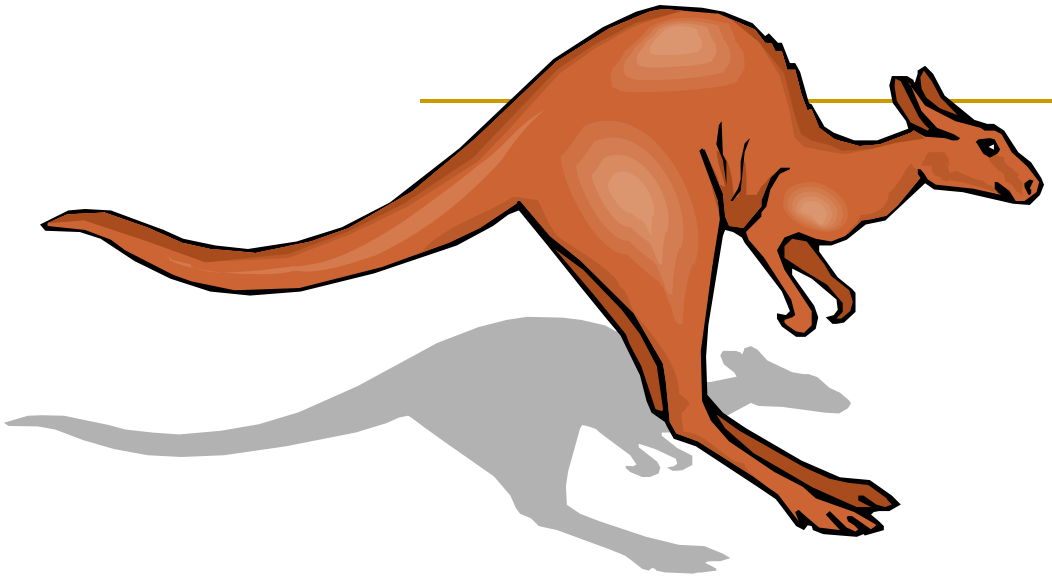
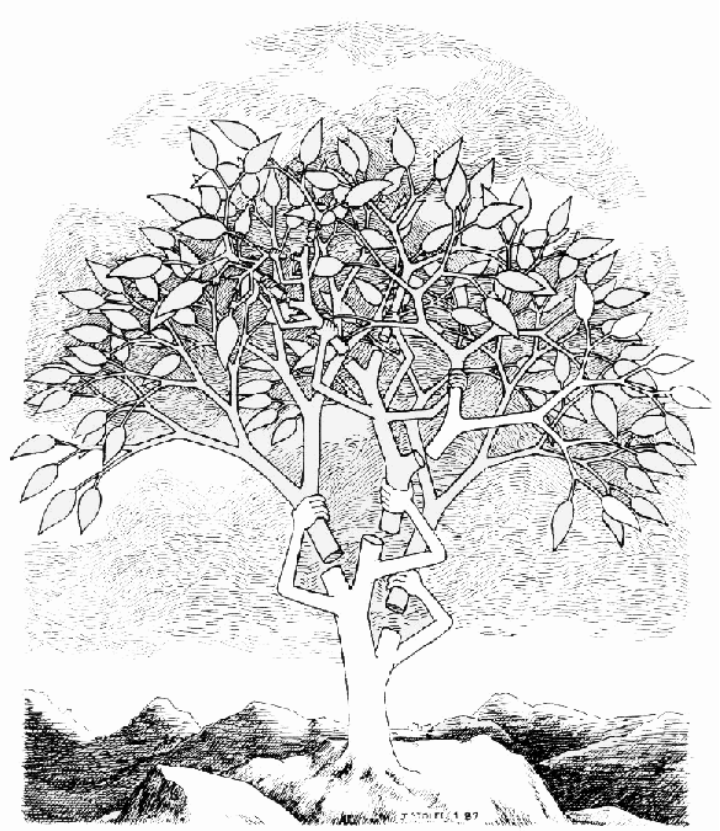
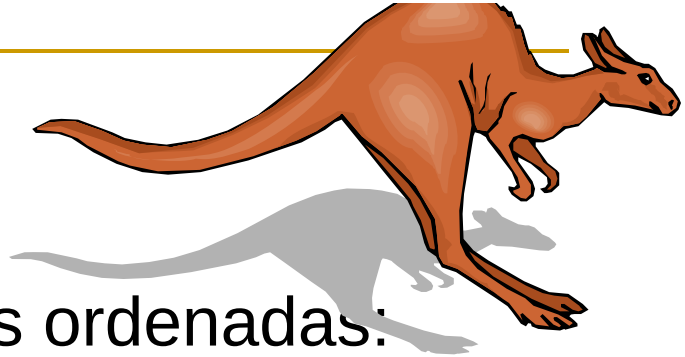
Otras implementaciones de Diccionarios



