Algoritmos y Complejidad

Análisis Amortizado de Estructuras de Datos

Pablo R. Fillottrani

Depto. Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

primer semestre 2024

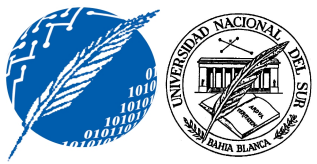
Análisis Amortizado de Estructuras de Datos

Usos y principios

- el análisis amortizado estudia el tiempo requerido para ejecutar una secuencia de operaciones sobre una estructura de datos
- si usamos el análisis normal en el peor caso, ejecutar N operaciones sobre una estructura de datos de n elementos lleva tiempo en $O(Nf(n))$, donde $f(n)$ es el tiempo en el peor caso de la operación
- en muchos casos esa cota no es ajustada debido a que el peor caso puede NO ocurrir las N veces, o incluso ninguna de esas veces
- entonces se introducen las técnicas de análisis amortizado para tratar de obtener una cota menor para la serie de operaciones


- el análisis amortizado se diferencia del análisis en el caso promedio en que no involucra probabilidades, y en que garantiza el tiempo en el peor caso de las N operaciones
- el análisis probabilístico produce un tiempo esperado que una determinada ejecución puede sobrepasar o no
- el análisis amortizado produce una cota en el tiempo de ejecución de la serie de operaciones (es decir, sigue siendo un análisis del peor de los casos)


- los resultados del análisis amortizado sirven para optimizar el diseño de la estructuras de datos, produciendo entonces estructuras de datos avanzadas
- si las operaciones sobre una estructura de datos se llaman en una secuencia conocida entonces conviene minimizar el tiempo de sus operaciones mediante el análisis amortizado
- la E.D. conjuntos disjuntos al utilizar compresión de caminos está aplicando esta metodología, obteniendo una cota $O(N \log^* n)$ para las N operaciones.



Tipos de análisis amortizado

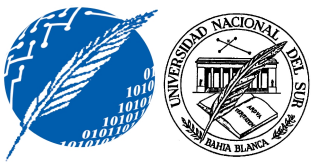
- existen diversas técnicas para realizar el análisis amortizado:

Técnicas de Análisis Amortizado
- método del agregado
- método contable
- método del potencial

- en general todas las técnicas son aplicables a todos los casos y dan resultados equivalentes
- sólo veremos en la materia el método del potencial


Método del potencial

- este método representa los ahorros hechos en operaciones previas mediante incrementos en la "energía potencial", que puede ser gastada en operaciones costosas siguientes
- la energía potencial representa las tareas realizadas en llamadas anteriores de una operación, que pueden ser usadas en próximas invocaciones a la operación
- se comienza con una estructura de datos inicial D_0, y para cada $i = 1, 2, \ldots, N$ sea c_i el costo real de la i-ésima operación y D_i la estructura de datos resultante después de aplicar esa operación sobre D_{i-1}



- una función potencial Φ mapea cada estado D_i de la estructura de datos a un número real, llamado el potencial de D_i
- el costo amortizado de la i-ésima operación se define como

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

- entonces, la sumatoria de los costos amortizados de la secuencia de operaciones es:

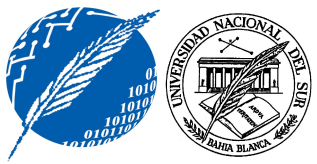
$$\sum_{i=1}^{N} \hat{c}_i = \sum_{i=1}^{N} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{N} c_i \; + \Phi(D_N) - \Phi(D_0)$$

- si se puede definir Φ tal que $\Phi(D_N) \geq \Phi(D_0)$ entonces este valor es una cota superior a la sumatoria de costos reales
- si la función está bien elegida y la E.D. está diseñada para ello, esta cota superior es más ajustada que N veces el peor caso


Pila extendida

- consideremos una E.D. Pila con la operación adicional `DesapilarMúltiple`(k) que elimina los k elementos que están más ariba en la pila
- las operaciones `Apilar`, `Desapilar` y `Tope` son de $\Theta(1)$ en el peor caso individual, pero la operación `DesapilarMúltiple`(k) es de $\Theta(\texttt{min}(k, n))$, siendo n los elementos en la pila
- esto da un tiempo de $O(nN)$ para cualquier secuencia de N operaciones
- consideraremos de costo real 1 a las tres primeras operaciones.


