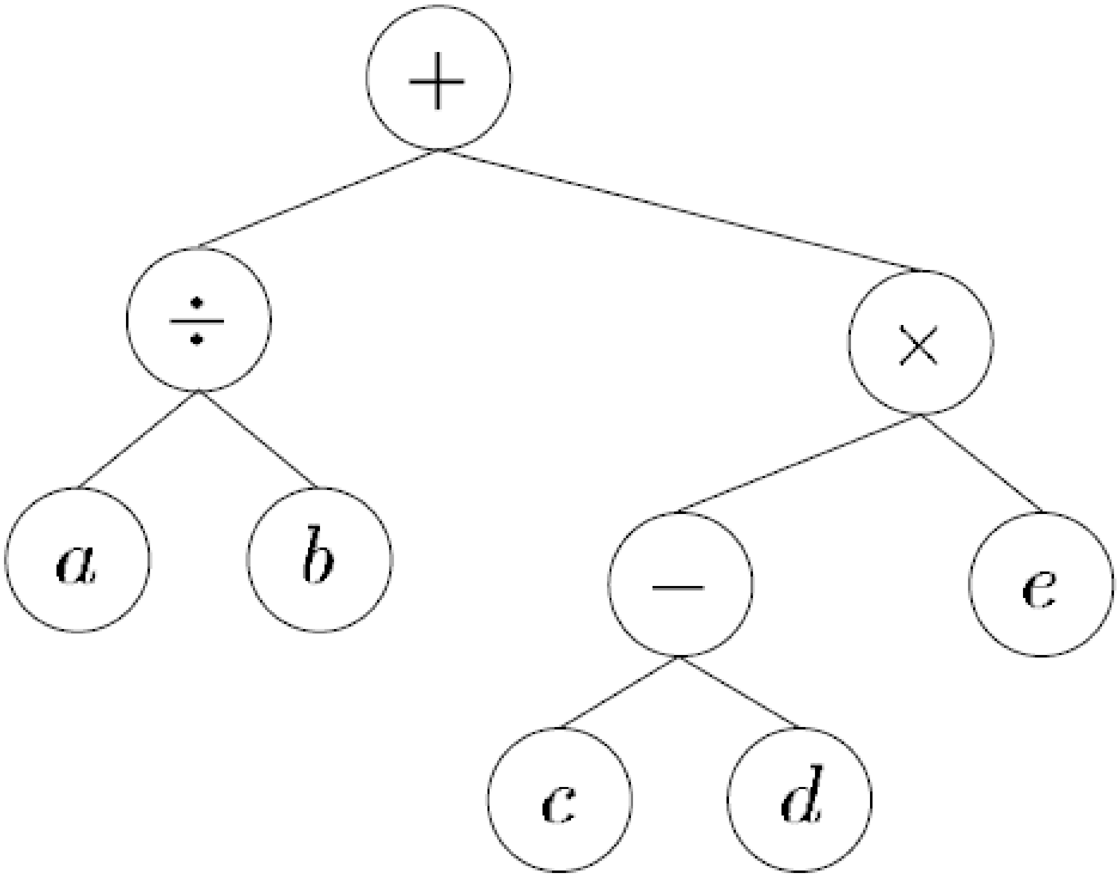
Árboles y sus definiciones I

- Un árbol es una colección de elementos llamados nodos, uno de los cuales se distingue como raíz, junto con una relación “de paternidad” que impone una estructura jerárquica sobre los nodos
  - Ej: un árbol genealógico

Árboles y sus definiciones II

- Definición recursiva de árbol
  - Ningún nodo es un árbol. Se llama árbol nulo (primer caso trivial)
  - Un solo nodo es un árbol. Ese nodo es la raíz del árbol (segundo caso trivial)
  - Si n es un nodo y A_1, A_2, … , A_k son árboles con raíces n_1, n_2, … , n_k respectivamente, se puede construir un nuevo árbol haciendo que n sea padre de n_1, n_2, … , n_k
    - n es la raíz de ese nuevo árbol, y A_1, A_2, … , A_k son subárboles de la raíz, y n_1, n_2, … , n_k son hijos directos de n

Árboles y sus definiciones III



Árboles y sus definiciones IV

- Ejemplos: árboles genealógicos, directorios, expresiones matemáticas, árboles sintácticos, capítulos y secciones de un libro
- Si $n_1, n_2, \ldots, n_k$ es una sucesión de nodos de un árbol tal que n_i es el padre de n_{i+1} para $1 \leq i < k$, entonces la secuencia se denomina **camino** del nodo n_1 al nodo n_k. La **longitud** de un camino es el número de nodos en el camino menos 1. Hay un camino de longitud 0 de cada nodo a si mismo.
- Si existe un camino de a a b, entonces a es **antecesor** de b, y b es un descendiente de a. Un antecesor (o **descendiente**) de un nodo que no sea el mismo se denomina **antecesor propio** (o **descendiente propio**).

Árboles y sus definiciones V

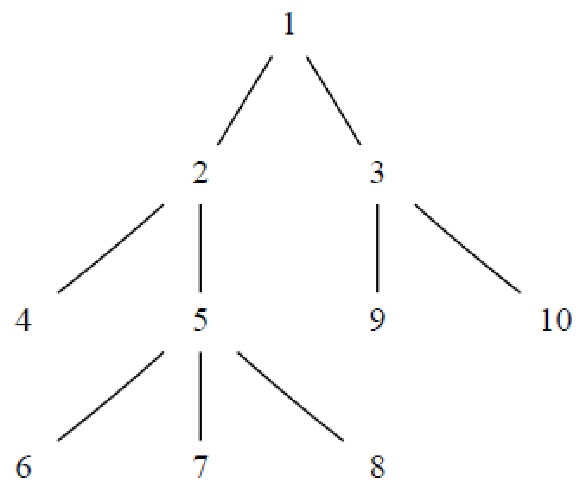
- Una **hoja** es un nodo sin descendientes propios. Un **subárbol** de un árbol es un nodo junto con todos sus descendientes.
- La **altura de un nodo** en un árbol es la longitud del camino más largo de ese nodo a una hoja. La **altura del árbol** es la altura de la raíz. La **profundidad** de un nodo es la longitud del camino único desde la raíz a ese nodo.
- A menudo se ordenan los hijos de izquierda a derecha. Si éste orden no importa hablaremos de un árbol **no ordenado**.
- Si a y b son hermanos y a está a la izquierda de b, diremos que todos los descendientes de a están a la izquierda de todos los descendientes de b. Dados dos nodos de un árbol, o uno es antecedente del otro, o uno está a la izquierda del otro.

Implementación de árboles I

- Existen dos implementaciones eficientes:
  - De hijo a padre
    - Una lista en la cual, por cada nodo, apuntamos a su padre
      - Recordemos que, en un árbol, cada nodo sólo tiene un padre (excepto la raíz, que no tiene ninguno y por lo tanto apuntaría a NULL)
  - De padre a hijos
    - Por cada nodo, guardamos todos sus hijos
  - También podemos tener una implementación mezcla de las dos:
    - Cada nodo apunta tanto a su padre como a sus hijos
    - Desperdiciamos algo más de memoria

Implementación de árboles II

- De hijo a padre, con un vector que empiece en la posición 1:
  - Cada nodo tiene a lo sumo un padre, por lo tanto podemos asociar a cada nodo esa información
  - Si numeramos los nodos desde 1 hasta MAX, podríamos utilizar un vector



Nodo	Padre
1	0
2	1
3	1
4	2
5	2
6	5
7	5
8	5
9	3
10	3

Implementación de árboles III

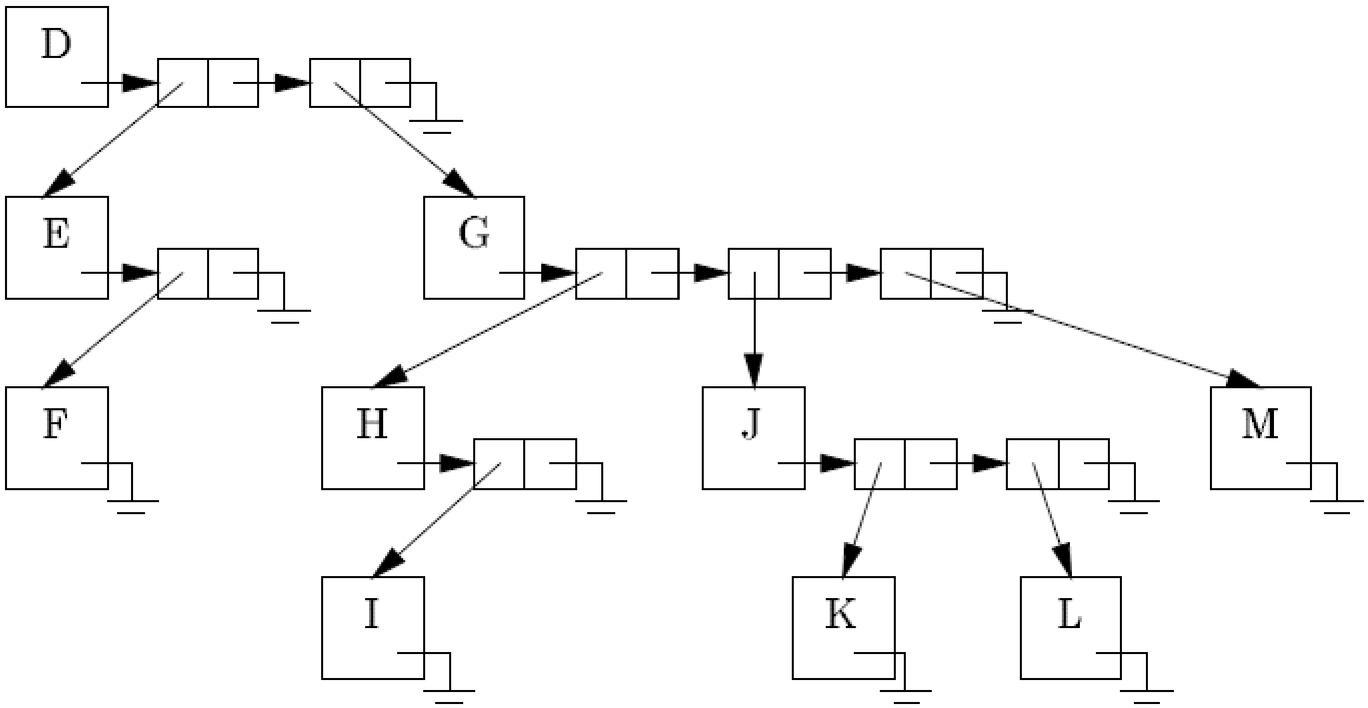
- Análisis temporal de la implementación de hijo a padre con un vector:
  - Obtener el padre de un nodo se puede realizar en O(1)
  - Sin embargo, averiguar todos los hijos de un nodo requiere recorrer todo el vector, por lo tanto es O(n), siendo n el número de nodos del árbol

