Estructuras de Datos
Clase 20 – Árboles de búsqueda

Dr. Sergio A. Gómez
http://cs.uns.edu.ar/~sag

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, Argentina

Motivaciones

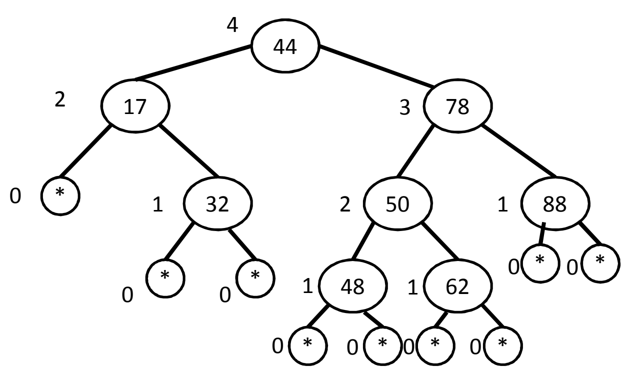
- El árbol binario de búsqueda permite implementar conjuntos y mapeos con un tiempo de operaciones buscar, insertar y eliminar con orden logarítmico en la cantidad de elementos en promedio.
- En el peor caso las operaciones tienen orden lineal en la cantidad de elementos (cuando las inserciones se realizaron en forma ascendente o descendente en cuyo caso el árbol degenera en una lista).
- Hay estructuras alternativas que garantizan tiempo de acceso de orden logarítmico en la cantidad de elementos y se los conoce como árboles de búsqueda balanceados: AVL y Árbol 2-3.

Árboles AVL

- Agregaremos una corrección al árbol binario de búsqueda para mantener una altura del árbol proporcional al logaritmo de la cantidad de nodos del árbol.
- Recordemos que el tiempo de búsqueda, inserción y borrado en un árbol binario de búsqueda es lineal en la altura del árbol.
- Entonces, si n=cantidad de elementos de un arbol T, tendríamos así que $T(n) = O(\log_2(n))$.
- <u>Propiedad del balance de la altura:</u> Para cada nodo interno v de T, las alturas de los hijos difieren en a lo sumo 1.
- Cualquier árbol binario de búsqueda que satisface esta propiedad se dice "árbol AVL" (por Adel'son-Vel'skii y Landis).

Árboles AVL: Ejemplo

- <u>Propiedad del balance de la altura:</u> Para cada nodo interno v de T, las alturas de los hijos difieren en a lo sumo 1.
- <u>Ejemplo:</u> Los * corresponden a nodos nulos (con altura 0 de acuerdo a GT).



Comentarios

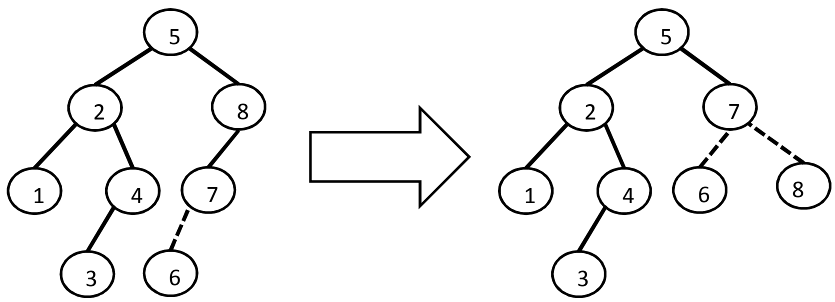
- Siguiendo a Goodrich & Tamassia diremos que los nodos nulos tienen altura 0.
- Como un AVL es un árbol binario de búsqueda, la operación de búsqueda no sufre alteraciones.
- La únicas operaciones a modificar son la de inserción y eliminación, las cuales deben verificar que se cumpla la propiedad de balance al finalizar la operación.
- Luego de cada modificación en un nodo, rebalancearán los hijos de dicho nodo en O(1) por medio de las llamadas "rotaciones".
- Las modificaciones se hacen desde la hoja donde se insertó el nodo hacia la raíz siguiendo el camino de llamadas recursivas.
- Las eliminaciones las haremos perezosas (lazy): marcaremos los datos eliminados con un booleano.