- para hacer el análisis amortizado, tomemos como función potencial $\Phi(D_i)$ la cantidad de elementos en la pila D_i
- dado que nunca es negativo, $\Phi(D_i) \geq \Phi(D_0)$ para todo i
- esto asegura que la sumatoria de costos amortizados es una cota superior a la sumatoria de costos reales



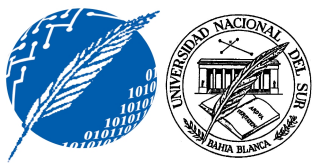
Análisis amortizado de las operaciones

- sea s la cantidad de elementos después de $i-1$ operaciones.
- para la operación `Apilar`:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + s + 1 - s = 2 \in \Theta(1)$$

- para la operación `Tope`:

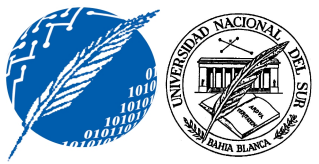
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + s - s = 1 \in \Theta(1)$$



- para la operación `Desapilar`:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s-1) - s = 0 \in \Theta(1)$$

- en todos estos casos el análisis amortizado no mejora la cota de la secuencia de operaciones obtenida con N veces el peor caso



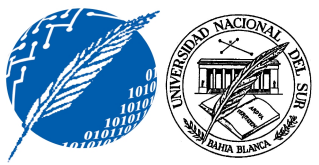
- para la operación `DesapilarMúltiple`(k), con $k \leq s$:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k + s - k - s = 0 \in \Theta(1)$$

- para la operación `DesapilarMúltiple`(k), con $k > s$:

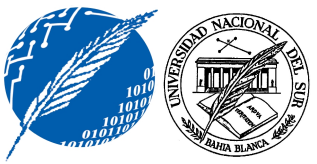
$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = s + 0 - s = 0 \in \Theta(1)$$

- este análisis amortizado permite afirmar que N operaciones sobre esta E.D. lleva tiempo en $\Theta(N)$ en el peor caso

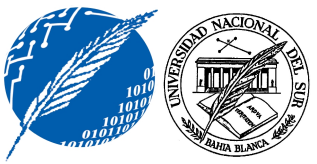


Contador binario

- se tiene un contador binario de n bits con la operación de incremento
- el costo de esta operación es la cantidad de bits cambiados
- en el peor caso, la cantidad de bits cambiados son todos los bits del contador, por lo que N incrementos es de $O(nN)$
- es claro que el peor de los casos no se da en todas las N operaciones, por lo que es válido realizar un análisis amortizado



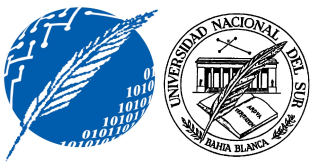
- para hacer un análisis amortizado de esta E.D. tomaremos como función potencial la cantidad de 1's en el contador
- si empezamos con el contador en cero, $\Phi(D_i) \geq \Phi(D_0)$ para todo i, por lo que podemos asegurar que la sumatoria de costos amortizados es una cota superior a la sumatoria de costos reales



- sea s la cantidad de 1's del contador antes del incremento, t de los cuales se modifican en la i-ésima operación
- el costo amortizado de la operación es:

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) = (1+t) + (s-t+1) - (s) = 2 \\ &\in \Theta(1) \end{aligned}$$

- luego, N operaciones de incremento del contador llevan en el peor caso tiempo de $\Theta(N)$
- se puede extender el análisis amortizado aún para el caso cuando el contador no empieza en 0 (ejercicio)



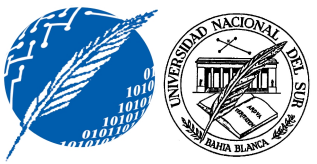
Tablas dinámicas

- en algunas aplicaciones no se conoce previamente la cantidad de objetos que pueden llegar a ser almacenados en una tabla
- es útil por lo tanto utilizar una tabla dinámica que vaya solicitando memoria a medida que la necesita
- las operaciones normales de `Inserción` y `Eliminar` toman tiempo en $\Theta(1)$
- pero el peor caso de una inserción con expansión es de orden de la cantidad de memoria asignada, lo que hace que la cota superior de una serie de operaciones no sea ajustada si la expansión es infrecuente


- para implementar esta E.D. será conveniente referirse al factor de carga $\alpha(T)$
- se define como la relación entre la cantidad de elementos y la memoria asignada:

$$\alpha(T) = \texttt{elementos/capacidad}$$

- también supondremos que la asignación de la nueva memoria es independiente de la anterior, o sea no se pueden juntar lo que se tenía antes con lo que se dispone ahora



- si T está vacía, por definición es $\alpha(T) = 1$
- supongamos en principio que se quiere implementar una tabla dinámica que sólo soporte inserciones
- es necesario decidir cuándo y cuánto expandir
- una heurística común es realizar la expansión en el momento que la tabla no permita una inserción (ie $\alpha(T) = 1$), incorporando el doble de la memoria hasta entonces disponible

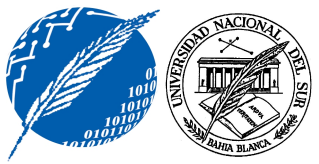
```
PROCEDURE Insertar(x)
  IF this.tamaño=0
    asignar memoria para un elemento
    this.tamaño ::= 1
  ENDIF
  IF this.tamaño=this.elementos
    asignar el doble de  memoria disponible
    trasladar todos los elementos
    this.tamaño::=2*this.tamaño
    liberar la memoria anterior
  ENDIF
  insertarElemental(x)
  this.elementos ::= this.elementos++
  RETURN
```


