Análisis Amortizado : analizar costo de n operaciones

El costo amortizado de una operación es el $\frac{\text{costo total de la secuencia}}{n}$

Esto se usa cuando el costo de n operaciones $< n \cdot O(T(n))$

con $T(n)$ el costo del peor caso.

> Ciertas operaciones son muy caras, pero otras muy poco

Técnicas:
- análisis global : observar los costos de toda la secuencia de forma global.
- contabilidad de costos : se reparte el costo real en cada paso, se debe encontrar una forma para repartir los costos.
- función potencial: c_i : costos reales, $\hat{c}_i$: secuencia costos amortizados
  ϕ_i : función potencial → representa el ahorro
  $\Delta\phi_i = \phi_i - \phi_{i-1}$, $\phi_n \geq \phi_0$
  $\hat{c}_i = c_i + \Delta\phi_i$

// Ejemplos : pila con multipop, incrementar número binario, realocar un arreglo

Colas de prioridad:
- heap
- colas binomiales
- colas de fibonacci

	insert	findMin	ExtractMin	Merge
heap	$O(\log(n))$	$O(1)$	$O(\log(n))$	$O(n+m)$
colas binomiales	$O(\log(n))$	$O(1)$	$O(\log(n))$	$O(\log(n+m))$
colas de fibonacci	$O(1)$	$O(1)$	$O(\log(n))$ *	$O(1)$

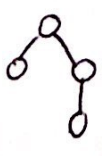
(Heapify y Init cuestan $O(n)$ y $O(1)$)

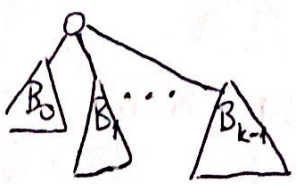
1. Colas Binomiales

↳ Árboles Binomiales

$B_0 =$ o

$B_1 =$ o—o

$B_2 =$ 

$B_k =$ 

B_k tiene:
- 2^k nodos
- es raíz de k hijos
- su altura es $k+1$
- tiene $\binom{k}{i}$ nodos a profundidad i.

Un <u>bosque binomial</u> es un conjunto de árboles binomiales sin ningún árbol repetido.

Con "n" la cantidad de nodos, ¿qué arboles binomiales necesitamos para un bosque binomial de tamaño n?

∃! bosque binomial ⟹ tomemos los B_k como 1s en la descomposición binaria.

> Esto viene de la cantidad de nodos de un árbol B_k es 2^k

// Ej: $n = 4 = \underset{B_2\,B_1\,B_0}{100} \rightarrow \{$  $\}$

$n = 5 = \underset{B_2\,B_1\,B_0}{101} \rightarrow \{$  $, 0\}$

★ Luego, un bosque binomial de n nodos tiene a lo más $\lceil \log_2 n \rceil$ árboles binomiales.

En una <u>cola binomial</u> almacenaremos los elementos en los nodos del bosque binomial.

↳ Suma de colas binomiales:

Es similar a la suma de binarios:

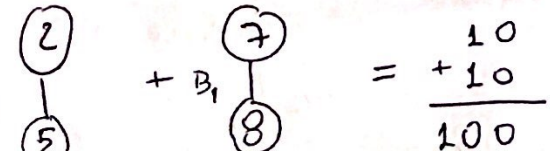
$0 + 0 = 0$

$0 + 1 = 1 + 0 = 1$

$1 + 1 = 10$

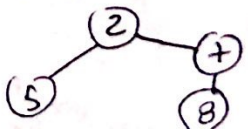
// Ej:

$$\begin{array}{cc} \overset{1}{1}\ \overset{1}{0}\ 1\ 1 & \{B_3, B_1, B_0\} \\ +\ 0\ 0\ 1\ 1 & +\ \{B_1, B_0\} \\ \hline 1\ 1\ 1\ 0 & \{B_3, B_2, B_1\} \end{array}$$

// Ej: 

$$= \begin{array}{r} 10 \\ +10 \\ \hline 100 \end{array}$$

Uno los dos árboles B_k en un árbol B_{k+1}:



Al sumar las colas binomiales C_x y C_y, requerimos $O(\log|X \cup Y|)$

2. Colas de Fibonacci

Es un bosque de árboles binomiales, en donde pueden existir árboles B_k repetidos. Se convierte en un bosque binomial cuando se extrae el mínimo.