Rotaciones

Son cuatro correspondientes a las cuatro combinaciones para la inserción de una clave a partir de un nodo raíz del subárbol considerado:

1) izquierda – izquierda: rotación simple de izquierda a derecha
2) izquierda – derecha: Rotación doble de izquierda a derecha
3) derecha – derecha: Rotación simple de derecha a izquierda
4) derecha – izquierda: Rotación doble de derecha a izquierda

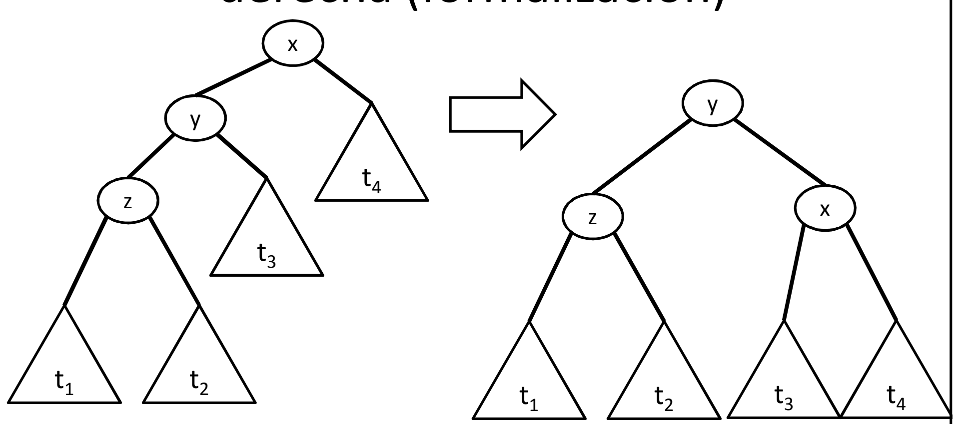
Ejemplo de rotación simple de izquierda a derecha



Antes de rotar Después de rotar

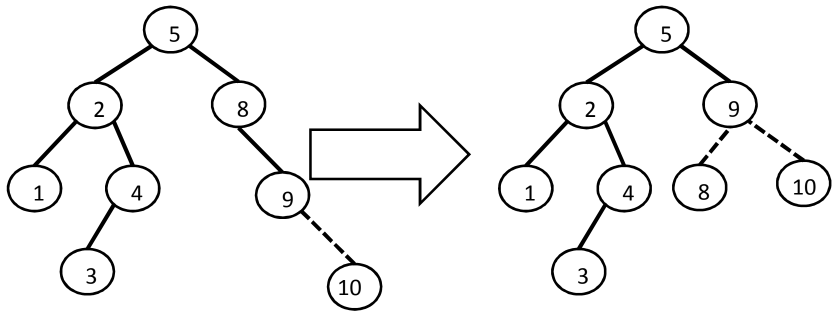
La inserción del 6 destruye la propiedad de AVL en el nodo 8, lo que se resuelve con una rotación simple de izquierda a derecha (tomado de Mark Allen Weiss, Data Structures and Algorithm Analysis in Java, Third Edition).

(1) Rotación simple de izquierda a derecha (formalización)



Corresponde a dos inserciones seguidas hacia la izquierda desde la raíz del subárbol considerado

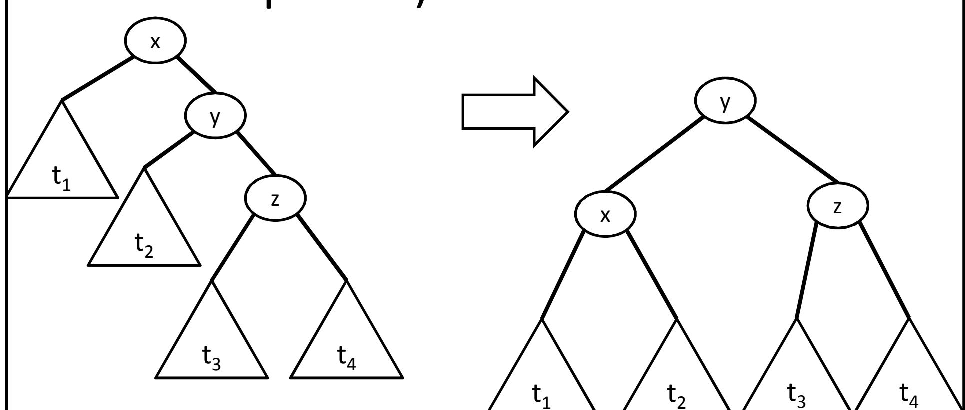
Ejemplo de rotación de derecha a izquierda



