Árboles de Búsqueda Balanceados

Balanced Search Trees

Motivación

- Los árboles binarios de búsqueda logran muy buen desempeño promedio, pero su peor caso es lineal (~N).
- Resulta de mucho interés encontrar estructuras que aún en el peor caso tengan un desempeño cercano a ~lg(N).
- Este objetivo se logra balanceando el árbol, de modo que todos los caminos desde la raíz sean de similar longitud.

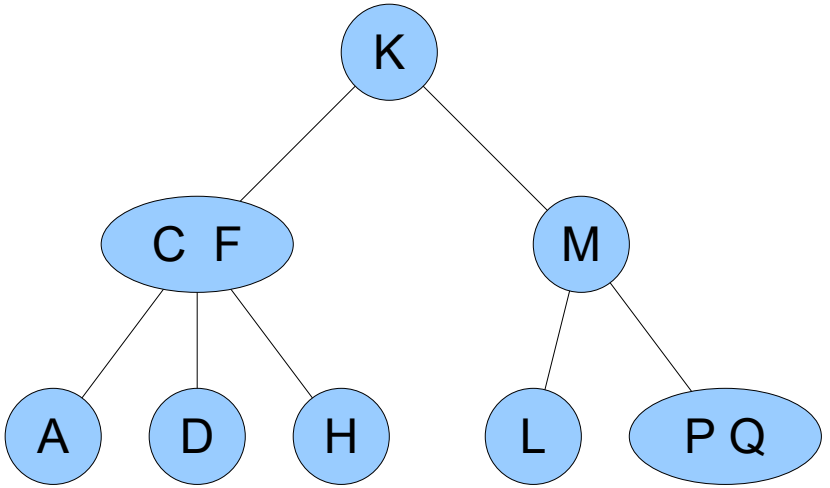
Árboles de búsqueda 2-3

Definición

Es un árbol que es o bien vacío o se compone de:

- Un 2-node que contiene una llave y dos enlaces, el enlace izquierdo a un árbol 2-3 con llaves menores y el enlace derecho a un árbol 2-3 con llaves mayores.
- Un 3-node que contiene dos llaves y 3 enlaces. El enlace izquierdo a un árbol 2-3 con llaves menores, el enlace del medio a un árbol 2-3 con llaves entre las dos llaves del nodo, y el enlace derecho a un árbol 2-3 con llaves mayores.

Ejemplo árbol 2-3



Árbol con 10 llaves.
Logitud de los caminos raíz-hojas : 2

Árboles 2-3

- Se dice que el árbol es balanceado si todas las hojas se encuentran a igual distancia (en aristas) de la raíz.
- Por defecto, se asume que el árbol siempre está balanceado. Las operaciones `put` / `delete` se pueden implementar de forma que se garantice esta condición.

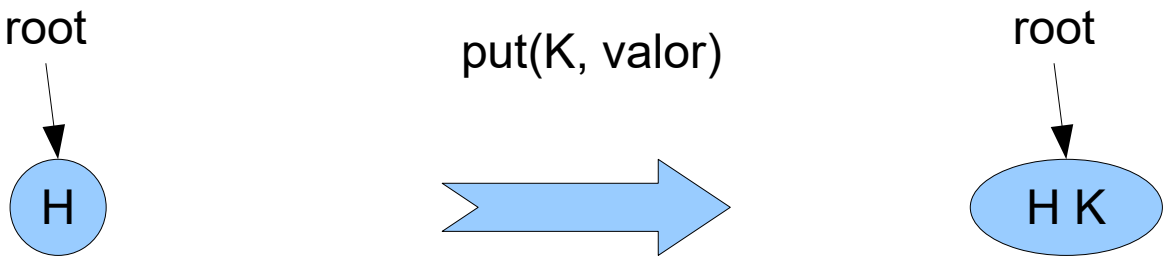
Búsqueda en árboles 2-3

- El algoritmo de búsqueda es esencialmente el mismo:
  - Se compara la llave a buscar con la(s) llave(s) en la raíz.
  - Si es menor, se sigue recursivamente el enlace izquierdo,
  - Si es 3-node y está entre las dos llaves, se sigue recursivamente el enlace central, o
  - Si es mayor, se sigue recursivamente el enlace derecho.

Insert: Caso de un 2-node

- El proceso de inserción se basa en la misma idea del árbol de búsqueda: Hacer una búsqueda hasta llegar a un enlace nulo.
- Si el nodo donde termina la búsqueda es un 2-node, se reemplaza por un 3-node. Si inicialmente el árbol estaba balanceado, luego de esta operación el árbol también es balanceado.