En la operación Extract Min, se extrae la raíz B_k que contiene al mínimo y agregamos sus hijos al bosque de árboles binomiales.

El nuevo bosque será: $C_s' = (C_s \setminus B_k) \cup \{B_0, B_1, \ldots, B_{k-1}\}$

Y luego, se convierte a un bosque binomial:

1- Creamos arreglo A de $\lceil \log_2 n \rceil$ punteros, donde $^*A[k] = B_k$

2- Recorremos la lista C_s', viendo c/árbol B_k:

si $A[k] == nulo$

$A[k] \leftarrow B_k$

si no

$B_{k+1} = {}^*A[k] \cup B_k$

$A[k] = nulo$

unimos B_{k+1} a C_s'.

3- El árbol binomial se encuentra en A. y calculamos el mínimo sobre las $\lceil \log_2 n \rceil$ raíces.

3. Union - Find

Construimos esta interfaz para el algoritmo de Kruskal que encuentra el árbol cobertor mínimo.

Con esta interfaz queda más claro cuando una arista forma o no un ciclo.

Dos nodos están en la misma clase de equivalencia si son nodos del mismo árbol. Podemos entonces definir a c/elemento con un representante de su clase.

→ Find (v): entrega el representante de la clase de equivalencia a la que pertenece v.

→ Union (u, v): une las clases de equivalencia representadas por u y v.

El algoritmo hace:

1- Ordenar las |E| aristas según su peso (guardadas en lista C)

2- T ← ∅ (el árbol de Kruskal que construiremos)

3- Una clase de equivalencia por nodo.

4- Recorremos las aristas en C mientras |T| < |V| - 1

e ← nueva arista (u, v)

si find (u) ≠ find (v):

T ← T + Union (find(u), find(v))

Así, realizamos O(e) Find y O(n) Union.

Pero ¿Cuánto cuesta Find y Union?

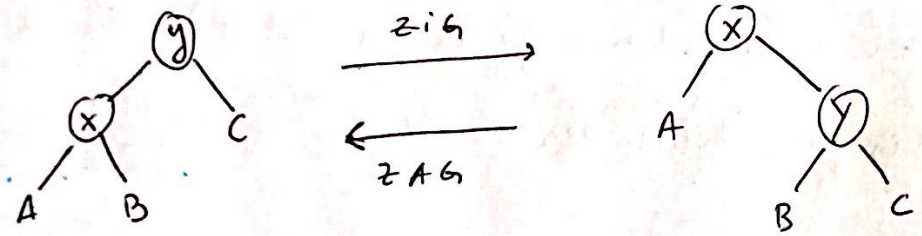
Vamos a guardar los |V| elementos en nodos y las clases de equivalencias serán árboles formados por los nodos de su clase de equivalencia. Cada nodo apunta a su padre y la raíz será el representante de su clase. De esta forma:

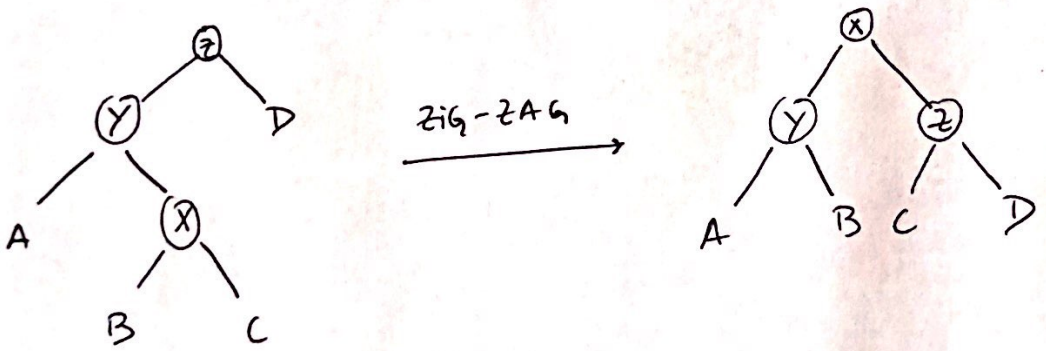
Find: debe recorrer los ancestros sucesivos de v hasta llegar a la raíz → O(log(n))