Implementación de árboles IV

- Implementación de padre a hijos:
  - Por cada nodo tendremos una lista con punteros a todos sus hijos
  - O bien, por cada nodo, tendremos un puntero a su primer hijo y a su hermano derecho (**actividad 7.1**)

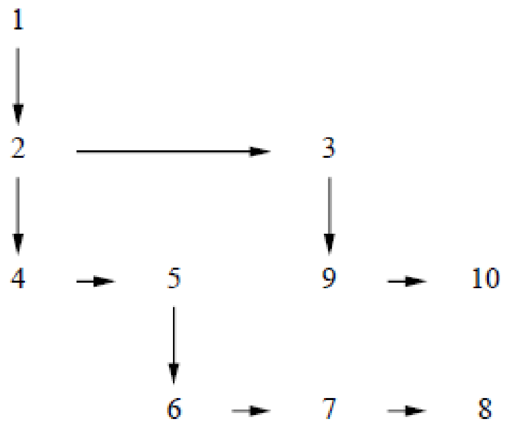
Implementación de árboles V

- Ejemplo: cada nodo tiene una lista enlazada con todos sus hijos. Es decir, cada elemento de la lista enlazada apunta a un nodo hijo



Implementación de árboles VI

- Ejemplo: cada nodo apunta a su primer hijo (si tiene) y a su hermano derecho (si tiene)



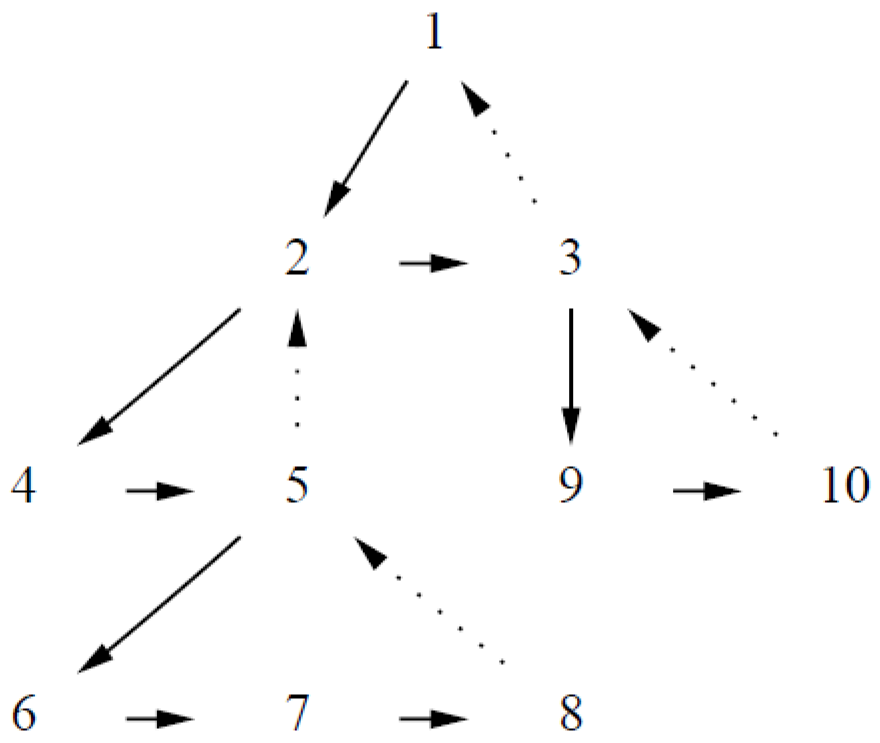
	HijoIzq	HermanoDer
1	2	0
2	4	3
3	9	0
4	0	5
5	6	0
6	0	7
7	0	8
8	0	0
9	0	10
10	0	0

Implementación de árboles VII

- Análisis de la implementación de padre a hijos:
  - Averiguar los hijos es eficiente: O(hijos)
  - Pero ahora es difícil encontrar el padre
    - Lo solucionamos añadiendo también, en cada nodo, un puntero al padre
    - Para no desperdiciar memoria, podríamos aprovechar el puntero al hermano derecho de los nodos que no tienen hermano derecho para apuntar a su padre

Implementación de árboles VIII

- Ejemplo de árbol en el cual el último hijo de un nodo apunta al padre:



Recorridos de árboles I

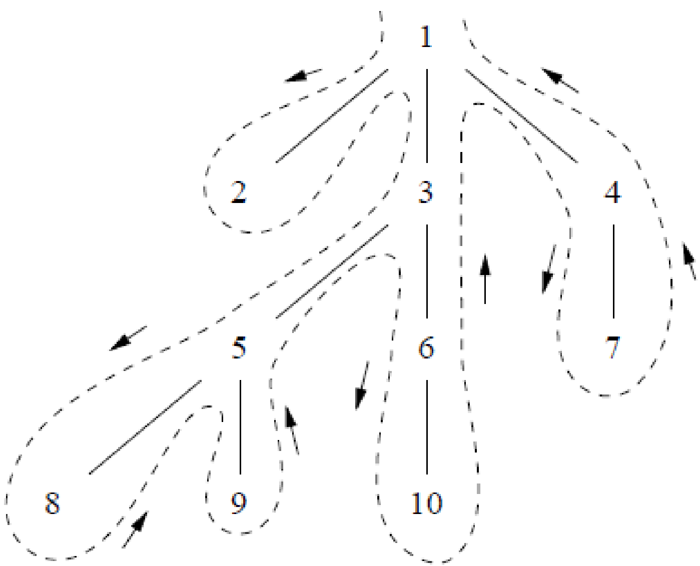
- Recorrer un árbol es transformarlo en una lista de nodos
  - Es decir, poner los nodos uno detrás de otro en un determinado orden
- Tipos de recorrido más habituales:
  - Preorden
  - Inorden
  - Postorden

Recorridos de árboles II

- Definición recursiva de los recorridos:
  - Para el árbol nulo, la lista vacía es el listado de los nodos para los 3 recorridos
  - Para el árbol con un solo nodo, la lista con ese nodo es el listado para los 3 recorridos
  - En otro caso, tenemos un árbol con raíz n y subárboles A_1, A_2, …, A_k
    - Preorden: ponemos n, ponemos A_1 en preorden, A_2 en preorden… y así sucesivamente hasta A_k en preorden
    - Postorden: ponemos A_1 en postorden, A_2 en postorden… y así sucesivamente hasta A_k en postorden. Finalmente ponemos n.
    - Inorden: ponemos A_1 en inorden, n, A_2 en inorden… y así sucesivamente hasta A_k en inorden

Recorridos de árboles III

- Otra forma de verlo (aunque mucho más difícil de programar) sería sacar los nodos en el orden en que pasamos por ellos al rodear el árbol:



- Pre: 1 2 3 5 8 9 6 10 4 7
- Post: 2 8 9 5 10 6 3 7 4 1
- In: 2 1 8 5 9 3 10 6 7 4
- Dada la línea discontinua, que se obtiene rodeando el árbol, en preorden se muestra un nodo la primera vez que se pasa por el, en postorden la última vez, y en inorden las hojas la primera vez y los nodos internos la segunda vez.