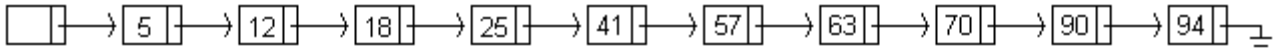


A Self-Adjusting Search Tree

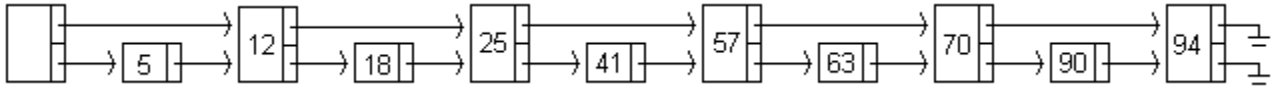
Skip lists



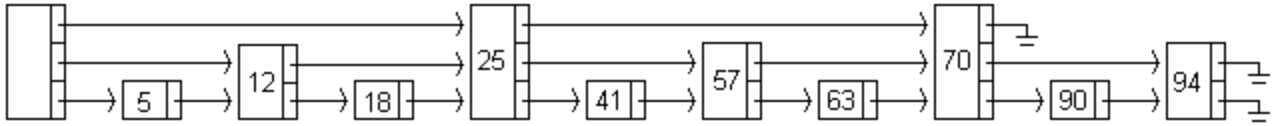
- Diccionarios con listas encadenadas ordenadas:
  - Muy simples pero ineficientes: O(n)



  - Pero….¿si tuviérarmos forma de avanzar “más rápido”?



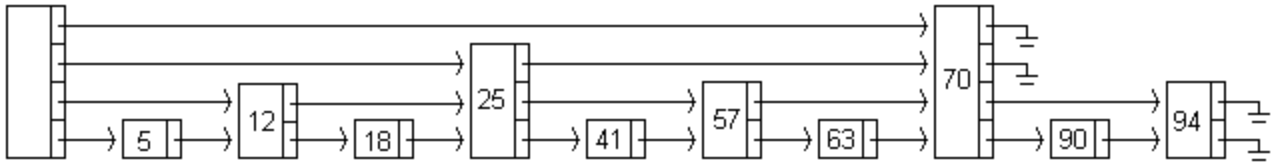
  - Así, $\left\lceil \frac{n}{2} \right\rceil + 1$ comparaciones, alcanzarían para una búsqueda….
  - Eso no parece suficiente, pero….podríamos acelerar más la cosa:



  - Por supuesto, queremos llevar esto al límite!

Skip lists (sigue)

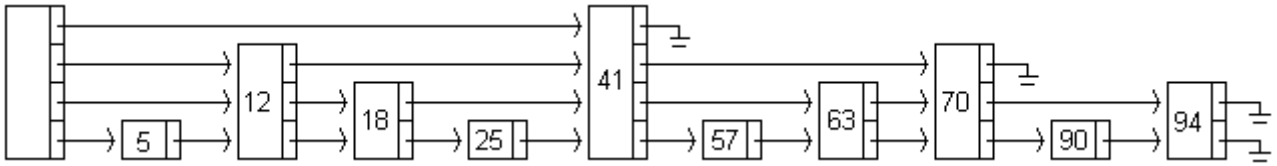
- En el límite, cada *2^i-ésimo* nodo posee una referencia al nodo 2^i posiciones más adelante en la lista.



- El número total de referencias es….
  - ¡solamente el doble!
- Pero como máximo se examinan $O(\log_2(n))$ nodos durante una búsqueda (¿a qué se parece?).
- Problema principal: rigidez en el caso dinámico (dificultad para mantenerla en el caso de borrados).

Skip lists (sigue)

- ¿Problemas de rigidez? ¡Relájese!
- En el modelo anterior, 1/2 de los nodos son de nivel 1, ¼ son de nivel 2, y $1/2^i$ son de nivel 2^i.
- Podemos “aleatorizar” o “randomizar” el requerimiento, y asignar el nivel de cada nodo probabilísticamente (por ejemplo, subiéndole el nivel mientras una moneda siga saliendo “cara”).