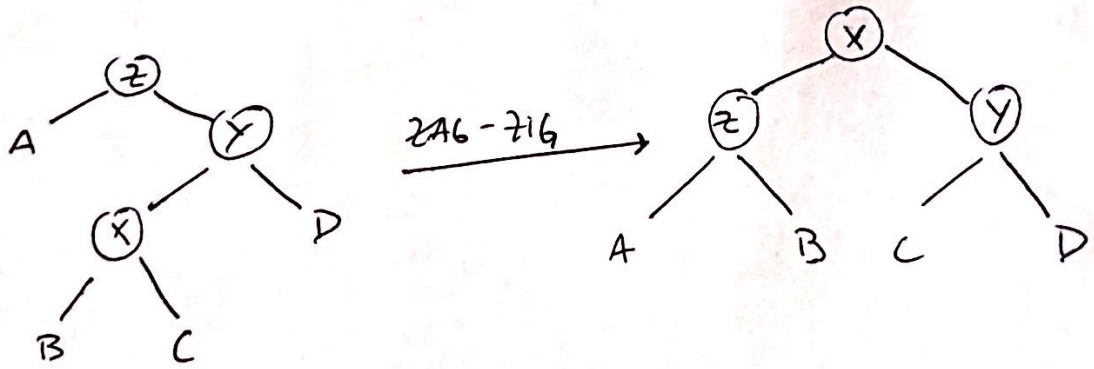
Union(u,v): cuelga u como un hijo de v (el árbol con menos nodos cuelga del que tiene más nodos) → O(1)

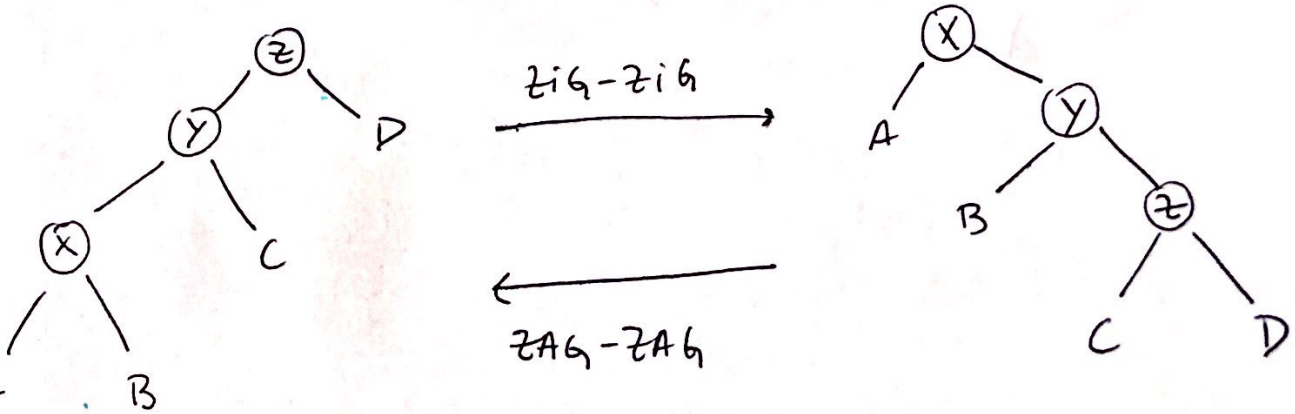
4. Splay Trees

Es un árbol binario de búsqueda que se balancea "solo". Cuando se accede a un nodo, este se lleva a la raíz. La operación de llevarlo a la raíz se llama splay(x) y está formada por una secuencia de las siguientes rotaciones:









La idea es que los elementos que se accedieron recientemente, se accederán + rápidos posteriormente.

Las operaciones buscar, insertar y borrar se detallan en el apunte.

Estas operaciones tienen un costo amortizado de $O(\log(n))$.

Y cuando se hace una secuencia de accesos a nodos con distinta probabilidad, el costo amortizado es $O(H)$, donde H es la entropía de esas probabilidades: $H = \sum_i p_i \log_2\left(\frac{1}{p_i}\right)$