Definición de árbol binario

- Un árbol binario es:
  - Un árbol vacío
  - Un nodo (raíz) y dos árboles binarios disjuntos llamados subárbol izquierdo y subárbol derecho
- Dicho "informalmente": un árbol binario es un árbol en el cual cada nodo tiene 0, 1 ó 2 hijos

Tipos de árboles binarios I

- Árbol binario estricto: árbol binario (no vacío) en el que:
  - Los dos subárboles son vacíos, o
  - Los dos subárboles son binarios estrictos
- Dicho de otra forma:
  - Un árbol binario estricto es un árbol binario no vacío en el cual cada nodo tiene 0 ó 2 hijos (pero no 1)

Tipos de árboles binarios II

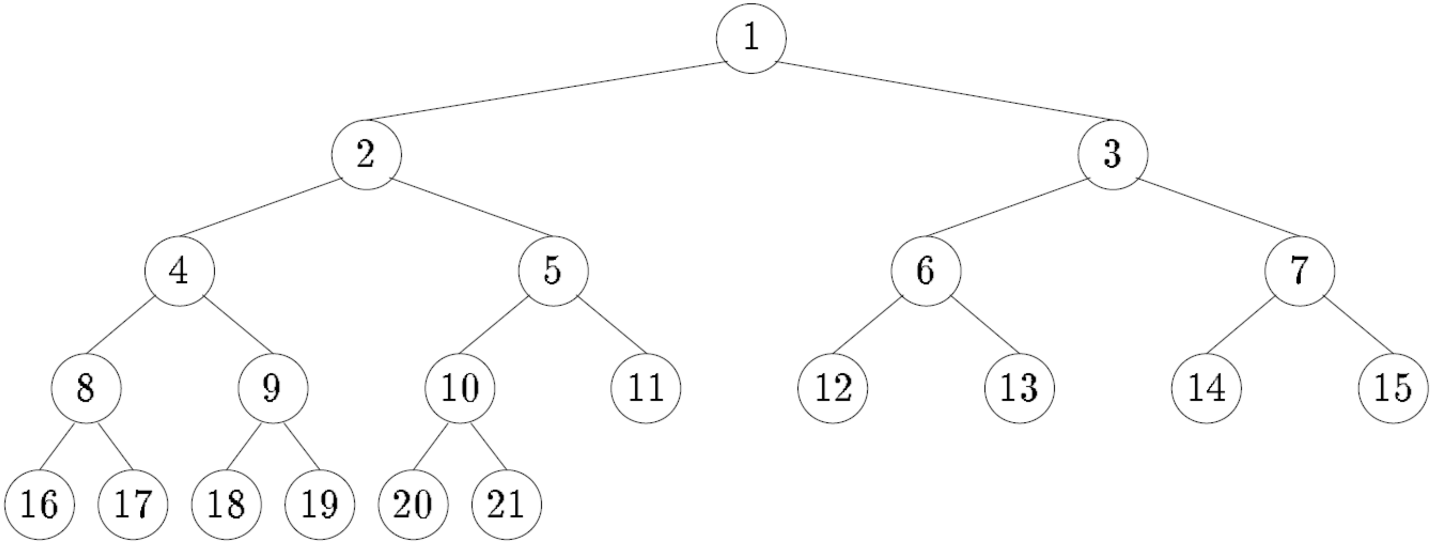
- Árbol binario perfectamente equilibrado: árbol binario en el cual:
  - Los subárboles izquierdo y derecho tienen la misma altura, y
  - Los dos son vacíos o los dos son perfectamente equilibrados
- Dicho de otra manera: un árbol binario perfectamente equilibrado es un árbol binario estricto y “perfecto”, con todas las ramas perfectamente simétricas

Tipos de árboles binarios III

- Árbol binario completo: un árbol binario de profundidad d en el cual:
  - Las hojas sólo están en el nivel d y en el nivel d-1, y
  - Para todo nodo n con un descendiente a la derecha en el nivel d,
    - Todos los descendientes a la izquierda de n que sean hojas también estarán en el nivel d
    - Todos los descendientes a la izquierda de n que no sean hojas tendrán dos hijos

Árboles binarios completos I

- Dicho de otra manera: un árbol binario completo es uno binario en el que vamos añadiendo nodos por niveles, de izquierda a derecha
- Ejemplo:

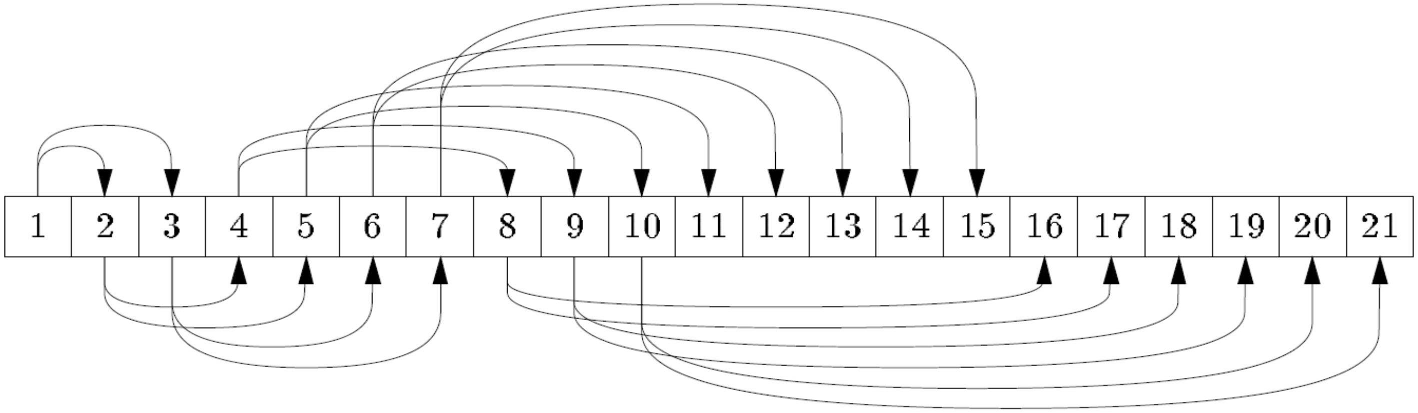


Árboles binarios completos II

- Un árbol binario completo es “casi” un árbol binario perfectamente equilibrado
  - De hecho, lo será cada vez que completemos un nivel más y aún no hayamos puesto la siguiente hoja del siguiente nivel
- Numeración típica de los nodos en un árbol binario completo:
  - Raíz es 1
  - Hijo izquierdo: doble del padre
  - Hijo derecho: doble del padre más 1

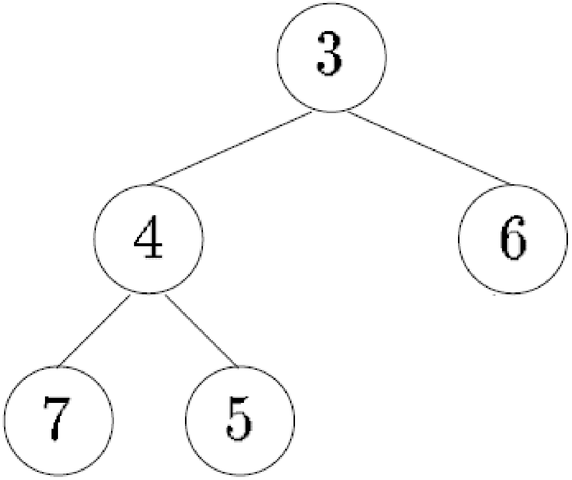
Árboles binarios completos III

- Implementación típica:
  - Con una lista contigua de nodos (vector)
  - Cada nodo está colocado en el lugar exacto del vector, según su numeración
    - Tener en cuenta que en C/C++ los vectores empiezan en 0, no en 1, por lo que habrá que restar uno a la posición en donde se guarda cada nodo
  - Añadir un nodo al final del árbol binario completo es simplemente insertarlo al final del vector



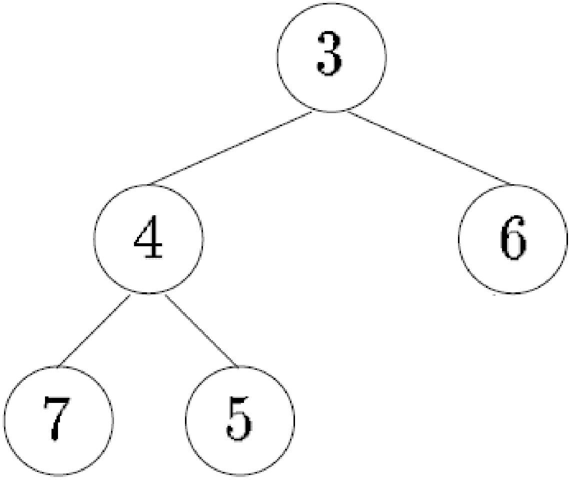
Montículos I

- Montículo (“heap”, en inglés): árbol binario completo que:
  - Está vacío (caso trivial) o,
  - Si tiene al menos un nodo:
    - Si la raíz tiene hijo izquierdo, la clave de la raíz es menor que la de su hijo izquierdo, y además el subárbol que empieza en dicho hijo es un montículo
    - Si la raíz tiene hijo derecho, la clave de la raíz es menor que la de su hijo derecho, y además el subárbol que empieza en dicho hijo es un montículo
- Informalmente: en un montículo la clave de cada nodo es menor que la de todos sus descendientes
- Tener cuidado con:
  - No confundir la clave de cada nodo con su numeración por niveles
  - Si admitimos duplicados, sustituir “menor” por “menor o igual” en la definición
  - Se podría definir también sustituyendo “menor” por “mayor” o por “mayor o igual” (dependerá de aplicaciones)



Montículos II

- Aplicaciones de los montículos:
  - Como cola de prioridad
    - Al hacer pop() de un montículo, sacamos siempre su raíz, que es el nodo con menor clave (con mayor prioridad)
  - Para implementar un nuevo algoritmo de ordenación muy eficiente llamado “HeapSort” (ordenación por montículo)
    - O(nlgn) en el peor caso, como Mergesort
      - En el caso promedio el QuickSort sigue siendo mejor