- Costo de búsqueda: el análisis completo puede ser complicado, pero intuitivamente es *O(log n)* en promedio (¡pero sin hacer hipótesis probabilísticas sobre el input!).
- Con una moneda “cargada” se pueden mejorar las constantes si p(cara)= *1/e*

ABB óptimos

- Si supiéramos las frecuencias de acceso a cada elemento…
- Podríamos armar un árbol en el que los elementos más accedidos estén más cerca de la raíz.
- ¿A qué nos hace acordar esto?
- En tiempo $O(n^2)$ se puede construir el árbol óptimo.
- Problemas:
  - la rigidez
  - desconocimiento a priori de las probabilidades (frecuencias) de acceso.
  - Variabilidad en el tiempo de las frecuencias de acceso: ¿puede haber algo mejor que el óptimo?

Splay Trees

- Idea: tratar de “tender” todo el tiempo al ABB óptimo (¡óptimo para ese momento!).
- Estructuras “auto-ajustantes” (self-adjusting)
- ¿Cómo? Cada vez que accedo a un elemento, lo “subo” en el árbol, llevándolo a la raíz.
- ¿Cómo? A través de rotaciones tipo AVL.
- ¿Cómo NO funciona? A través de rotaciones simples entre el elemento accedido y su padre hasta llegar a la raíz.
- ¿Cómo sí funciona? Splaying (Sleator & Tarjan, 1985)

Splay Trees

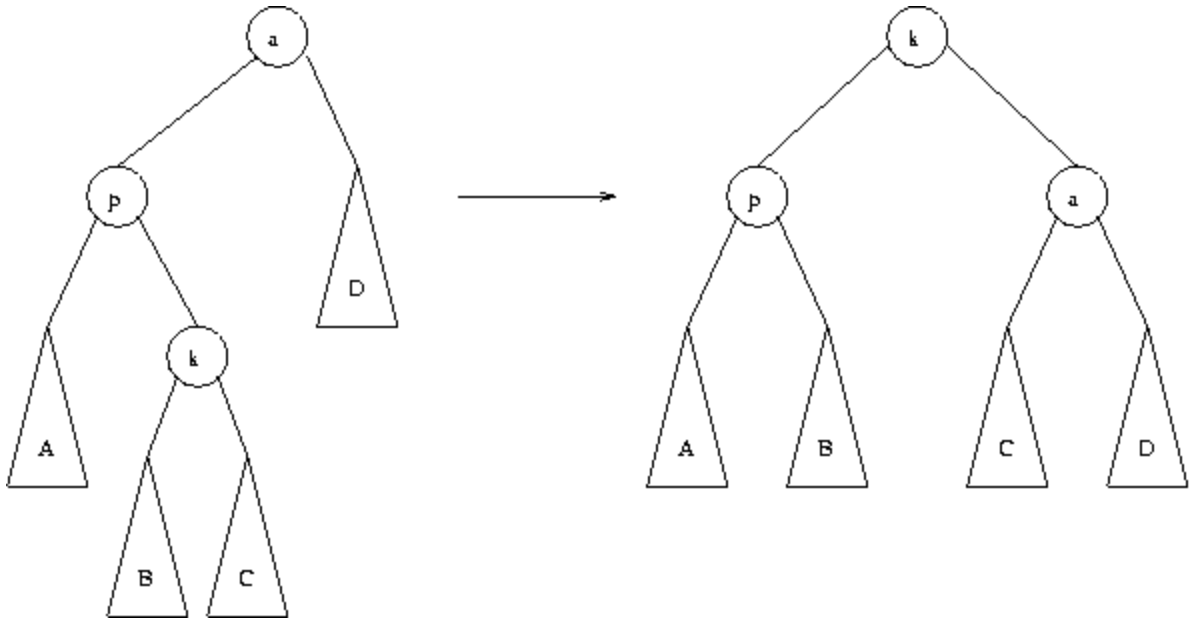
- Más simples de implementar que árboles balanceados (no hay que verificar condiciones de balanceo).
- Sin requerimientos de memoria adicionales (no hay que almacenar factores de balanceo ni nada parecido).
- Muy buena performance en secuencias de acceso no uniformes.
- Si bien no podemos garantizar *O(log n)* por operación, sí podemos garantizar *O(m log n)* para secuencias de *m* operaciones. Una operación en particular podría requerir tiempo *O(n).*
- En general, cuando una secuencia de *M* operaciones toma tiempo *O(M f(N))*, se dice que el costo *amortizado* en tiempo de cada operación es *O(f(N))*. Por lo tanto, en un splay tree los costos amortizados por operacion son de *O(log(N))*.
- Más aún, (Teorema de Optimalidad Estática): asintóticamente, los ST son tan eficientes como *cualquier* ABB fijo.
- Por último, (Conjetura de Optimalidad Dinámica): asintóticamente, los ST son tan eficientes como *cualquier ABB que se modifique* a través de rotaciones.