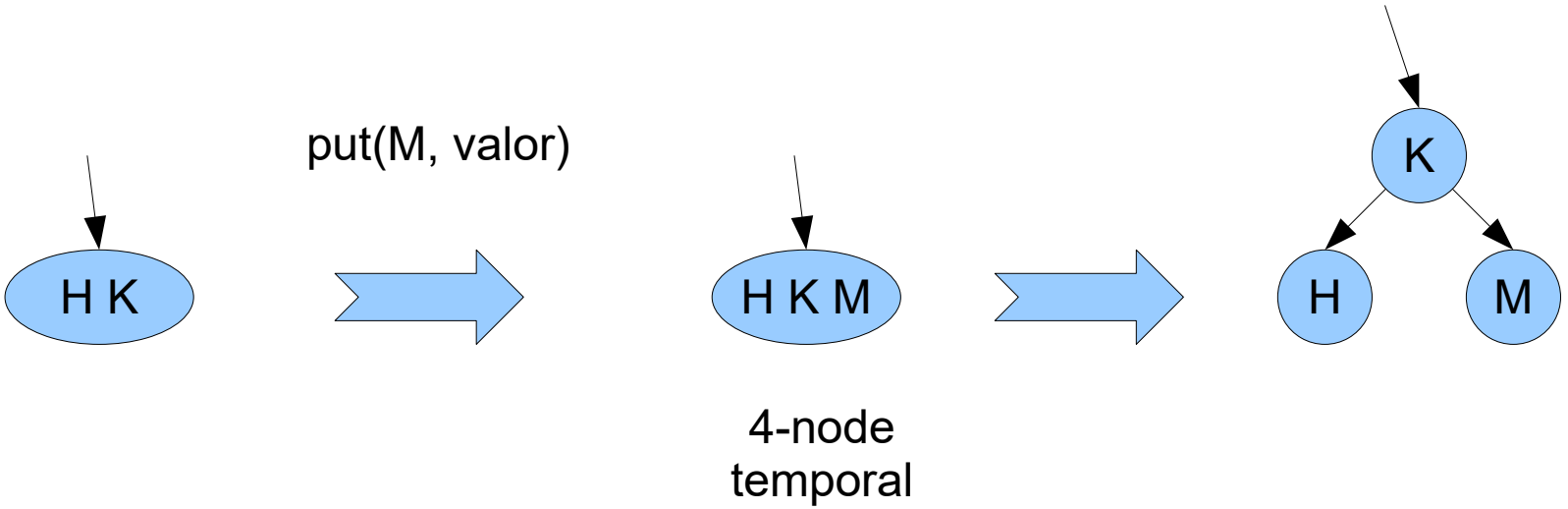
Ejemplo:
Adición de llave a un 2-node



Insert: Caso de un 3-node sencillo

- Se tiene un solo 3-node
- Para insertar, se crea temporalmente un 4-node donde se agrega la nueva llave.
- Un 4-node se puede dividir en tres 2-nodes.
- La altura del árbol aumenta en 1 y al finalizar se mantiene balanceado el árbol.

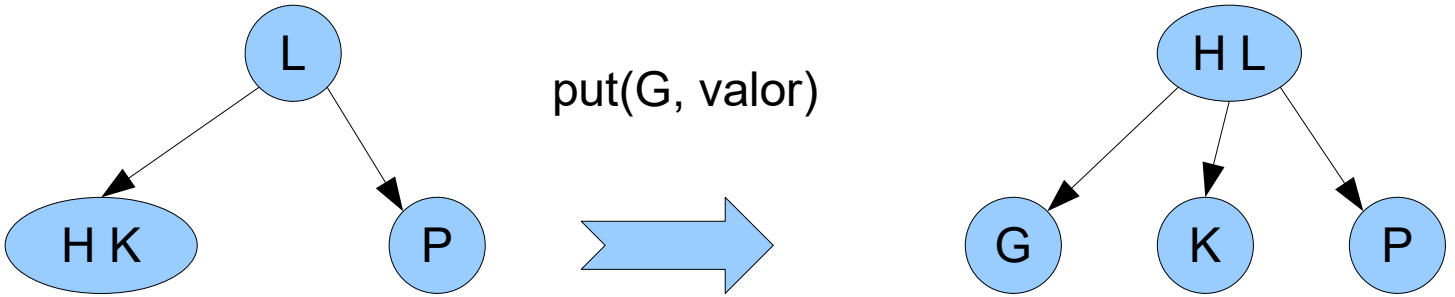
Adición a un 3-node sencillo



Insert: Caso de un 3-node cuyo padre es un 2-node

- Similar al anterior, se parte de crear un 4-node temporal en el nodo donde termina la búsqueda.
- La llave de la mitad se agrega al 2-node padre convirtiéndolo en un 3-node.
- Las dos llaves de los extremos se convierten en 2-nodes.
- Al terminar, el árbol se preserva balanceado.