- el tiempo en el peor caso de una inserción en una tabla de n elementos es de $\Theta(n)$, por lo que una serie de N operaciones es de $O(nN)$
- con esta heurística se asegura que el factor de carga nunca es menor que $0,5$, por lo que el espacio usado no superará el doble de lo máximo necesitado
- la expansión de la tabla ocurre solamente cuando la cantidad de elementos es potencia de 2
- la mayoría de las inserciones son de tiempo constante


- se necesita una función potencial que valga 0 inmediatamente después de una expansión, y la cantidad de elementos insertados inmediatamente antes de la expansión
- una posibilidad es:

$$\Phi(T_i) = 2 * T_i.\text{elementos} - T_i.\text{tamaño}$$

- vale que $\Phi(T_i) \geq \Phi(T_0)$ si se comienza con la tabla vacía



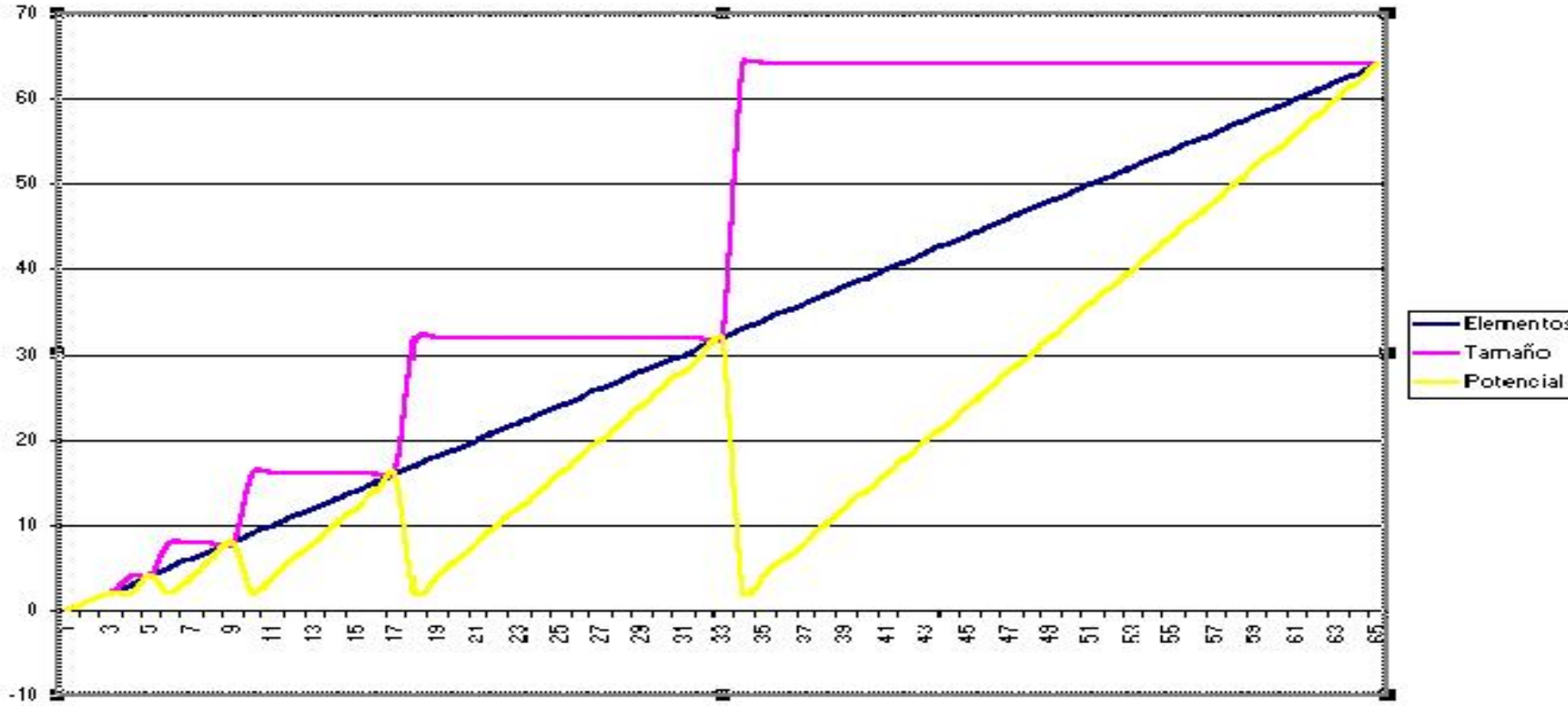
- el análisis amortizado para una inserción es:
  - sin expansión