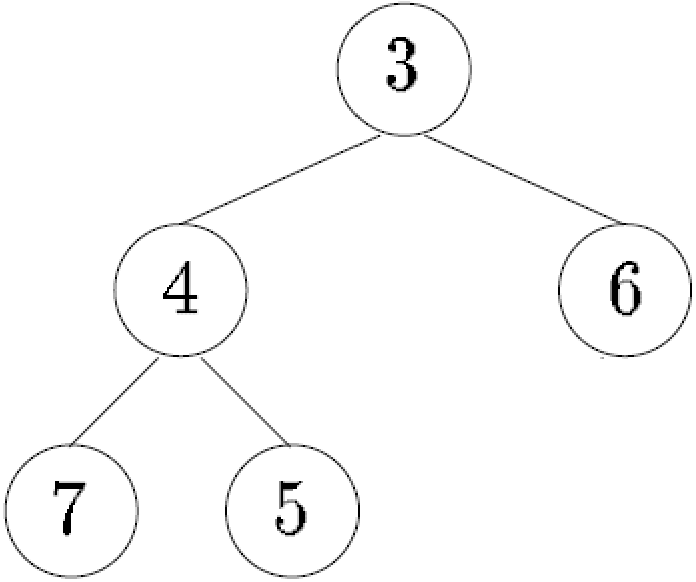


Montículos III

- Insertar en un montículo:
  - Se coloca el nuevo elemento en el último lugar y, mientras tenga padre y la clave del padre sea mayor (o menor, según cómo hayamos definido el montículo) que la de él, se intercambian ambos

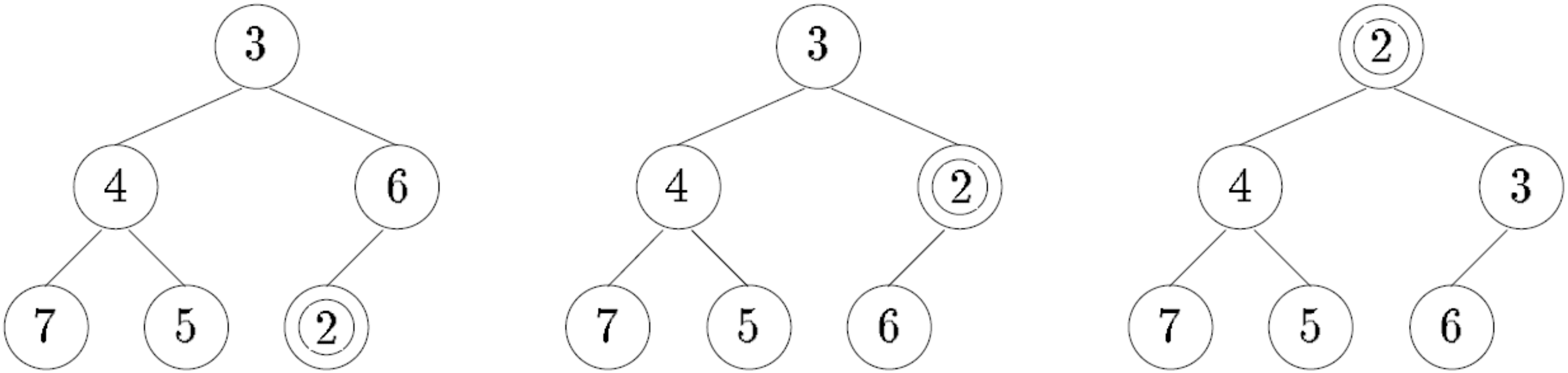
Montículos IV

- Ejemplo de insertar:
  - Quiero insertar el 2 en el siguiente montículo, en el cual cada nodo tiene una clave estrictamente menor que sus descendientes:
    - En el algoritmo descrito antes, recordemos que la clave de cada nodo tenía que ser mayor que la de sus descendientes



Montículos V

- Ejemplo de insertar (cont.):



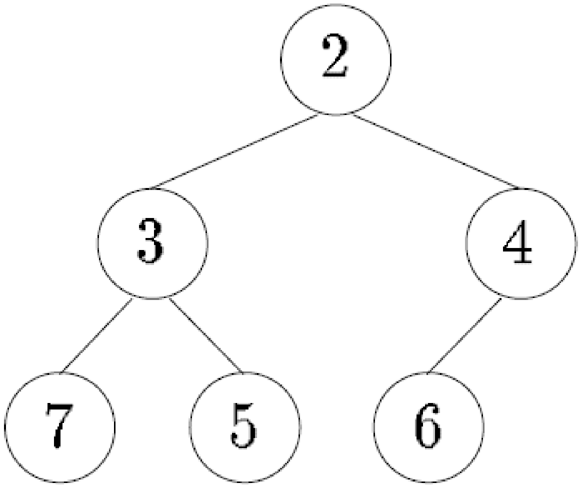
(a) (b) (c)

Montículos VI

- Análisis temporal de insertar:
  - El peor caso es cuando la hoja insertada tiene que ascender hasta la raíz
  - ¿Por cuántos niveles asciende?
    - Un árbol binario perfectamente equilibrado tiene $\log_2 n$ niveles, siendo n el número de nodos
    - Un árbol binario completo es "casi" un árbol binario perfectamente equilibrado, excepto en unas pocas hojas que le faltan para completar el nivel
      - Esas pocas hojas se desprecian en el infinito (no lo demostramos)
      - En cualquier caso, la hoja que insertamos siempre está en el nivel más inferior
    - Por lo tanto, la hoja insertada asciende del orden de $\log_2 n$ niveles en el peor caso
  - Por lo tanto, insertar es O(lgn)

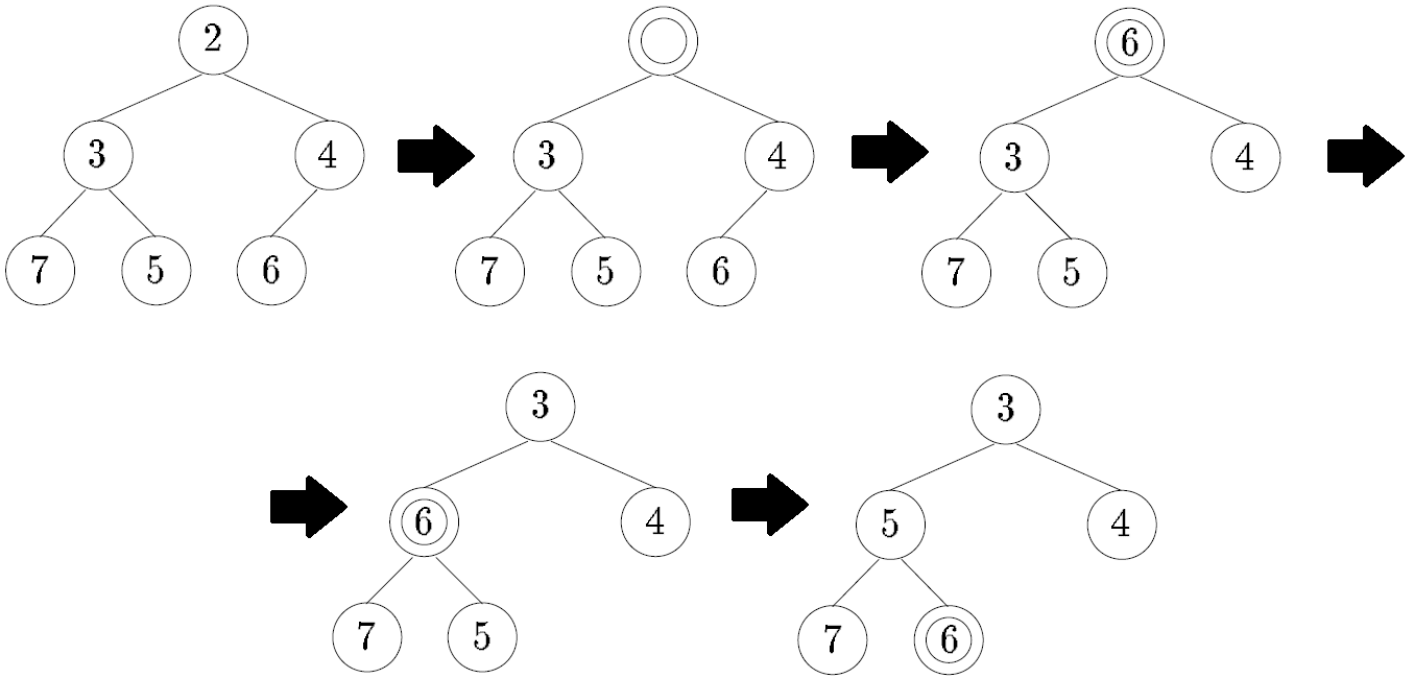
Montículos VII

- Eliminar
  - Eliminamos siempre la raíz
  - Ponemos el último nodo en el lugar de la raíz que teníamos antes
  - Ahora todo el árbol es un montículo, con la posible excepción de la raíz
  - Por lo tanto hay que "reestructurar" el montículo: ir bajando la raíz hasta que quede en el lugar correcto y el árbol entero vuelva a ser un montículo
    - Cada vez que la raíz no está en su sitio, la intercambiamos con su hijo de menor clave
    - Si el montículo estuviera definido al revés (cada nodo es mayor que sus descendientes), entonces el intercambio sería con su hijo de mayor clave
  - Ej: vamos a eliminar la raíz (el 2) del siguiente montículo



Montículos VIII

- Ejemplo de eliminar:



Montículos IX

- Análisis temporal de eliminar
  - Sacar la raíz y poner en su lugar el último elemento es O(1), porque estamos trabajando sobre una lista contigua
  - Reestructurar es bajar la raíz un cierto número de niveles
    - En el peor caso tendrá que bajar hasta el nivel más inferior, por lo tanto será O(lgn)
  - Por lo tanto eliminar es O(lgn)

HeapSort I

- El algoritmo “HeapSort” permite ordenar un vector, viéndolo como un montículo
- Conseguiremos una complejidad temporal de O(nlgn)
- Tendremos dos versiones del algoritmo, según la complejidad espacial:
  - Gastando memoria adicional O(n). Algoritmo sencillo de implementar.
  - Sin gastar memoria adicional. Algoritmo más complicado (no lo vemos).