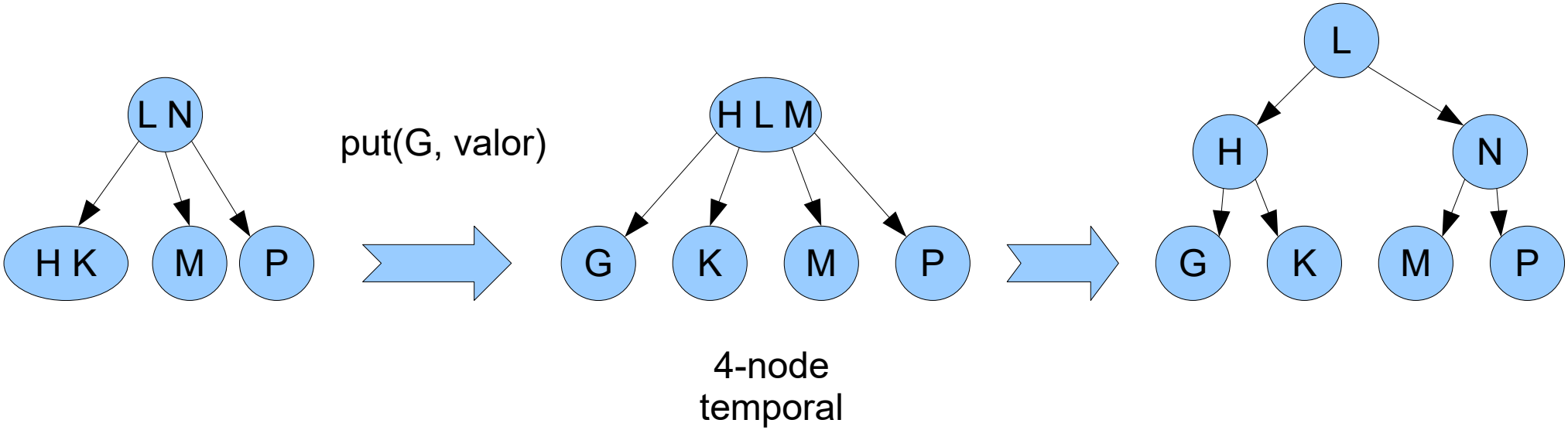
Adición a 3-node con padre 2-node



Insert: Caso de un 3-node cuyo padre es un 3-node

- Se generaliza el caso anterior:
- Se agrega en el último nodo creando un 4-node temporal.
- Se mueve la llave de la mitad al nodo padre, convirtiendolo en un 4-node.
- Se continua el proceso hasta llegar a un 2-node o a la raíz.
- En caso de llegar a la raíz, está se divide en tres 2-nodes, aumentando la altura del árbol en uno.
- En cualquier caso, el árbol continua balanceado.

Adición a 3-node con padre 3-node



Propiedades globales

- Si se parte de un árbol balanceado, cualquiera de los casos de inserción arroja un árbol balanceado al terminar.
- El ordenamiento de las llaves también se preserva en cualquiera de estos casos, mantiendo la propiedad de ordenamiento de llaves.

Altura del árbol 2-3

Proposición

En el *peor caso*, la altura de un árbol 2-3 se encuentra entre

- $\log_3(N)$: Si el árbol está todo compuesto de 3-nodes.
- $\lg(N)$: Si el árbol está todo compuesto de 2-nodes.

Está propiedad garantiza que las búsquedas e inserciones son siempre de orden ~log(N).

Implementación

- Aunque ofrece garantías de peor-caso logarítmicas, la implementación de los árboles 2-3 es dispendiosa por el gran número de casos que hay que manejar.
- Se prefiere utilizar una implementación aproximada, los *árboles rojo-negros*.

Comparativo bibliotecas

Texto guía RedBlackBST<Key extends Comparable<Key>, Value>	**Bibliotecas Java** java.util.TreeMap<K,V>
Value get(k)	V get(k)
void put(k,v)	void put(k,v)
void delete(k)	V remove(k)
int size() boolean isEmpty()	int size() boolean isEmpty() void clear()
Key min() Key max()	K firstKey() K lastKey()
Key floor() Key ceil()	K ceilingKey(k) K floorKey(k)
	K lowerKey(k) K higherKey(k)
int rank(k)	
Key select(i)	
int size(lo,hi)	
Iterable<Key> keys()	Set<K> keySet() Collection<V> values()
Iterable<Key> keys(lo,hi)	SortedMap<K,V> subMap(from,to)