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) = \\ &= 1 + (2 * (elem_{i-1} + 1) - tam_{i-1}) - \\ &\quad -(2 * elem_{i-1} - tam_{i-1}) = \\ &= 1 + 2 = 3 \in \Theta(1)\end{aligned}$$

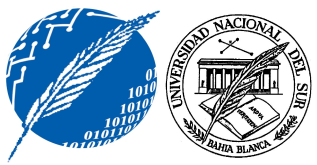
  - con expansión

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) = \\ &= 1 + elem_{i-1} + (2 * (elem_{i-1} + 1) - 2 * tam_{i-1}) - \\ &\quad -(2 * elem_{i-1} - tam_{i-1}) = \\ &= 1 + elem_{i-1} + 2 - tam_{i-1} = 3 \in \Theta(1)\end{aligned}$$


Comparación



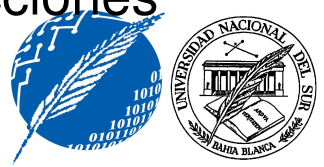
- para implementar la operación `Eliminar` es suficiente con remover el elemento de la tabla
- sin embargo, es deseable también implementar una contracción cuando el factor de carga de la table sea suficientemente pequeño
- se quiere entonces acotar por abajo el factor de carga, y acotar por arriba el costo amortizado de las operaciones
- una estrategia natural es contraer la tabla a la mitad cuándo el factor de carga es menos de $0,5$. Sin embargo, esta estrategia no es buena.



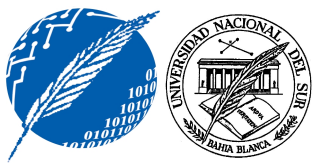
- el problema con esta estrategia surge cuando no se ejecutan suficientes inserciones después de una expansión como para pagar el gasto de una contracción
- en términos del potencial, no hay energía potencial que compense ese gasto
- por ejemplo:

$$\underbrace{\mathbf{I}, \mathbf{I}, \ldots, \mathbf{I}}_{n/2}, \mathbf{I}, \mathbf{D}, \mathbf{D}, \mathbf{I}, \mathbf{I}, \ldots$$

- el problema está en que la expansión y contracción se realizan con una misma cota al factor de carga
- luego en el peor caso puede haber expansiones y contracciones en $O(N)$ operaciones de una secuencia de N



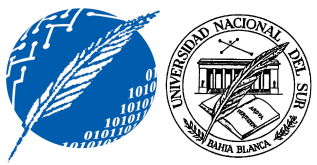
- se puede mejorarla bajando la cota inferior al factor de carga: ponerla en $0,25$ en lugar de $0,5$
- entonces, se expande cuando $\alpha(T) = 1$, pero sólo se contrae cuando $\alpha(T) < 0,25$
- después de una expansión vale $\alpha(T) = 0,5$ y deben ser eliminados la mitad de los elementos para que ocurra una contracción
- análogamente, después de una contracción $\alpha(T) = 0,5$ y deben ser insertados otros tantos elementos para que ocurra una expansión



- para el análisis amortizado la función potencial debe ser 0 inmediatamente después de una expansión y de una contracción; y aumentar a medida que el factor de carga se acerca a 1 o disminuye a $0,25$
- una posibilidad es:

$$\Phi(T_i) = \begin{cases} 2 * T_i.\text{elementos} - T_i.\text{tamaño} & \text{si } \alpha(T) \geq 0,5 \\ T_i.\text{tamaño}/2 - T_i.\text{elementos} & \text{si } \alpha(T) < 0,5 \end{cases}$$

- esta función cumple con los requerimientos de una función potencial



- el análisis amortizado para una inserción cuando $\alpha(T_{i-1}) \geq 0,5$ es idéntico al caso anterior
- si $\alpha(T_{i-1}) < \alpha(T_i) < 0,5$ entonces:

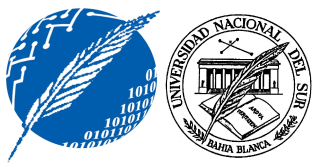
$$\begin{aligned} \hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= 1 + (tam_{i-1}/2 - (elem_{i-1} + 1)) - (tam_{i-1}/2 - elem_{i-1}) \\ &= 1 - 1 = 0 \in \Theta(1) \end{aligned}$$

- y si $\alpha(T_{i-1}) < 0,5$ pero $\alpha(T_i) \geq 0,5$ entonces:

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\ &= 1 + (2 * (elem_{i-1} + 1) - tam_{i-1}) - (tam_{i-1}/2 - elem_{i-1}) \\ &= 3 + 3 * elem_{i-1} - 3/2 * tam_{i-1} \\ &\leq 3 + 3 * (0,5 * tam_{i-1}) - 3/2 * tam_{i-1} = 3 \in \Theta(1) \end{aligned}$$