HeapSort II

- Pasos del algoritmo que gasta memoria adicional (actividad 7.2):
  - Partimos de un vector desordenado, y lo queremos ordenar de menor a mayor
  - Creamos un montículo temporal vacío, en el cual cada elemento sea menor que todos sus descendientes
  - Tomamos cada elemento del vector y lo vamos insertando en el montículo
  - Una vez que el montículo tenga todos los elementos, vamos obteniendo elemento a elemento del montículo, y lo vamos poniendo en sucesivas posiciones del vector de origen, de la primera posición a la última
  - Nos queda el vector de origen, ordenado de menor a mayor. Podemos ahora destruir el montículo temporal.
    - Como vemos, necesitamos un montículo temporal de n elementos durante la ejecución del algoritmo, por lo tanto la complejidad espacial es O(n)

HeapSort III

- Aproximación sencilla al análisis temporal de HeapSort en el peor caso:
  - La fase de construcción del Montículo (mediante inserciones sucesivas) es O(nlgn), porque insertamos n veces y cada inserción es O(lgn)
  - El proceso de extraer repetidamente los elementos es también O(nlgn), ya que obtenemos n elementos, y cada obtención es O(lgn)
  - Por lo tanto el algoritmo es O(nlgn) en el peor caso
  - En promedio, se comporta peor que el QuickSort

HeapSort IV

- Realmente el análisis temporal de HeapSort es más complicado (no lo vemos), ya que:
  - Las primeras veces que insertamos en el montículo hay muy pocos elementos en él aún, por lo tanto esas primeras veces la inserción es más rápida que O(lgn)
  - Igualmente, las últimas veces que extraemos los elementos del montículo ya quedan muy pocos elementos en él, por lo tanto esas últimas veces la inserción es más rápida que O(lgn)
  - Habría que calcular, con sumatorios y series, la media en cada uno de los dos casos. No obstante, el resultado final del algoritmo sigue siendo O(nlgn)

Definición de ABB I

- Recordemos que la operación más importante de una lista (o base de datos) es la búsqueda
  - Por lo tanto es muy importante optimizar todo lo que podamos el tiempo de las búsquedas
- Hasta ahora, las mejores búsquedas son:
  - Si tenemos una tabla hash, O(1)
    - Problema: a costa de desperdiciar mucha memoria para que la tabla esté poco cargada
    - Problema: como la tabla hash se implementa con una lista contigua, necesitamos encontrar un gran bloque de memoria contigua
  - Si tenemos una lista contigua, O(lgn) con la búsqueda binaria
    - Problema: la lista tiene que estar ordenada
    - Problema: las listas contiguas requieren grandes bloques de memoria contigua
    - Problema: las eliminaciones/inserciones son costosas, sobre todo al principio de la lista
  - Si tenemos una lista enlazada, O(n) con la búsqueda secuencial

Definición de ABB II

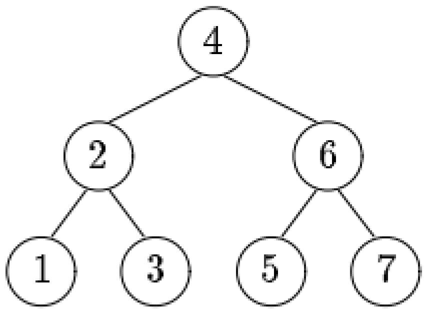
- ¿Podríamos conseguir búsquedas más eficientes en las listas enlazadas?
  - Si al menos pudiéramos conseguir búsquedas en un tiempo O(lgn), igualaríamos el rendimiento de las listas contiguas, pero sin tener los problemas de una lista contigua
- Los árboles binarios de búsqueda (ABB, para abreviar) son estructuras de datos parecidas a las listas enlazadas, y en las cuales podemos buscar en O(lgn)
  - Además las inserciones/eliminaciones se harán en O(lgn) también

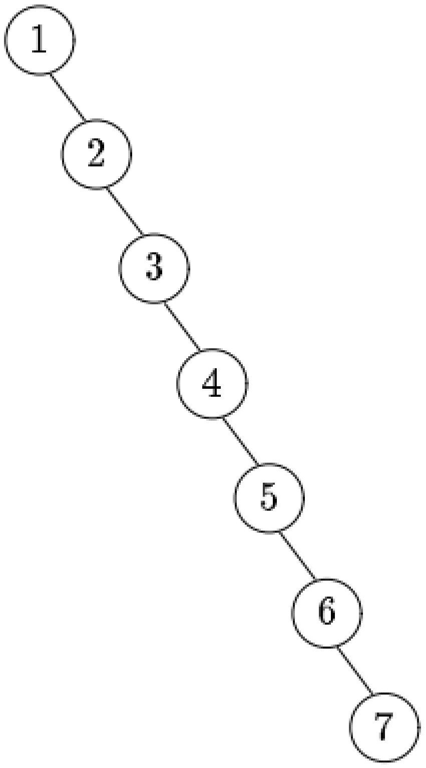
Definición de ABB III

- Un árbol binario de búsqueda es:
  - Un árbol vacío
  - Un árbol binario en el cual se cumple todo lo siguiente:
    - Cada elemento del subárbol izquierdo es menor que la raíz
    - Cada elemento del subárbol derecho es mayor que la raíz
    - Los subárboles izquierdo y derecho son también árboles binarios de búsqueda
- Si queremos admitir duplicados, modificar:
  - Menor por menor o igual, o
  - Mayor por mayor o igual
- El recorrido en inorden procesa los elementos en orden

Definición de ABB IV

- Ejemplos:





Implementación ABB

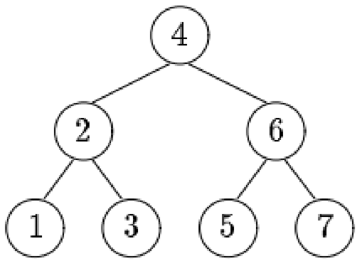
- Cada nodo apunta a sus dos hijos directos
  - También podemos añadir un puntero al padre para facilitar los algoritmos
- Es el mismo concepto que una lista enlazada, pero bidimensional en vez de unidimensional

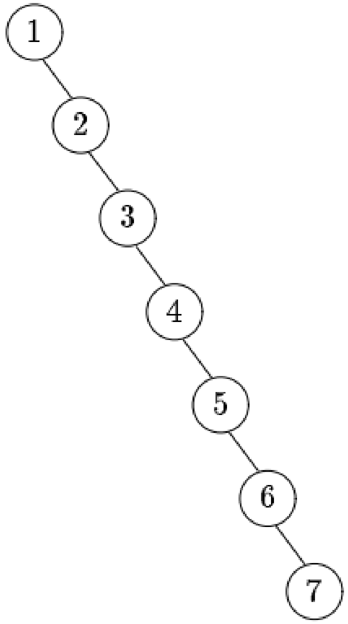
Buscar en ABB I

- Algoritmo de búsqueda:
  - Se compara el elemento buscado con la raíz. Si es igual hemos encontrado el elemento y por lo tanto hemos terminado.
  - Si el elemento buscado es menor que la raíz, se busca (llamada recursiva) en el subárbol izquierdo.
  - Si el elemento buscado es mayor que la raíz, se busca (llamada recursiva) en el subárbol derecho

Buscar en ABB II

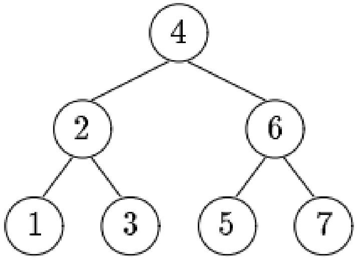
- Análisis temporal de buscar:
  - El peor caso es cuando tenemos que buscar en todos los niveles hasta encontrar el elemento buscado
  - En dicho peor caso, tardaremos más o menos según la topología del árbol binario de búsqueda:
    - Mejor caso en cuanto a topología: si el árbol binario de búsqueda es perfectamente equilibrado (primer ejemplo) el número de niveles es lgn, por lo tanto tardamos O(lgn)
    - Peor caso en cuanto a topología: si el árbol es totalmente lineal (segundo ejemplo), el número de niveles es n, por lo tanto tardamos O(n).
      - El árbol es como una lista enlazada