Splaying

- Si accedemos a la raíz del árbol, no hacemos nada.
- Si accedemos a k, y el padre de k es la raíz, hacemos una rotación simple.
- Si accedemos a k, y el padre de k no es la raíz, hay dos casos posibles (y sus especulares): rotación zig-zag, y rotación zig-zig.
- Como efecto del splaying no sólo se mueve el nodo accesado hacia la raíz, sino que *todos* los nodos del camino desde la raíz hasta el nodo accesado se mueven aproximadamente a la mitad de su profundidad anterior, a costa de que algunos *pocos* nodos bajen como máximo dos niveles en el árbol.

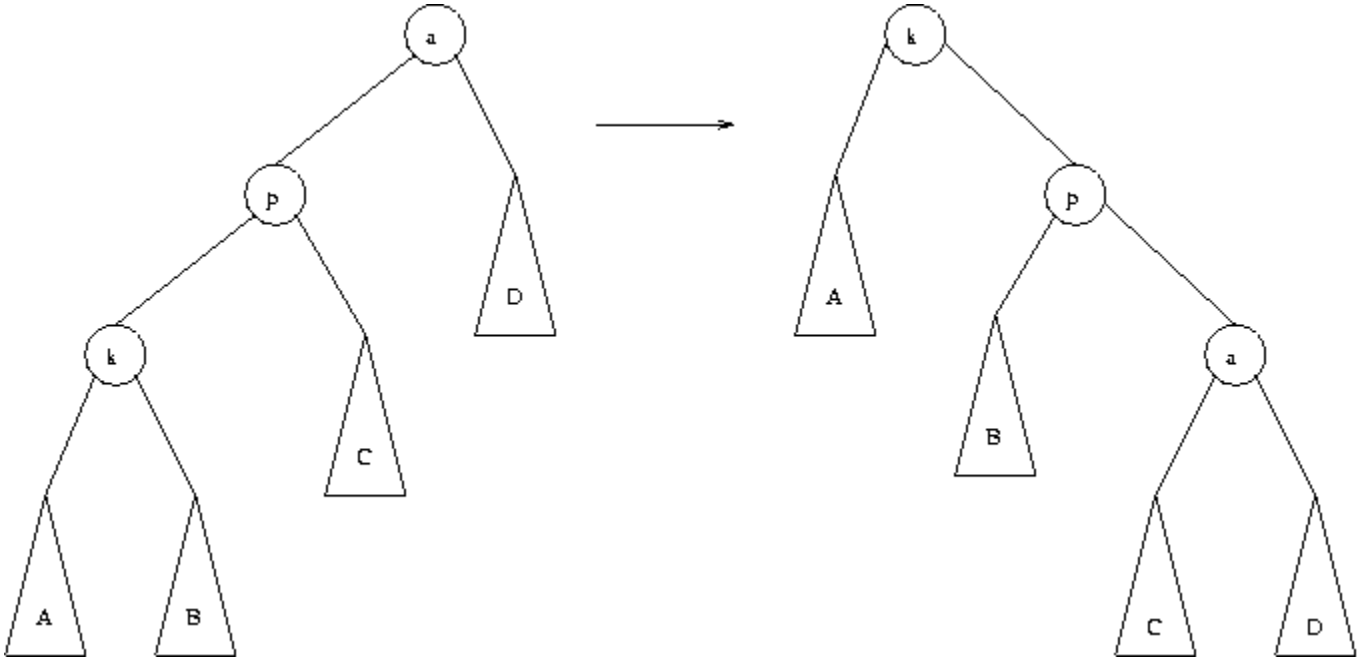
Splay Trees (sigue)