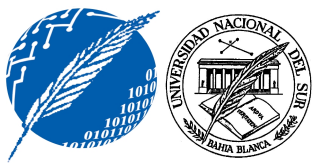

- el análisis amortizado para una eliminación cuando $\alpha(T_{i-1}) < 0,5$ tiene que considerar si existe contracción o no
- eliminiación sin contracción,

$$\begin{aligned}
\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= 1 + (tam_{i-1}/2 - (elem_{i-1} - 1)) - (tam_{i-1}/2 - elem_{i-1}) \\
&= 1 + 1 = 2 \in \Theta(1)
\end{aligned}$$

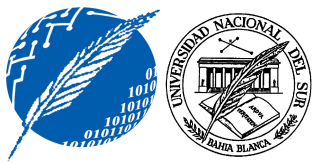


- eliminación con contracción, o sea $\alpha(T_{i-1}) = 0,25$

$$
\begin{aligned}
\hat{c}_i &= c_i + \Phi(T_i) - \Phi(T_{i-1}) \\
&= (elem_{i-1} + 1) + (tam_i/2 - elem_i) - \\
&\qquad (tam_{i-1}/2 - elem_{i-1}) \\
&= (elem_{i-1} + 1) + (tam_{i-1}/4 - (elem_{i-1} - 1)) - \\
&\qquad (tam_{i-1}/2 - elem_{i-1}) \\
&= elem_{i-1} + 2 - tam_{i-1}/4 \\
&= tam_{i-1}/4 + 2 - tam_{i-1}/4 = 2 \in \Theta(1)
\end{aligned}
$$



- el análisis amortizado para una eliminación cuando $\alpha(T_{i-1}) \geq 0,5$ queda como ejercicio
- es necesario considerar dos subcasos: cuando la eliminación ocasiona que el factor de carga pase la cota, y cuando la eliminación mantiene el factor de carga por encima de la cota



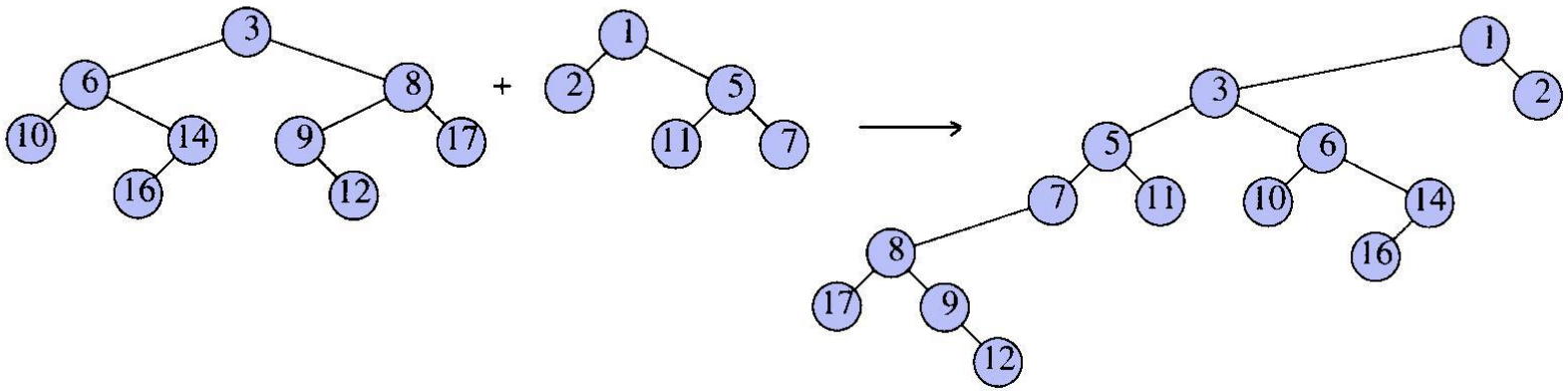
Heaps asimétricos

- un *heap* asimétrico (o *skew heap*) es un árbol binario que respeta la propiedad de *heap*, no tiene restricciones estructurales, y soporta las operaciones `inserción`, `eliminarMin` y `mezcla`
- un árbol satisface la propiedad de *heap* si cumple que el nodo con menor valor está en la raíz, y todos sus subárboles también cumplen la propiedad de *heap*
- las operaciones tienen una implementación muy sencilla
- `inserción` y `eliminarMinimo` se implementan en base a `mezclar`


- la operación `mezclar` tiene una implementación particular

```
PROCEDURE mezcla(T1,T2)
  IF T1=nil and T2=nil THEN this::=T2; RETURN
  IF (T2=nil) or (T1!=nil and T1.raiz()<T2.raiz())
    this.setRaiz(T1.raiz())
    this.setHD(T1.hi())
    this.setHI(T1.mezcla(T2, T1.hd())
  ELSE
    this.setRaiz(T2.raiz())
    this.setHD(T2.hi())
    this.setHI(T2.mezcla(T1,T2.hd()))
  ENDIF
  RETURN
```