Insertar en ABB I

- Se busca la posición donde debe de estar el elemento y ahí se inserta
- Observamos que siempre se insertan hojas
- Ej: queremos insertar el 2,5.
  - La búsqueda nos dice que tenemos que ponerlo como hijo izquierdo de 3
- Ej: queremos insertar el 5,5
  - La búsqueda nos dice que tenemos que ponerlo como hijo derecho de 5
- Complejidad temporal: igual que buscar



Insertar en ABB II

- Análisis temporal en el peor caso, con la mejor o peor topología:
  - Primero hay que buscar el lugar en donde insertaremos
    - Mismo análisis y resultados que en buscar
      - O(lgn) con la mejor topología
      - O(n) con la peor topología
  - Una vez encontrado, insertar ahí una hoja es O(1)
  - Por lo tanto, mismas complejidades que en buscar
    - O(lgn) con la mejor topologías
    - O(n) con la peor topología

Insertar en ABB III

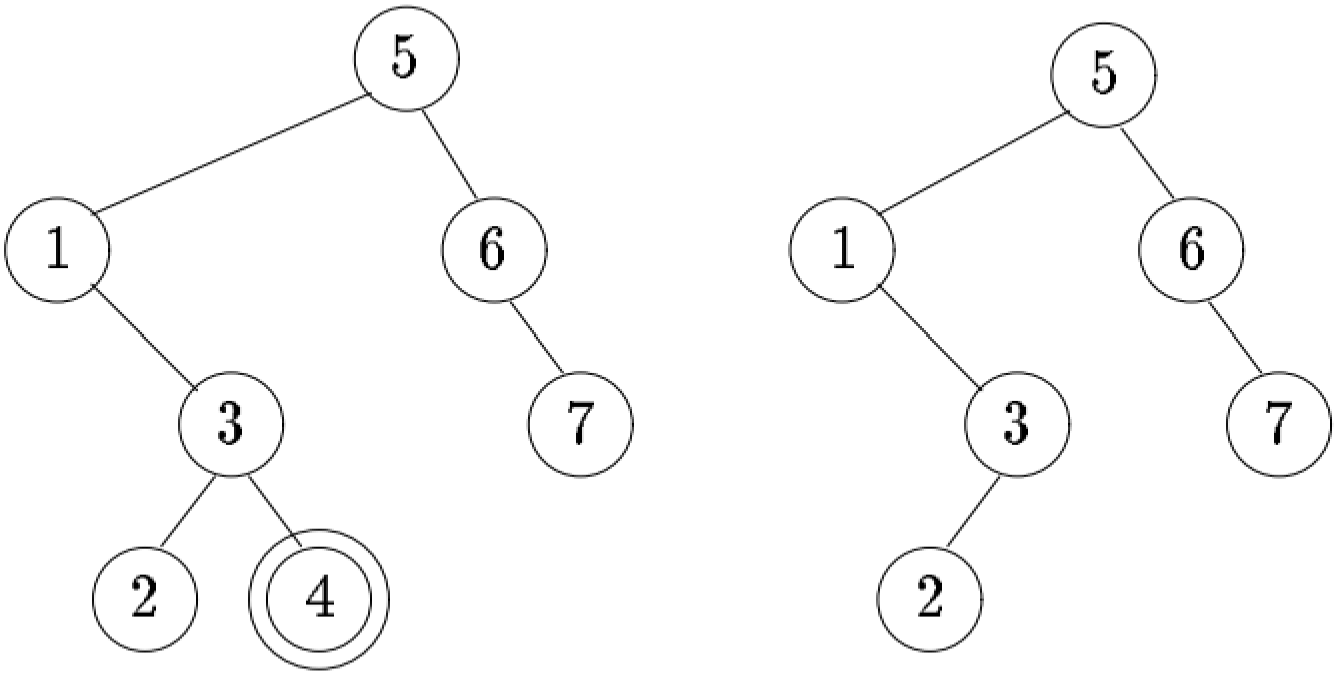
- ¿Cuál es el análisis temporal de insertar en el peor caso, pero con una topología media?
  - Habría que calcular la profundidad media de un árbol binario de búsqueda
    - Para ello, primero habría que calcular el número medio de hijos de cada nodo (número entre 0 y 2)
  - Se demuestra que, en el caso medio en cuanto a topología, insertar tiene una complejidad temporal de O(lgn) también (como en la mejor topología)
    - Lógicamente, con constantes ocultas mucho más altas que si el árbol binario de búsqueda fuera perfectamente equilibrado

Borrar en ABB I

- Buscamos el nodo a borrar
- Si es una hoja, se elimina sin más
- Si es un nodo interno:
  - Se sustituye por el mayor de los nodos del subárbol izquierdo (si hay subárbol izquierdo), o…
  - …se sustituye por el menor de los nodos del subárbol derecho (si hay subárbol derecho)
  - Habrá que borrar dicho nodo del subárbol del cual lo hemos sacado (llamada recursiva)

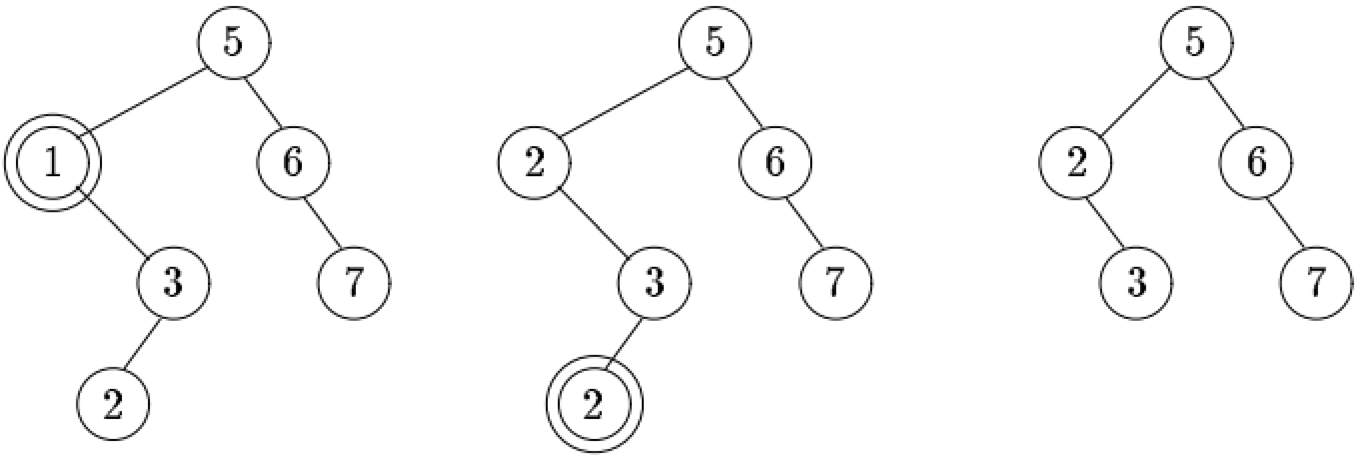
Borrar en ABB II

- Ejemplo: eliminar una hoja



Borrar en ABB III

- Ejemplo: eliminar el nodo interno “1”:

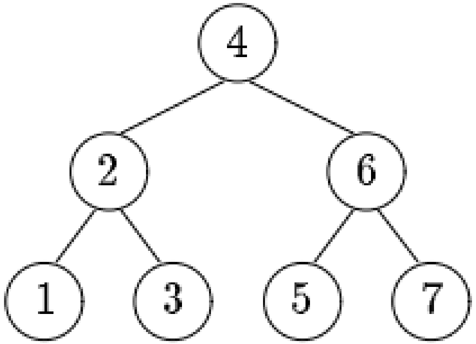


Borrar en ABB IV

- Cómo borrar sin deteriorar la topología del ABB (**actividad 7.3**):
  - Si un nodo sólo tiene un subárbol hijo (izquierdo o derecho), no nos queda más remedio que tomar ese subárbol
  - En caso de que el nodo tenga los dos subárboles hijos, es conveniente ir alternando: unas veces la sustitución la hacemos del subárbol derecho y otras veces del izquierdo
    - De este modo minimizamos la probabilidad de que el árbol se vaya convirtiendo poco a poco en lineal (el peor caso para las búsquedas)
    - Lo ideal sería sustituir en el subárbol que tuviera más altura, pero para eso tendríamos que analizar cada subárbol y eso nos costaría tiempo

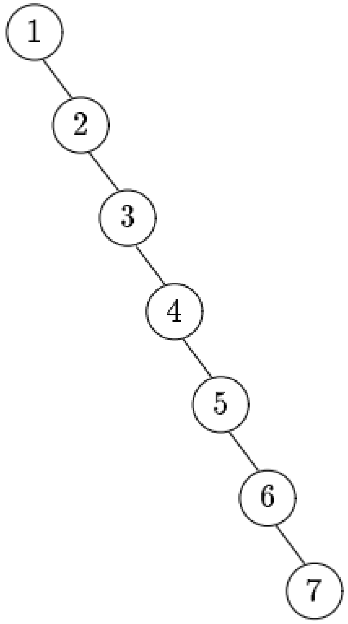
Borrar en ABB V

- Análisis temporal de eliminar (peor caso) cuando el ABB es perfectamente equilibrado (la mejor topología):
  - Si el ABB es perfectamente equilibrado, la primera vez que busquemos el máximo/mínimo lo encontraremos en una hoja
    - Encontrar dicha hoja es O(lgn) porque está en el último nivel
    - Eliminar una hoja es O(1), y no es necesario volver a hacer una llamada recursiva
    - Por lo tanto, tendremos O(lgn)



