Análisis amortizado

Algoritmos y Estructura de Datos

Análisis Amortizado

Análisis de peor caso: Determinar el tiempo de ejecución en el peor de los casos de una operación de estructura de datos en función del tamaño de entrada n.

- puede ser demasiado pesimista si la única forma de encontrar una operación costosa es cuando hubo muchas operaciones baratas previas

Análisis amortizado: Determinar el tiempo de ejecución en el peor de los casos de una secuencia de n operaciones de estructura de datos.

- Ej. A partir de una pila vacía implementada con un array dinámico, cualquier secuencia de n operaciones push y pop tarda O(n) tiempo en el peor de los casos.

Aplicaciones de análisis amortizado

- Splay trees.
- Dynamic table (ej: array dinámicos, tablas de hash)
- Fibonacci heaps.
- Garbage collection.
- Move-to-front list updating.
- …

Multi pop stack

Operaciones de soporte en un conjunto de elementos:

- PUSH(S, x): añade el elemento x a la pila S.
- POP(S): elimina y devuelve el elemento añadido más recientemente.
- MULTI-POP(S, k): elimina los elementos k añadidos más recientemente.

Teorema: A partir de una pila vacía, cualquier secuencia entremezclada de n operaciones PUSH, POP y MULTI-POP cuesta $O(n^2)$.

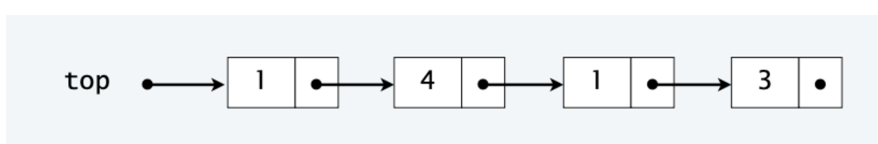
Límite superior muy pesimista

Prueba:

Usando una lista enlazada:

POP y PUSH tardan $O(1)$ tiempo cada uno.

MULTI-POP tarda $O(n)$ tiempo.



Multi pop stack

Operaciones de soporte en un conjunto de elementos:

- PUSH(S, x): añade el elemento x a la pila S.
- POP(S): elimina y devuelve el elemento añadido más recientemente.
- MULTI-POP(S, k): elimina los elementos k añadidos más recientemente.

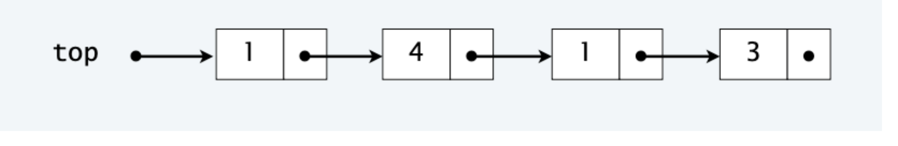
Teorema: A partir de una pila vacía, cualquier secuencia entremezclada de n operaciones PUSH, POP y MULTI-POP cuesta O(n).

Prueba:

Un elemento se extrae como máximo una vez por cada vez que se inserta.

Hay ≤ n operaciones PUSH.

Por lo tanto, hay ≤ n operaciones POP (incluidos los realizados dentro de MULTI-POP).



Pila multipop: método del banquero

Créditos: 1 crédito paga por un PUSH o un POP.

Invariante: Cada elemento de la pila tiene 1 crédito.

Contabilidad.

- PUSH(S, x): carga 2 créditos.
- 1 crédito para pagar por el push de x ahora
- 1 crédito para pagar para pagar por el futuro pop más adelante
- POP(S): carga 0 créditos.
- MULTIPOP(S, k): carga 0 créditos.

Teorema:. A partir de una pila vacía, cualquier secuencia entremezclada de n operaciones PUSH, POP y MULTI-POP tarda O(n) tiempo.

Prueba:

- Invariante ⇒ número de créditos en la estructura de datos ≥ 0.
- Costo amortizado por operación ≤ 2.
- Costo real total de n operaciones ≤ suma de los costos amortizados ≤ 2n.

Pila multipop: método potencial

Función de potencial: Sea $\Phi(D)$ = número de elementos actualmente en la pila.

$\Phi(D_0) = 0$.

$\Phi(D_i) \geq 0$ por cada D_i.