Antes de rotar Después de rotar

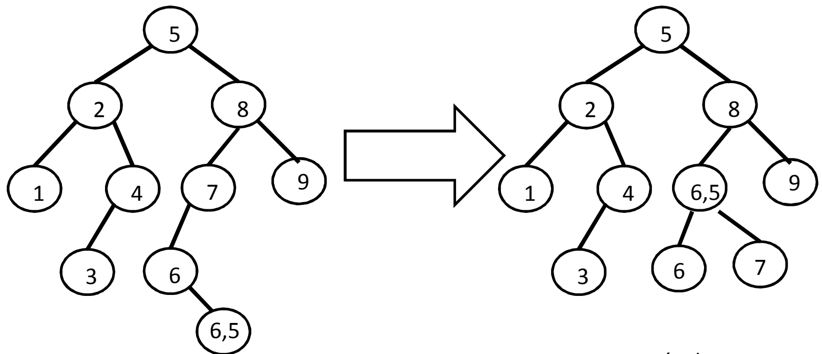
La inserción del 10 destruye la propiedad de AVL en el nodo 8, lo que se resuelve con una rotación simple de derecha y izquierda.

(3) Rotación der-der (simple derecha a izquierda): Formalización



Corresponde a dos inserciones seguidas hacia la derecha de la raíz del subárbol considerado

Ejemplo de rotación doble de izquierda a derecha

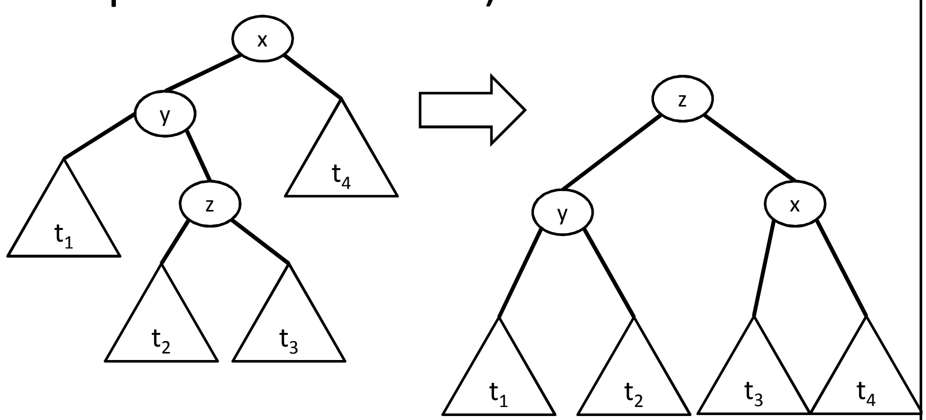


Antes de rotar

Después de rotar

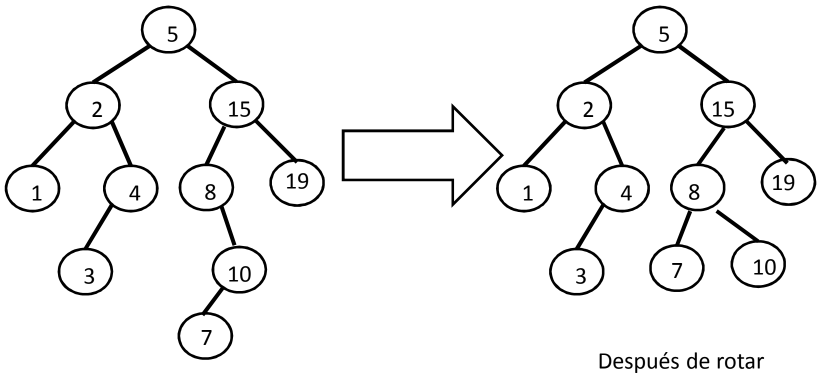
La inserción del 6,5 destrute la propiedad de AVL en el nodo 7, entonces hay que rotar.