Ejemplo



- en la práctica esto corresponde a una mezcla de los caminos derechos, y el resultado se inserta como el nuevo camino izquierdo
- en el peor caso esta operación toma tiempo de $O(n)$ ya que puede eventualmente recorrer todos los nodos de ambos árboles
- el tiempo real de la mezcla lleva tiempo proporcional a la cantidad de nodos en el camino derecho
- pero como estos nodos pasan a continuación a estar en el camino izquierdo, las próximas mezclas no los considerarán
- amerita realizar un análisis amortizado de esta operación


- para realizar un análisis amortizado de la operación se toma como función potencial a la cantidad de nodos pesados: aquellos nodos que tienen al menos la mitad de sus descendientes en el subárbol derecho
- los nodos que no son pesados se denominan livianos.
- la función potencial vale 0 en el árbol vacío, y nunca es negativa, con lo que cualquier sumatoria de costos amortizados que comiencen en vacío es una cota de los costos reales

- en un *heap* asimétrico, siempre la cantidad de nodos del camino derecho determina el tiempo de la operación `mezcla`
- acotar estos nodos nos permite determinar los tiempos amortizados

Lema 1

Sea T un heap *asimétrico de n elementos, entonces la cantidad de nodos livianos en el camino derecho es de $O(\log n)$.*

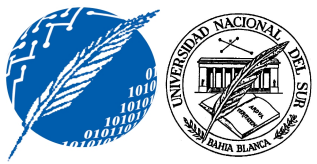
Demostración.

Por inducción sobre la cantidad de nodos en el camino derecho. □

- el análisis amortizado de la operación `mezcla`, siendo p_1, p_2 y l_1, l_2 la cantidad de nodos pesados y livianos del camino derecho de cada árbol mezclado, es:

$$\begin{aligned} \hat{c}_i &= p_1 + l_1 + p_2 + l_2 + \Phi(T_3) - \Phi(T_1, T_2) \\ &\leq p_1 + l_1 + p_2 + l_2 - p_1 - p_2 + l_1 + l_2 \\ &= 2(l_1 + l_2) \in O(\log n_1 + \log n_2) = O(\log n) \end{aligned}$$

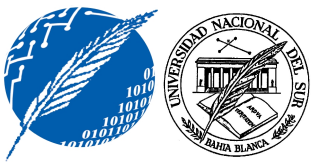
considerando que en una mezcla, todos los nodos pesados del camino derecho se transforman en nodos livianos en el resultado, mientras que algunos nodos livianos del camino derecho se transforman en nodos pesados



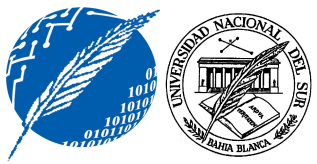
- las operaciones de `inserción` y `eliminarMinimo`, como dependen de la mezcla también tienen tiempo amortizado logarítmico
- esto implica que una serie de N operaciones sobre skew heaps, empezando de la estructura vacía, toman tiempo de $O(N \log n)$ en el peor caso.


Heaps de Fibonacci

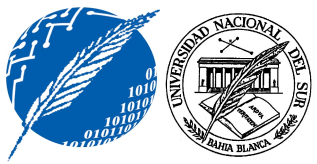
- los *heaps* de Fibonacci son un tipo de estructuras de datos que implementa las operaciones de los denominados mergeable heaps
- es decir sus operaciones son:
  - `crearHeap()`
  - `minimo():Elemento`
  - `insertar(x:Elemento)`
  - `eliminarMinimo():Elemento`
  - `mezclar(H1, H2:FibHeap)`
  - `disminuirClave(x:Elemento,k:Clave)`
  - `eliminar(x:Elemento)`



- si no se necesita las tres últimas operaciones, los *heaps* binarios tradicionales constituyen una implementación eficiente
- sin embargo, la mezcla es de tiempo de $\Theta(n)$ y la modificación de clave de $\Theta(\log n)$ en el peor caso (ejercicio)
- los *heaps* asimétricos soportan eficientemente la operación de `mezcla`, pero no la de `disminuirClave`
- los *heaps* de Fibonacci mejoran los tiempos de las dos últimas operaciones, a costa de conservar los tiempos de las restantes sólo bajo análisis amortizado en lugar del peor caso
- para estos tiempos, suponemos la búsqueda de un nodo resuelta, ie en tiempo constante