Teorema: A partir de una pila vacía, cualquier secuencia entremezclada de n operaciones PUSH, POP y MULTI-POP tarda $O(n)$.

Prueba: [Caso 1: push]

Supongamos que la i-ésima operación es un PUSH.

El costo real $c_i = 1$.

El costo amortizado $\hat{c}_i = c_i + \Phi(Di) - \Phi(Di{-}1) = 1 + 1 = 2$.

Pila multipop: método potencial

Teorema: A partir de una pila vacía, cualquier secuencia entremezclada de n operaciones PUSH, POP y MULTI-POP tarda O(n).

[Caso 2: pop]

- Supongamos que la i-ésima operación es un POP.
- El costo real $c_i = 1$.
- El costo amortizado $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_i-1) = 1 - 1 = 0$.

[Caso 3: multi-pop]

- Supongamos que la i-ésima operación es un MULTI-POP de k objetos.
- El costo real $c_i = k$.
- El costo amortizado $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_i-1) = k - k = 0$.

Pila multipop: método potencial

Teorema: A partir de una pila vacía, cualquier secuencia entremezclada de n operaciones PUSH, POP y MULTI-POP tarda (n).

2 para push; 0 para pop y multi-pop

[juntando todo]

- Coste amortizado $\hat{c}_i \leq 2$.
- Suma de los costos amortizados $\hat{c}_i$ de las n operaciones ≤ 2 n.
- Costo total $\leq$ suma del costo amortizado ≤ 2 n.

Tabla dinámica

Almacenar elementos en una tabla (por ejemplo, para una tabla hash, un montón binario).

Dos operaciones: INSERTAR y ELIMINAR.

Demasiados elementos insertados ⇒ expandir la tabla.

Demasiados elementos eliminados ⇒ tabla de contratos.

Requisito: si la tabla contiene m elementos, entonces espacio = $\Theta(m)$.

Teorema. A partir de una tabla dinámica vacía, cualquier secuencia entremezclada de n operaciones INSERT y DELETE tarda $O(n^2)$ tiempo.

Prueba. Cada INSERT o DELETE tarda $O(n)$ tiempo. ▪

Límite superior muy pesimista

Tabla dinámica: solo insertar

- Al insertar en una tabla vacía, asigne una tabla de capacidad 1.
- Al insertar en una tabla completa, asigne una nueva tabla del **doble** de la capacidad y copie todos los elementos.
- Inserte el elemento en la tabla.

insert	old capacity	new capacity	insert cost	copy cost
1	0	1	1	–
2	1	2	1	1
3	2	4	1	2
4	4	4	1	–
5	4	8	1	4
6	8	8	1	–
7	8	8	1	–
8	8	8	1	–
9	8	16	1	8
⋮	⋮	⋮	⋮	⋮

Tabla dinámica: solo inserción (método agregado)

Teorema. [a través del método agregado] A partir de una tabla dinámica vacía, cualquier secuencia de n operaciones INSERT tarda O(n) tiempo.

Prueba: Sea c_i el costo de la i-ésima inserción.

c_i = { i: si i-1 es potencia de 2;

1: sino es

A partir de una tabla vacía, el costo de una secuencia de n operaciones INSERT es:

$$\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j < n + 2n = 3n$$

Tabla dinámica: solo inserción (método potencial)

Teorema. A partir de una tabla dinámica vacía, cualquier secuencia de n operaciones INSERT tarda O(n) tiempo.

Prueba : Sea $\Phi(D_i)$ = 2 tamaño(D_i) – capacidad(D_i).

- $\Phi(D_i) = 0$.
- $\Phi(D_i) \geq 0$ por cada D_i.

inmediatamente después de la duplicación capacidad (D_i) = 2 tamaño (D_i)

1	2	3	4	5	6		

Tamaño = 6, capacidad = 8, Φ = 4

Costo amortizado ĉ = c + (Φ(D') – Φ(D))
El costo de la operación mas el "cambio de potencial", (la diferencia esntre el estado nuevo y el Viejo)

Tabla dinámica: solo inserción (método potencial)

Teorema: A partir de una tabla dinámica vacía, cualquier secuencia de n operaciones INSERT tarda O(n) tiempo.

Sea $\Phi(D_i) = 2\ \text{tamaño}(D_i) - \text{capacidad}(D_i)$.