(2) Rotación izq-der (doble de izquierda a derecha): Formalización



Corresponde a una inserción en el hijo izquierdo y luego en hijo derecho del hijo izquierdo de la raíz del subárbol considerado

Ejemplo de rotación doble de derecha a izquierda

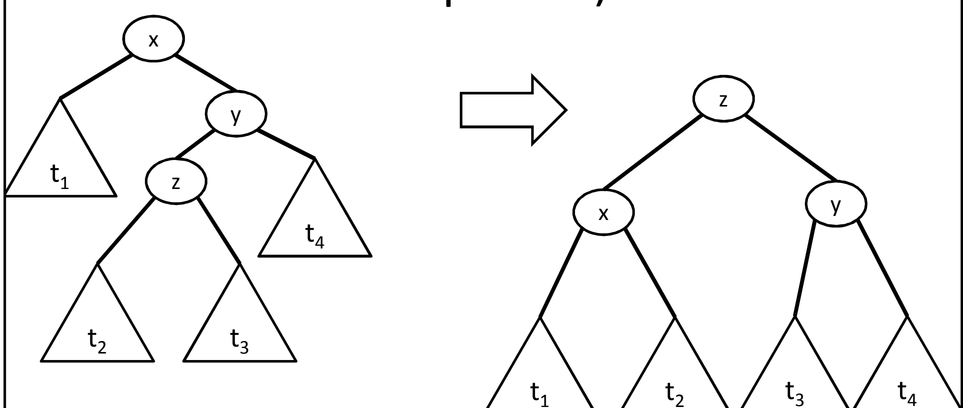


Después de rotar

Antes de rotar

La inserción del 7 destrute la propiedad de AVL en el nodo 8, entonces hay que rotar.

(4) Rotación der-izq (doble de derecha a izquierda)



Corresponde a una inserción en el hijo derecho de la raíz y luego en el hijo izquierdo del hijo derechos de la raíz del subárbol considerado

Implementación Java de la Inserción

La única diferencia con el NodoABB es que en cada nodo mantengo la altura de dicho nodo.

```
// Archivo: NodoAVL.java
public class NodoAVL<E>
{
	private NodoAVL<E> padre;
	private E rotulo;
	private int altura;   private boolean eliminado; // ←
diferencia!
	private NodoAVL<E> izq, der;

	public NodoAVL<E>(E rotulo){
		altura = 0;
		// Al crear un nodo dummy anoto que su altura es 0.
		...    // Resto Idem NodoABB}
	// setters y getters incluyendo la altura y eliminado
}
```



```
public class AVL<E>
{
	NodoAVL<E> raiz;
	Comparator<E> comp;

	public AVL(Comparator<E> comp)
	{
		raiz = new NodoAVL<E>(null);
		this.comp = comp;
	}

	public void insert(E x)
	{
		insertaux( raiz, x );
	}

	private int max(int i, int j )
	{
		return i>j ? i : j;
	}
```

```
private void insertaux( NodoAVL<E> t, E item ) {
  if( t.getRotulo() == null )  {
            t.setRotulo( item );   t.setAltura( 1 );   t.setIzq( new NodoAVL<E>( null ) );
            t.setDer( new NodoAVL<E>( null ) );     t.getIzq().setPadre( t );
            t.getDer().setPadre( t );
  }  else   {
      int comparacion = comp.compare( item, t.getRotulo() );
      if( comparacion == 0 )      t.setRotulo( x ); // nada mas cambia
      else  if( comparacion < 0 ) {                                          {
            insertaraux( t.getLeft(), item );
            if( Math.abs( t.getLeft().getAltura() - t.getRight().getAltura() ) > 1 ) {
            // Rebalancear mediante rotaciones:  testeo por rotaciones (i) o (ii)
            // Si estoy aca => item < x, debo testear si (item < y) o (item > y)
            // si item < y => rotacion (i);      si item > y => rotacion (ii)
            E x = t.getRotulo();   // no se usa, es solo para la explicación
            E y = t.getLeft().getRotulo();
            E z = t.getLeft().getLeft().getRotulo(); // no se usa, es solo para la explicación
            int comp_item_y = comp.compare( item, y );
            if( comp_item_y < 0 )  rotacion_i(t);   // item < y => rotacion (i)
            else  rotacion_ii(t);  // item > y => rotacion (ii)
    }  else {   /* Caso simétrico pero insertando hacia la derecha y luego testeo por
rotaciones (iii) y (iv) */
            } t.setAltura( max(t.getLeft().getAltura(), t.getRight().getAltura()) + 1 );
 } }
```