Borrar en ABB VI

- Análisis temporal de eliminar (peor caso) cuando el ABB tiene topología lineal (la peor topología):
  - Si el árbol es lineal, cada vez que busquemos el máximo/mínimo en un subárbol izquierdo/derecho lo encontraremos en su raíz
    - Por lo tanto la búsqueda tardará O(1) en vez de O(n)
    - Es necesario volver a hacer otra llamada recursiva, que tardará T(n-1)
    - Tenemos por lo tanto T(n) = 1 + T(n-1), lo cual, operando, nos da O(n)



Borrar en ABB VII

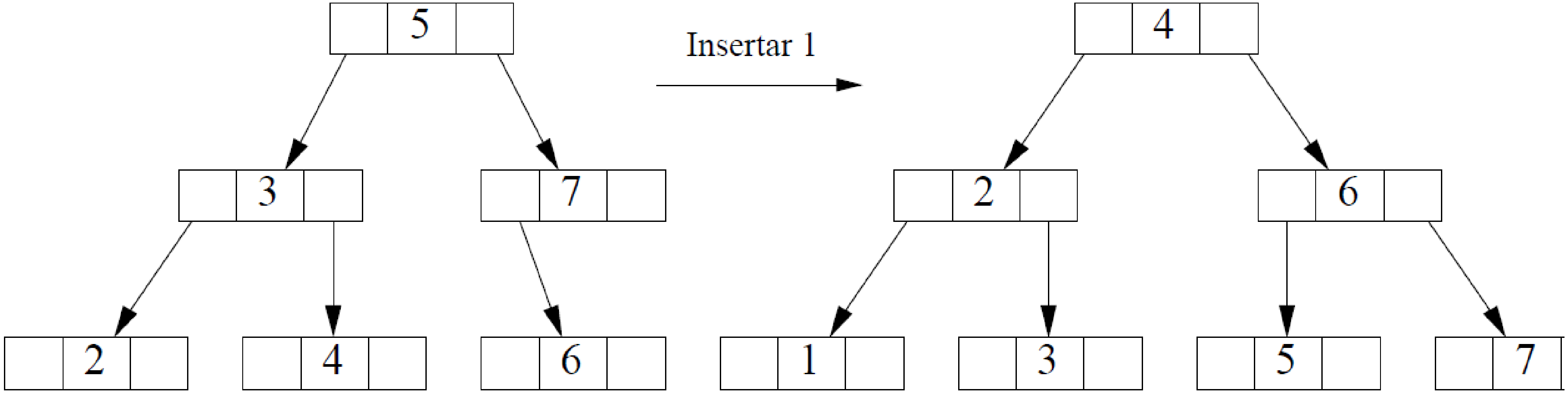
- Análisis temporal de eliminar (peor caso) cuando el ABB tiene una topología media:
  - Al igual que con insertar, habría que calcular la profundidad media de un árbol binario de búsqueda
    - Para ello, al igual que con insertar, primero habría que calcular el número medio de hijos de cada nodo (número entre 0 y 2)
  - Se demuestra que, en el caso medio en cuanto a topología, tanto insertar como eliminar tienen una complejidad temporal de O(lgn) también (como en la mejor topología)
    - Lógicamente, con constantes ocultas mucho más altas que si el árbol binario de búsqueda fuera perfectamente equilibrado

AVL I

- Como hemos visto, la mejor topología es cuando el árbol binario de búsqueda es un árbol perfectamente equilibrado
  - O, al menos, un árbol binario completo
- ¿Podríamos modificar el algoritmo de insertar y eliminar para que el árbol se mantuviese siempre con esta topología?
  - De este modo buscar siempre sería O(lgn)
  - Además, nos gustaría que insertar y eliminar siguieran siendo O(lgn)

AVL II

- Ejemplo de lo que tendría que pasar al insertar un "1", para mantener el árbol con una buena topología:

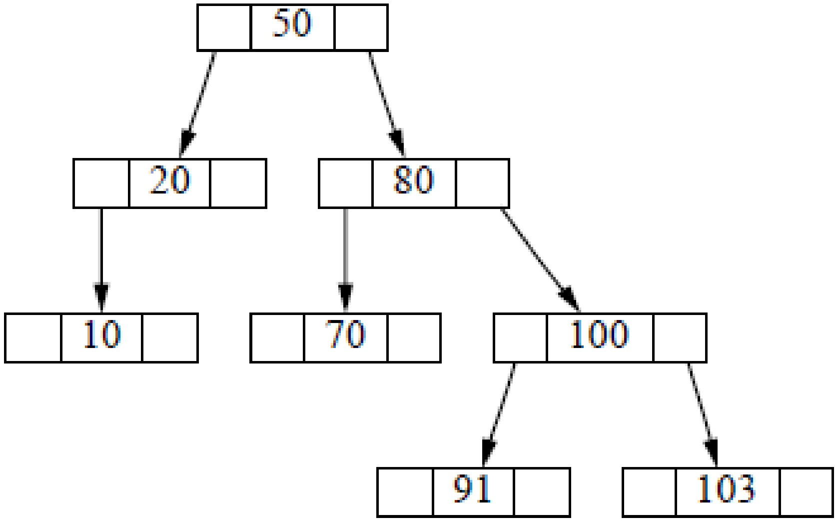


AVL III

- Los árboles AVL resuelven este problema
  - Sus siglas son las primeras letras de sus creadores (1962): Adelson-Velskii y Lands
- Definición: un árbol AVL es un árbol binario de búsqueda en el que para cada uno de sus nodos, las alturas de sus dos subárboles difieren como mucho en 1 en valor absoluto
  - Un AVL será “casi” un árbol binario perfectamente equilibrado
    - Ese “casi” es despreciable cuando el número de nodos es muy grande

AVL IV

- Asociaremos a cada nodo un factor de equilibrio $F_E = H_D - H_I$, que será igual a la diferencia de las alturas del subárbol derecho (H_D) y del izquierdo (H_I)
- Por lo tanto, en un árbol AVL, cada nodo tendrá únicamente -1, 0 ó 1 como posibles valores para F_E
- Asumimos, para nuestros cálculos, que la altura del árbol vacío es -1
- Ejercicio: ¿cuál es la altura y el factor de equilibrio de cada uno de los nodos del siguiente árbol?



AVL V

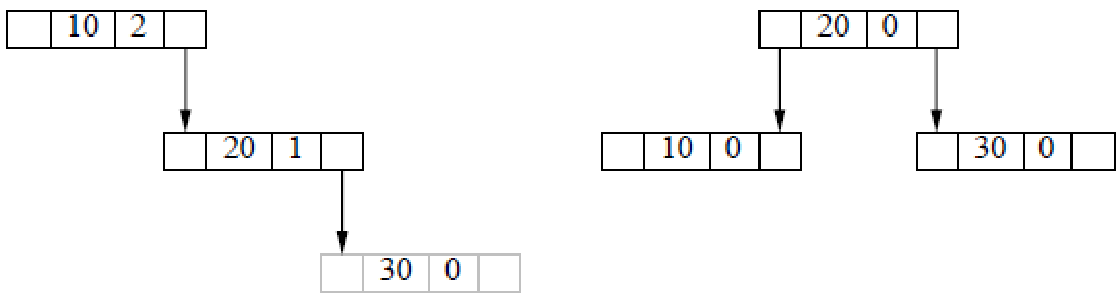
- Implementación de un AVL:
  - Un AVL es un tipo especial de ABB (hereda de ABB)
  - Ahora los nodos del AVL contendrán:
    - Contenido (como antes)
    - Puntero al padre y a los dos hijos (como antes)
    - Altura actual del nodo
    - Factor de equilibrio del nodo

AVL VI

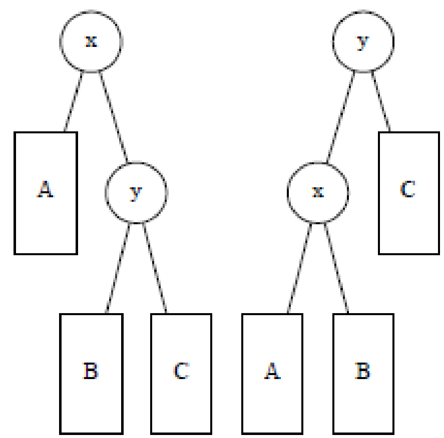
- Inserción:
  - Se inserta como en cualquier ABB. Recordemos que siempre se inserta una hoja en los ABB. Puede que el árbol deje de ser AVL
  - Ir ascendiendo por el árbol, desde el elemento insertado, hasta encontrar un nodo con F_E = 2 ó -2
  - Reequilibramos el subárbol que empieza en dicho nodo, realizando una “rotación”. Habrá cuatro casos:
    - Rotación simple a la izquierda
    - Rotación simple a la derecha
    - Rotación compuesta derecha-izquierda
    - Rotación compuesta izquierda-derecha
  - Una vez reequilibrado, el F_E de ese nodo ha cambiado (se ha quedado en 0, 1 ó -1), y por lo tanto su altura puede que también
    - Por lo tanto es posible que también cambie la altura y consecuentemente el F_E de sus ascendentes.
    - Por lo tanto tenemos que seguir ascendiendo hasta la raíz para ir actualizando la altura y el F_E de todos sus ascendentes
  - Al insertar, se realizará una rotación como máximo (no lo demostramos)