Caso 0. [primera inserción]

- Costo real $c_1 = 1$.
- $\Phi(D_1) - \Phi(D_0) = (2\ \text{tamaño}(D_1) - \text{capacidad}(D_1)) - (2\ \text{tamaño}(D_0) - \text{capacidad}(D_0))$
  $= 1$.
- Costo amortizado $\hat{c}_1 = c_1 + (\Phi(D_1) - \Phi(D_0))$
  $=1+ 1 = 2$.

Tabla dinámica: solo inserción (método potencial)

Teorema: A partir de una tabla dinámica vacía, cualquier secuencia de n operaciones INSERT tarda O(n) tiempo.

Sea $\Phi(D_i) = 2$ tamaño(D_i) – capacidad(D_i).

Caso 1. [sin expansión de la tabla]

- capacidad(D_i) = capacidad(D_{i-1}).

- Costo real $c_i = 1$.
- $\Phi(D_i) - \Phi(D_{i-1})$ = (2 tamaño(D_i) – capacidad(D_i)) – (2 tamaño(D_{i-1}) – capacidad(D_{i-1}))
  = 2.
- Costo amortizado $\hat{c}_i = c_i + (\Phi(D_i) - \Phi(Di–1))$
  =1+ 2
  = 3.

Tabla dinámica: solo inserción (método potencial)

Teorema: [método potencial] A partir de una tabla dinámica vacía, cualquier secuencia de n operaciones INSERT tarda O(n) tiempo.

Caso 2. [expansión de la tabla]

- capacidad(D_i) = 2 capacidad(D_{i-1}).

- Costo real c_i = 1 + capacidad (D_{i-1}).
- Φ(Di) – Φ(Di–1) = (2 tamaño(D_i) – capacidad(D_i)) – (2 tamaño(D_{i-1}) – capacidad(D_{i-1}))
  = 2 – capacidad(D_i) + capacidad(D_{i-1})
  = 2 – capacidad(D_{i-1}).
- Costo amortizado $\hat{c}_i = c_i + (\Phi(D_i) - \Phi(D_{i-1}))$
  = 1 + capacidad(D_{i-1}) + (2 – capacidad(D_{i-1}))
  = 3.

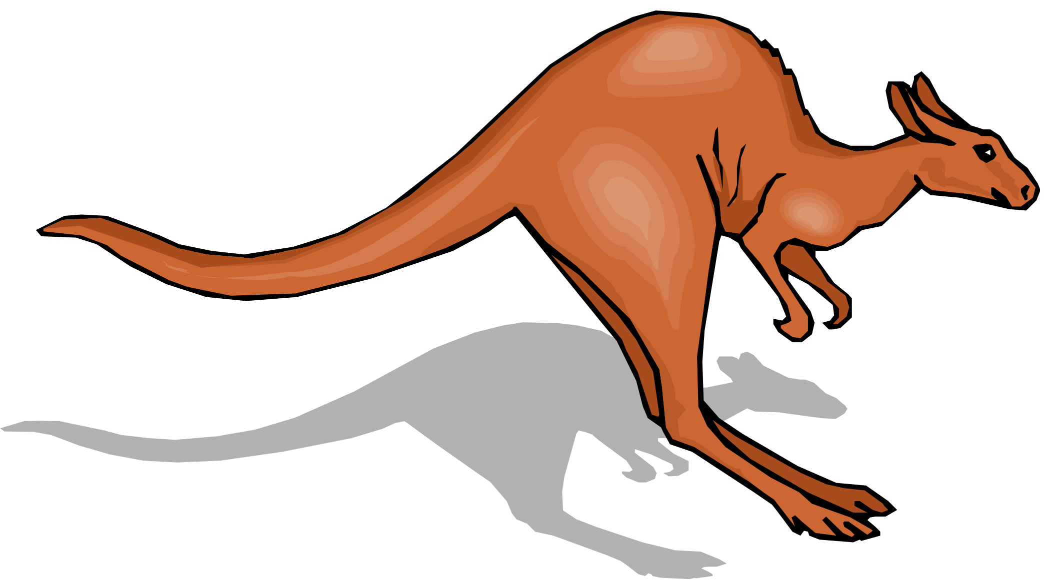
Tabla dinámica: solo inserción (método potencial)