- Rotación Zig-zag



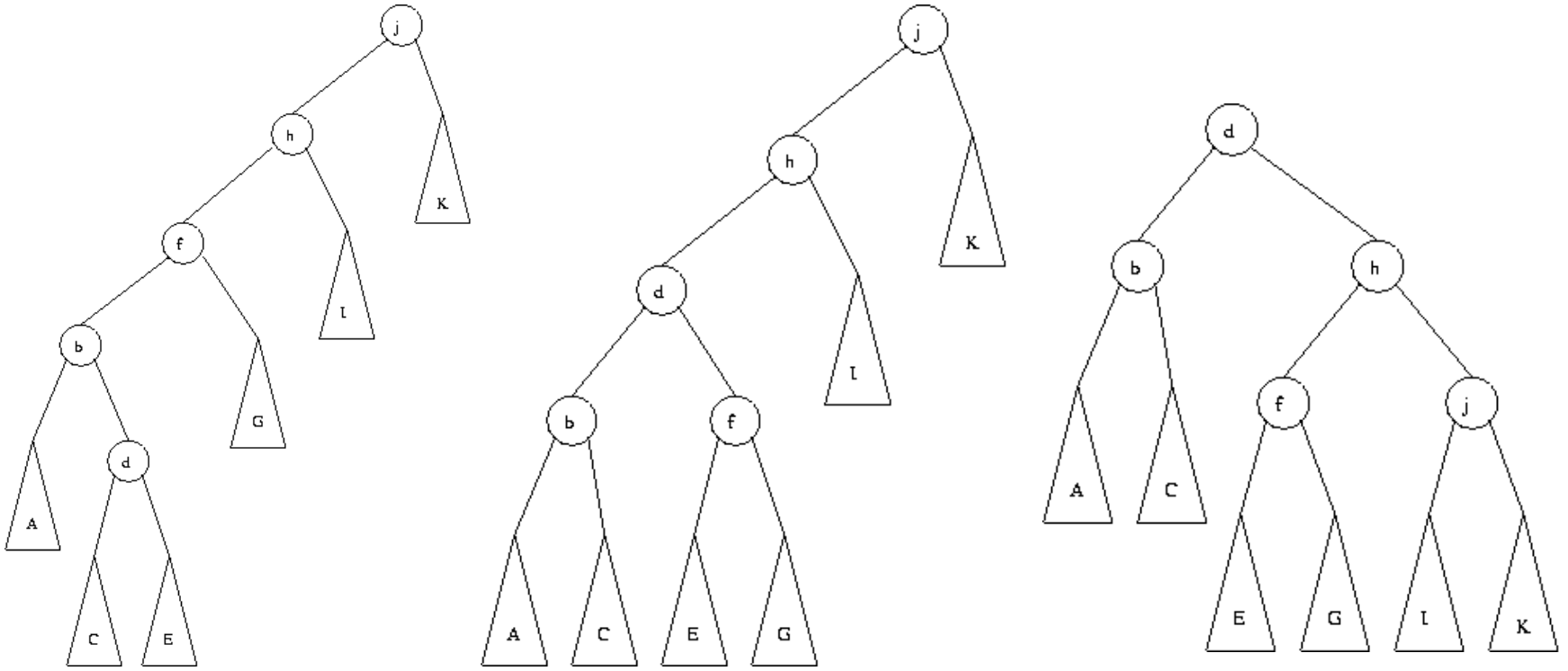
Splay Trees (sigue)

- Rotación zig-zig



Ejemplo

Acceso al nodo **d**



Inserción y Borrado

- Para insertar, hacemos “como siempre”: buscamos el lugar donde correspondería insertar y efectuamos las rotaciones resultantes de esa búsqueda. El elemento insertado queda entonces en la raíz.
- Para borrar un elemento de un splay tree se puede proceder de la siguiente forma: se realiza una búsqueda del nodo a eliminar, lo cual lo deja en la raíz del árbol. Si ésta es eliminada, se obtienen dos subárboles *Sizq* y *Sder*. Se busca el elemento mayor en *Sizq, m*, que pasa a ser su nueva raíz. Como *m* era el elemento mayor, no tenía hijo derecho, por lo que se cuelga *Sder* como subárbol derecho y *Sizq-m* como subárbol izquierdo, con lo que termina la operación de eliminación.
- Simulaciones:
  - http://www.cs.nyu.edu/algvis/java/SplayTree.html
  - http://algoviz.cs.vt.edu/Catalog (muchas cosas!)

Última: listas auto-ajustantes

- ¿Volvemos al principio? Implementación de diccionarios con listas.
- Pero….listas especiales: el elemento accedido, se mueve a la raíz. Política Move-to-front (MTF).
- Teorema: el costo total para una secuencia de *m* operaciones en una lista MTF es a lo sumo el doble que el de cualquier implementación del diccionario usando listas (Sleator & Tarjan, 1985).
- Otra forma de decirlo: MTF es 2-competitivo. Esto es un ejemplo de Análisis de Competitividad: medir algoritmos comparando su costo amortizado en el peor caso con el del (posiblemente imposible de implementar) algoritmo óptimo.
- El análisis de competitividad se usa especialmente para problemas on-line, que son problemas en los que hay que tratar de optimizar algo…¡sin conocer completamente los datos de entrada!