Complejidad temporal de la inserción

- Noten que las rotaciones se hacen en los nodos del camino desde la raíz hasta la hoja donde se insertó la nueva clave.
- Como las rotaciones se implementan con asignaciones de referencias (posiciones), cada rotación se hace en tiempo constante.
- La cantidad de rotaciones es del orden de la altura del árbol.
- La altura es proporcional al logaritmo base 2 de la cantidad de nodos del árbol.
- Por lo tanto, el tiempo de insertar es del orden del logaritmo de la cantidad de nodos del árbol.

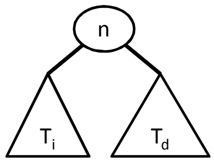
Árboles 2-3: Definiciones

Un "árbol 2-3" es un árbol tal que cada nodo interno (no hoja) tiene dos o tres hijos, y todas las hojas están al mismo nivel.

La definición recursiva es: T es un árbol 2-3 de altura h si:

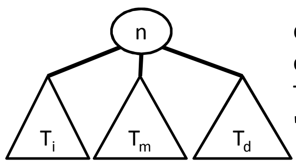
a) T es vacío (es decir de altura -1)

b) T es de la forma:



donde n es un nodo y T_i y T_d son árboles 2-3 cada uno de altura h-1.
T_i se dice "subárbol izquierdo" y T_d "subárbol derecho".

c) T es de la forma:



donde n es un nodo y T_i, T_m y T_d son árboles 2-3 cada uno de altura h-1.
T_i se dice "subárbol izquierdo“, T_m se dice "subárbol medio“ y T_d "subárbol derecho".

Árboles 2-3: Definiciones

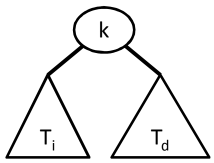
- <u>Propiedad:</u> Si un árbol 2-3 no contiene ningún nodo con 3 hijos entonces su forma corresponde a un árbol binario lleno.

Árboles 2-3: Definiciones

Un arbol 2-3 es un "arbol 2-3 de búsqueda" si T es un árbol 2-3 tal que

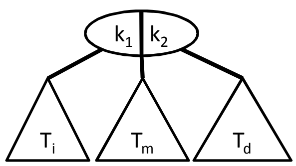
a) T es vacío

b) T es de la forma:



n contiene una clave k, y
k es mayor que las claves de T_i
k es menor que las claves de T_d
T_i y T_d son árboles 2-3 de búsqueda

c) T es de la forma:



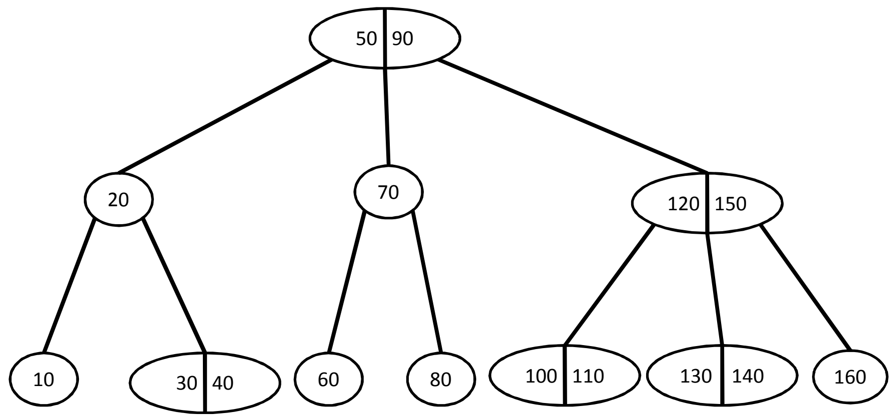
n contiene dos claves k_1 y k_2, y
k_1 es mayor que las claves de T_i,
k_1 es menor que las claves de T_m
k_2 es mayor que las claves de T_m
k_2 es menor que las claves de T_d
T_i, T_m y T_d son árboles 2-3 de búsqueda