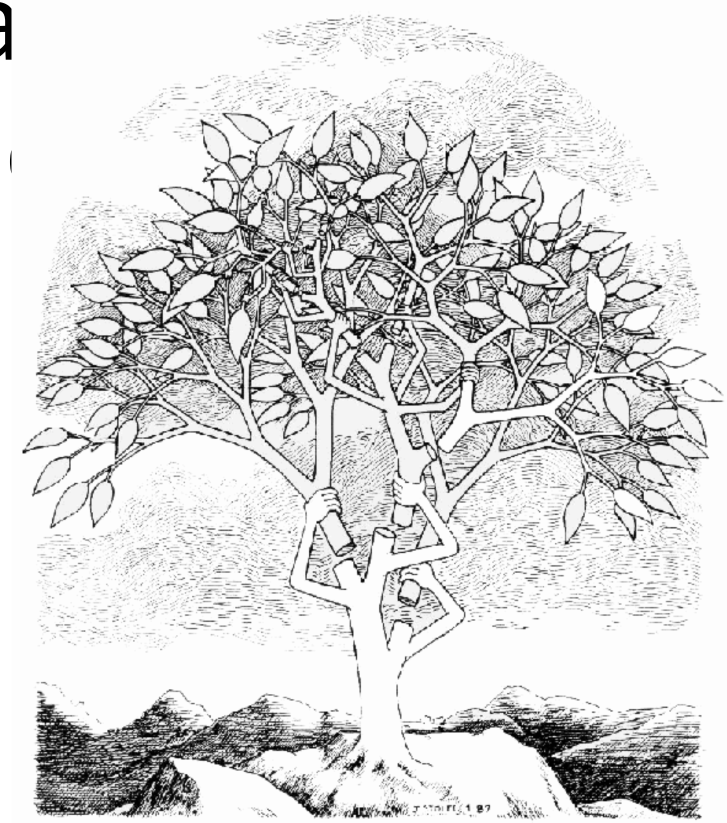
Teorema: A partir de una tabla dinámica vacía, cualquier secuencia de n operaciones INSERT tarda O(n) tiempo.

[poniendo todo junto]

- Costo amortizado por operación $\hat{c}_i \leq 3$.
- Costo real total de n operaciones ≤ suma del costo amortizado ≤ 3 n. ▪
-

Otras implementa
Diccionari



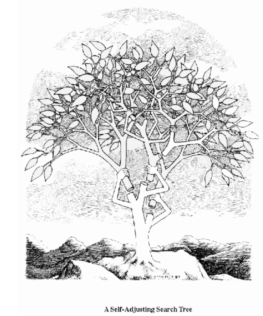


A Self-Adjusting Search Tree

ABB óptimos

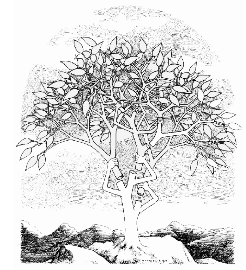
- Si supiéramos las frecuencias de acceso a cada elemento…
- Podríamos armar un árbol en el que los elementos más accedidos estén más cerca de la raíz.
- ¿A qué nos hace acordar esto?
- En tiempo $O(n^2)$ se puede construir el árbol óptimo.
- Problemas:
  - la rigidez
  - desconocimiento a priori de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