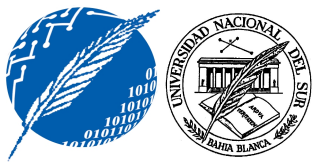


- la implementación de los *heaps* de Fibonacci está basada en otra E.D., las *heaps* binomiales que también son *mergeable heaps*
- una cola binomial es una foresta de árboles binomiales que cumple con las siguientes propiedades:
  - cada árbol de la foresta cumple con la propiedad de *heap*.
  - existe en la foresta a lo sumo un árbol binomial de cada rango.
- un árbol binomial de rango k, notado B_k, se define inductivamente como: B_0 tiene un sólo nodo, y B_k se forma enlazando dos árboles binomiales de rango $k-1$ de manera que uno sea el hijo extremo izquierdo del otro



Comparación heaps con mezcla

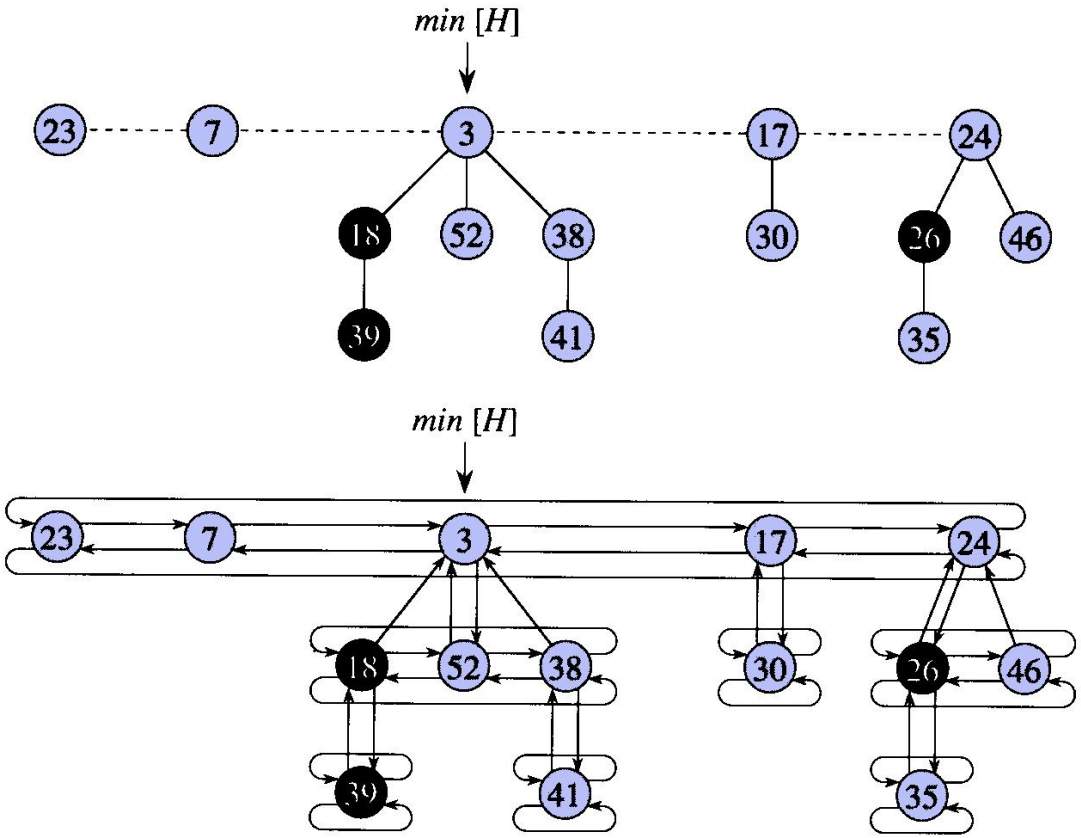
operación	*Heaps* binarios (peor caso)	*Heaps* binomiales (peor caso)	*Heaps* de Fibonacci (amortizado)
`crearHeap()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`minimo()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`insertar(x)`	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
`eliminarMinimo()`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
`mezclar(H1, H2)`	$\Theta(n)$	$O(\log n)$	$\Theta(1)$
`disminuirClave(x,k)`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
`eliminar(x)`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$