Árboles 2-3: Definiciones

<u>Reglas para ubicar entradas en los nodos de un árbol 2-3 de búsqueda:</u>

1) Si n tiene dos hijos, entonces contiene una entrada
2) Si n tiene tres hijos, entonces contiene dos entradas
3) Si n es una hoja, entonces contiene una o dos entradas

Ejemplo de árbol 2-3



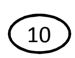
Búsqueda en un árbol 2-3

```
Recuperar( T, clave ) --> valor
Sea R la raiz de T
SI clave está en R ENTONCES  RETORNAR valor igual a valor asociado a la entrada
SINO SI R es una hoja ENTONCES RETORNAR nulo { falla }
SINO
    SI R tiene una entrada ENTONCES
            Sea k la clave de R
            SI clave < k ENTONCES  RETORNAR Recuperar( Ti(T), clave )
            SINO RETORNAR Recuperar( Td(T), clave )
 SINO
    SI R tiene dos entradas ENTONCES
            Sean k1 y k2 las claves de R
            SI clave < k1 ENTONCES RETORNAR Recuperar( Ti(T), clave )
            SINO SI clave < k2 ENTONCES
                        RETORNAR Recuperar( Tm(T), clave )
                   SINO
                        RETORNAR Recuperar( Td(T), clave )
```

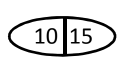
Inserción

- Igual que en el ABB, siempre se inserta en una hoja siguiendo el camino de la búsqueda.
- Si la hoja tiene 2 claves, terminamos.
- Si la hoja tiene 3 claves, se produce un rebalse (overflow) y se debe partir el nodo en 2 nodos, la clave del medio sube al padre, quien queda a cargo de administrar ese hijo que se duplicó y ver dónde ubicar esa clave.
- Si el padre tenía 1 clave y 2 hijos, no hay problema, porque ahora tendrá 2 claves y 3 hijos.
- Si el padre ya tenia 2 claves y 3 hijos, ahora pasaría a tener 3 claves y 4 hijos, lo cual no puede ocurrir. Entonces el proceso anterior se repite.
- El proceso termina cuando terminamos en un nodo intermedio (con 2 claves o 3 hijos) o llegamos a crear una nueva raíz y el árbol crece 1 nivel.
- Como este proceso tiene T(h) = O(h), ocurre que T(n) = O(log(n)).

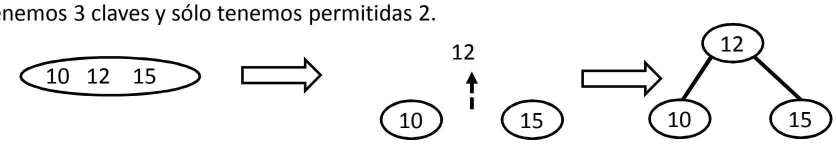
1) Insertamos 10 en el árbol vacío:



2) Insertamos 15: como hay lugar en la única hoja, allí ponemos la nueva clave y terminamos

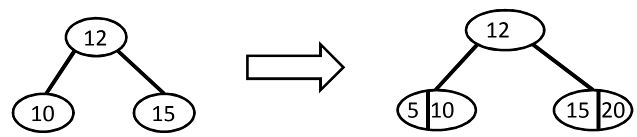


3) Insertamos 12: vamos a la única hoja y ponemos el 12 allí, pero hay rebalse porque tenemos 3 claves y sólo tenemos permitidas 2.