A Self-Adjusting Search Tree

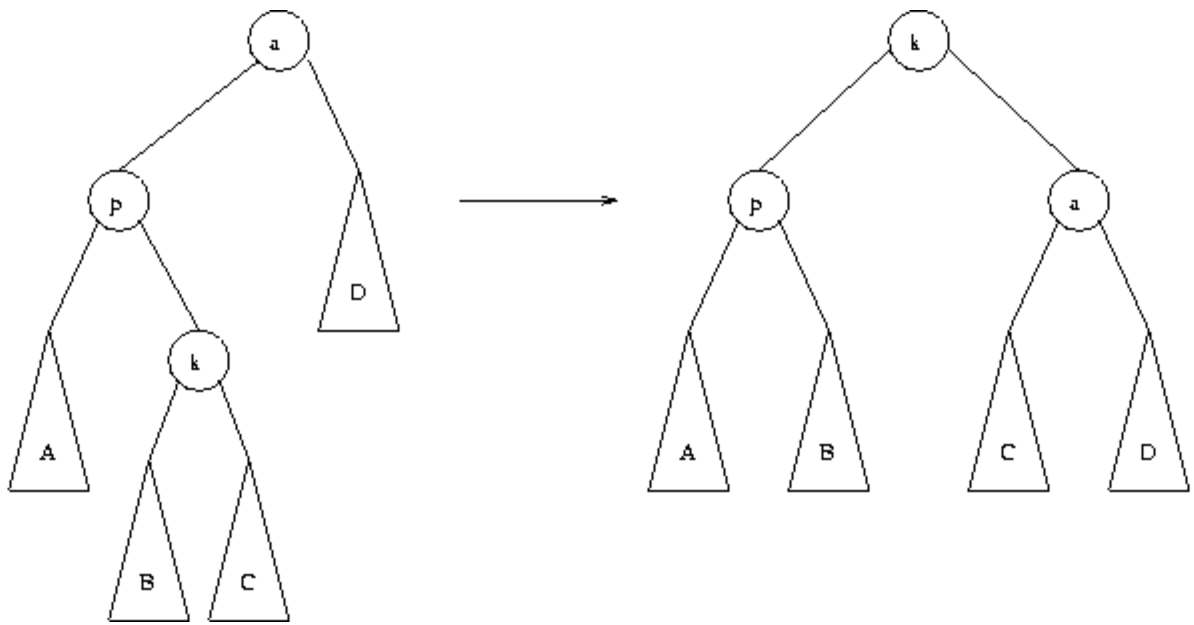
- Más simples de implementar que árboles balanceados (no hay que verificar condiciones de balanceo).
- Sin requerimientos de memoria adicionales (no hay que almacenar factores de balanceo ni nada parecido).
- Muy buena performance en secuencias de acceso no uniformes.
- Si bien no podemos garantizar *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.

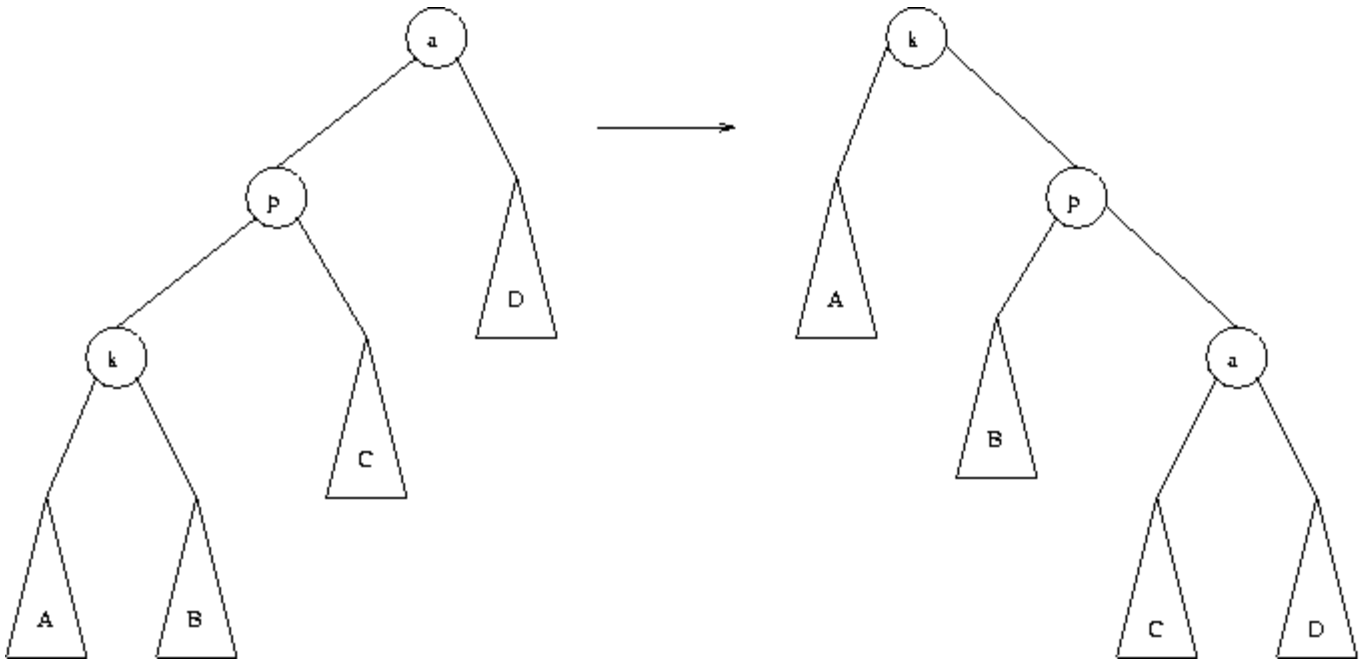
Splay Trees (sigue)