AVL VII

- Rotación simple a la izquierda

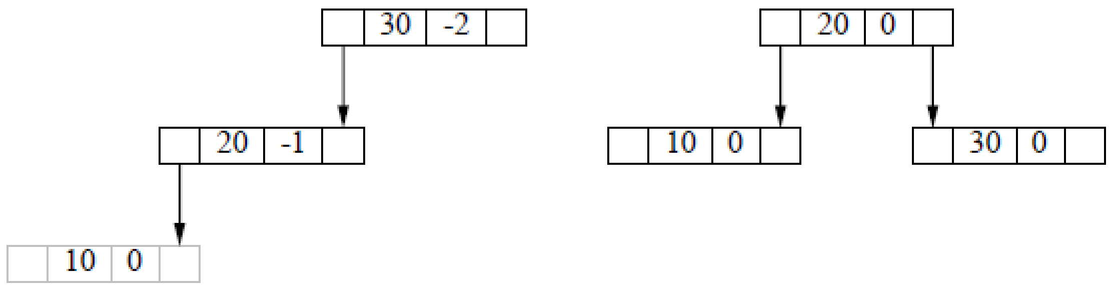


Cuando un nodo tiene factor de equilibrio de 2 y su hijo derecho 1

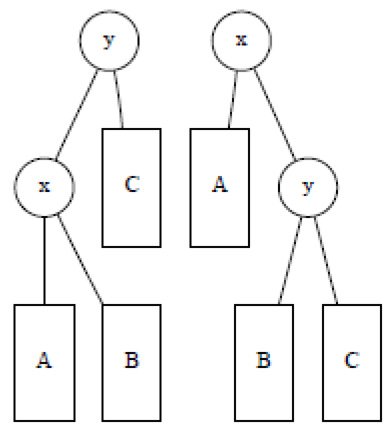


AVL VIII

- Rotación simple a la derecha

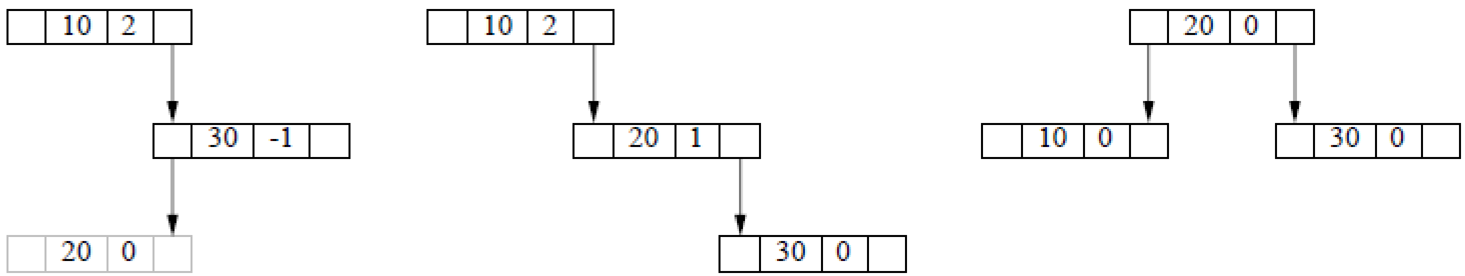


Cuando un nodo tiene factor de equilibrio de -2 y su hijo izquierdo -1

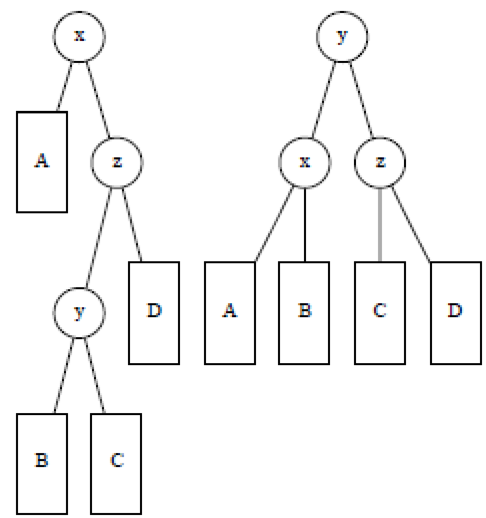


AVL IX

- Rotación compuesta derecha-izquierda

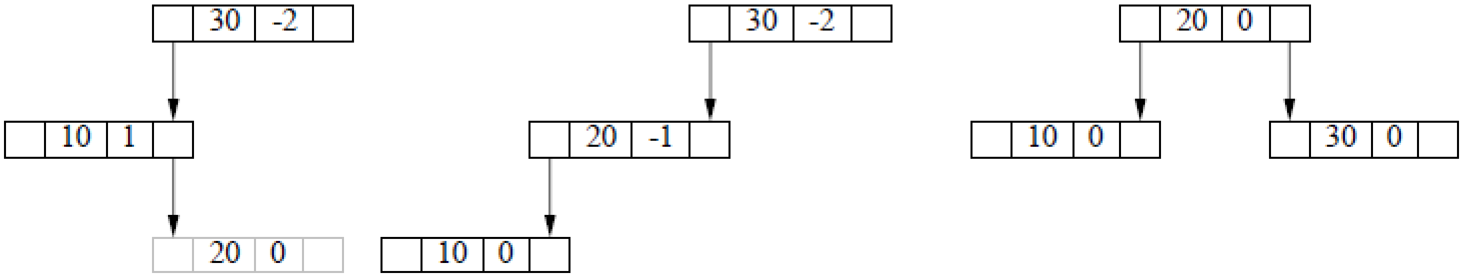


- Cuando el factor de equilibrio de un nodo es 2, y el de su hijo derecho -1

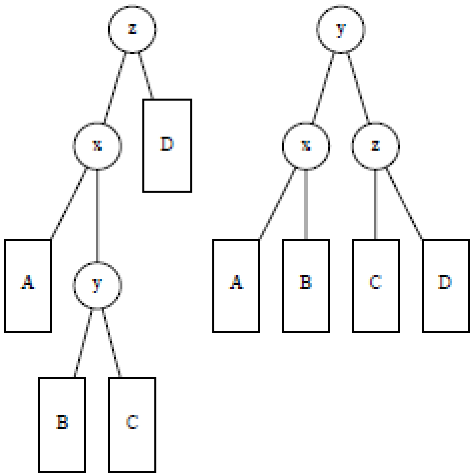


AVL X

- Rotación compuesta izquierda-derecha



- Cuando el factor de equilibrio de un nodo es -2, y el de su hijo izquierdo 1



AVL XI

- Análisis de insertar (actividad 7.4):
  - Inicialmente insertamos como en cualquier ABB, por lo tanto este paso es O(lgn). Recordemos que siempre insertamos una hoja en un ABB.
  - Ahora tenemos que ir ascendiendo hasta la raíz (lgn niveles). Por cada nodo, hacemos una de las siguientes tres acciones:
    - Al principio, no hacemos nada especial, por lo tanto tardamos O(1) en ese nodo
    - Cuando encontramos el nodo con un factor de equilibrio incorrecto, realizamos una rotación para corregirlo. Hacer una rotación es sólo cambiar un número limitado de punteros, por lo tanto es O(1)
    - A partir de entonces, actualizamos la altura y factor de equilibrio de los nodos siguientes. Esto también nos cuesta un tiempo de O(1)
  - Por lo tanto:
    - Primera fase de insertar: O(lgn)
    - Segunda fase para ir ascendiendo por el árbol: O(lgn)
  - Por lo tanto, concluimos que el algoritmo entero es O(lgn)
    - Igual que en un árbol binario de búsqueda que sea perfectamente equilibrado
    - Pero obviamente con constantes ocultas más altas, pues tenemos que “bajar hasta abajo” (para insertar la hoja) y luego “subir hasta arriba” (para reequilibrar el árbol)

AVL XII

- Eliminar:
  - Se elimina como en cualquier árbol binario de búsqueda. Es posible que el árbol deje de ser AVL
  - Al eliminar en un árbol binario de búsqueda, al final, después de todas las llamadas recursivas, recordemos que acabamos eliminando una hoja (caso trivial)
  - Desde el padre de dicha hoja eliminada, vamos ascendiendo por el árbol y reequilibramos todos los nodos cuyo $F_E = 2$ ó -2
    - En insertar sólo hacíamos un reequilibrio, pero en eliminar podemos llegar a hacer varios (no lo demostramos)
  - El reequilibrado tiene dos casos más que en insertar. Los dos nuevos casos son:
    - Un nodo tiene $F_E = 2$ y su hijo derecho $F_E = 0$. Se arregla con una rotación simple a la izquierda
    - Un nodo tiene $F_E = -2$ y su hijo izquierdo $F_E = 0$. Se arregla con una rotación simple a la derecha

AVL XIII

- Análisis de eliminar:
  - Primero eliminamos el nodo como en cualquier ABB, por lo tanto O(lgn) en tiempo
  - El análisis del reequilibrado es parecido que en insertar: tenemos que ir ascendiendo hasta la raíz
    - Por lo tanto esta fase también será O(lgn)
    - Ahora podemos hacer varias rotaciones en vez de una, así pues las constantes ocultas aumentarán
      - Recordemos que cada rotación es O(1)
  - Por lo tanto, concluimos que el algoritmo “eliminar” entero es O(lgn)