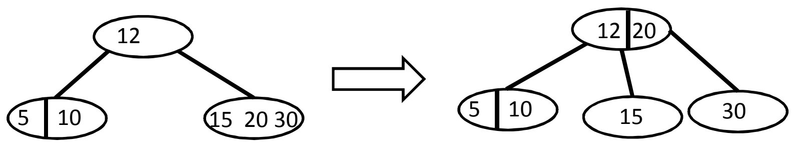


Partimos el nodo en 2 nodos dividiendo las claves y le pasamos al padre los 2 nodos junto con la clave del medio. Como no hay padre, el árbol crece en un nivel al crear un nuevo nodo para acomodar la clave con los 2 nodos como sus hijos.

4) Cualquier clave que inserte, siempre va a una hoja siguiendo el criterio de búsqueda. Inserto 20 y 5. Como no hay rebalse porque había lugar en las hojas, terminamos.



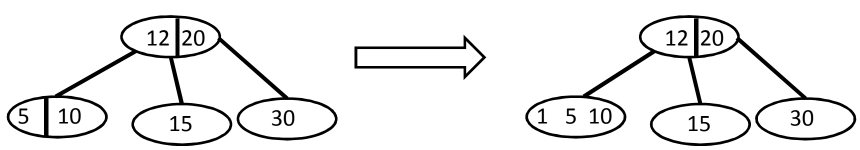
5) Inserto 30, el cual termina en el hijo derecho de 12.



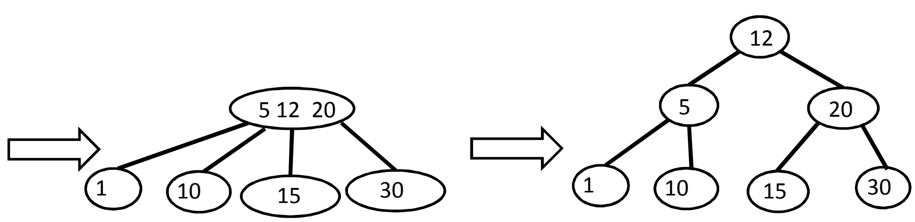
Hay rebalse porque tengo 3 claves y solo tengo permitidas 2

Parto el nodo rebalsado en 2 nodos y se los paso a su padre junto con la clave del medio (que es el 20) y terminamos porque la raíz tiene lugar para otra clave y otro hijo extra.

6) Inserto 1 en el árbol del paso (5):



Hay rebalse



Parto el nodo y subo el 5 al padre (que es la raíz). Pero ahora la raíz tiene rebalse (3 claves y 4 hijos, situación no permitida).

Parto el nodo raíz en 2 nodos y subo el 12 creando una nueva raíz

RESUMEN:
Para insertar un valor X en un árbol 2-3,
primero hay que ubicar la hoja L en la cual X terminará.
Si L contiene ahora dos valores, terminamos.
Si L contiene tres valores, hay que partirla en dos hojas
L1 y L2. L1 se queda con el valor más pequeño, L2 con el
más grande, y el del medio se manda al padre P de L.
Los nodos L1 y L2 se convierten en los hijos de P.

Si P tiene sólo 3 hijos (y 2 valores), terminamos.
En cambio, si P tiene 4 hijos (y 3 valores), hay
que partir a P igual que como hicimos con una hoja
sólo que hay que ocuparse de sus 4 hijos. Partimos a P
en P1 y P2, a P1 le damos la clave más pequeña
y los dos hijos de la izquierda y a P2 le damos
la clave más grande y los dos hijos de la derecha.

Luego de esto, la entrada que sobra se manda al padre de P
en forma recursiva. El proceso termina cuando la entrada
sobrante termina en un nodo con dos entradas o el árbol crece
1 en altura (al crear una nueva raíz).

29

Bibliografía

- Capítulo 10, Secciones 2 y 4 de M. Goodrich & R. Tamassia, Data Structures and Algorithms in Java. Fourth Edition, John Wiley & Sons, 2006.
- <u>Árboles 2-3:</u> Basado en Paul Helman & Robert Veroff. Intermediate Problem Solving and Data Structures. Walls and Mirrors, Benjamming Cummings, Menlo Park, 1986.

Estructuras de datos - Dr. Sergio A. Gómez 30