- Rotación Zig-zag



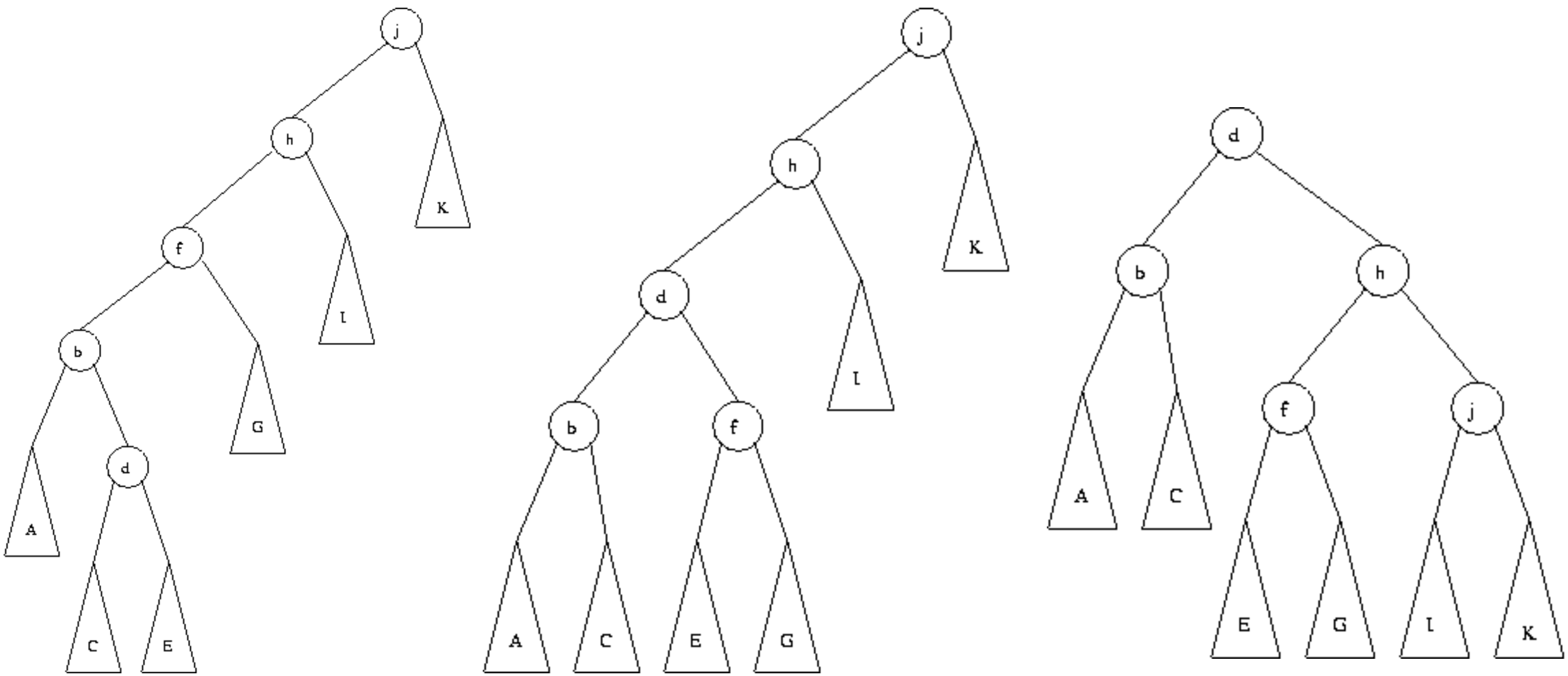
Splay Trees (sigue)

- 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!