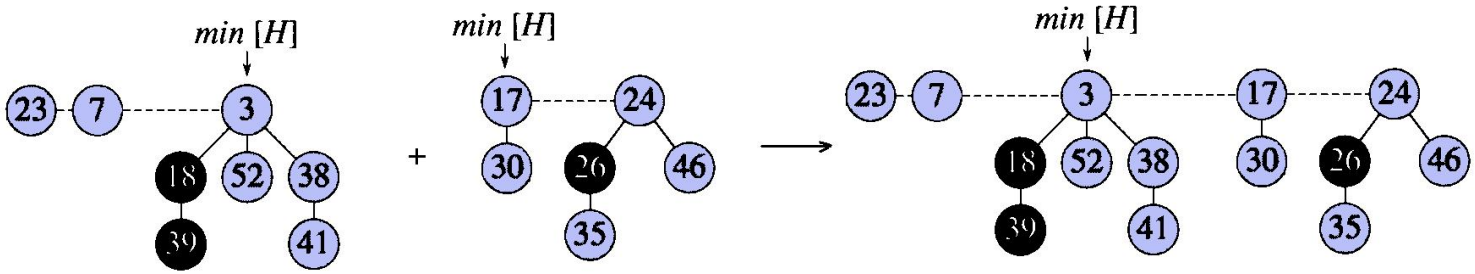
Heaps de Fibonacci

- un *heap* de Fibonacci se basa en la implementación de *heap binomial*. Estan formados por una foresta de árboles, los cuales se inician como árboles binomiales.
- la diferencia está en que a medida que se ejecutan las operaciones no necesariamente siempre estos árboles mantienen su estructura binomial
- su utilización sería útil en algoritmos como el de Dijkstra para los caminos más cortos con orígen único, donde en cada iteración de ciclo *greedy* no sólo se necesita seleccionar el nodo más próximo en el *heap*, sino también disminuir la distancia de los restantes nodos

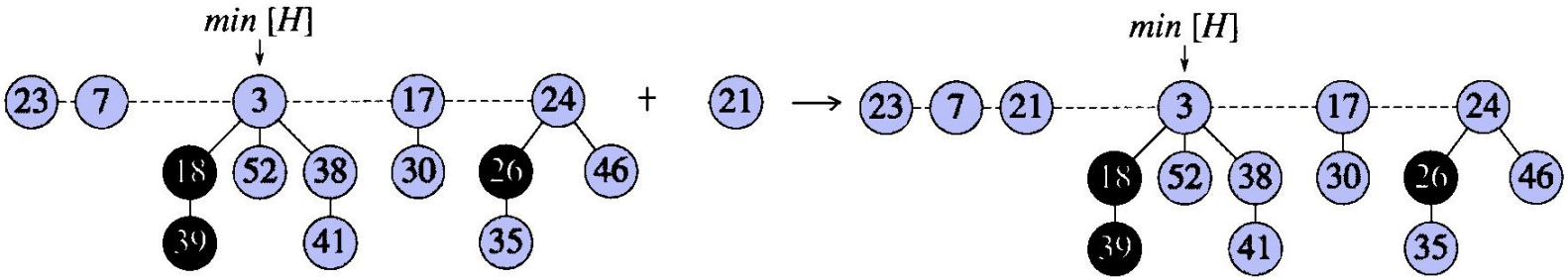

- las operaciones sobre los *heaps* de Fibonacci se diferencian de las de los *heaps* binomiales en dos aspectos:
  - mezcla perezosa: dos *heaps* se mezclan simplemente uniendo las forestas. Esto implica que no siempre exista un único árbol de cada rango. Favorece el tiempo de `mezcla` e `inserción`, pero aumenta el de `eliminarMinimo`.
  - cortes para mejorar el tiempo del *percolate* en la implementación de `eliminarMinimo`. Entonces cuando un nodo tiene clave menor que el padre, se elimina cortando el subárbol y agregándolo como un árbol nuevo a la foresta. Favorece el `disminuirClave`, pero perjudica a `eliminarMinimo`
  - cortes en cascada si un padre ha perdido más de un hijo, lo que asegura mantener la cantidad de descendientes de todos los nodos



min [H]
23
7
3
17
24
18
52
38
30
26
46
39
41
35
min [H]
23
7
3
17
24
18
52
38
30
26
46
39
41
35

- la operación de `Mezcla` de dos *heaps* de Fibonacci, como se implementa como mezcla perezosa, sólo realiza la unión de las dos forestas y calcula el nuevo mínimo



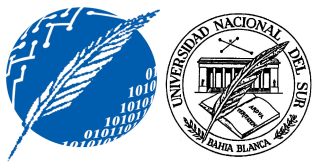
- la operación de `Inserción` se implementa simplemente agregado el nodo a insertar como nuevo árbol en la foresta



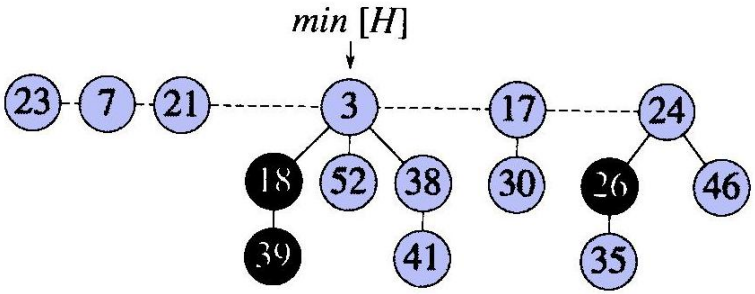
- la operación de `EliminarMinimo` es la encargada de restaurar la propiedad de que la foresta sea una colección de árboles donde existe a lo sumo uno de cada rango
- la implementación utiliza un procedimiento auxiliar `consolidar()` que controlan que para cada árbol no existe otro de su rango
- si esto sucede, los mezcla y crea un árbol de rango superior

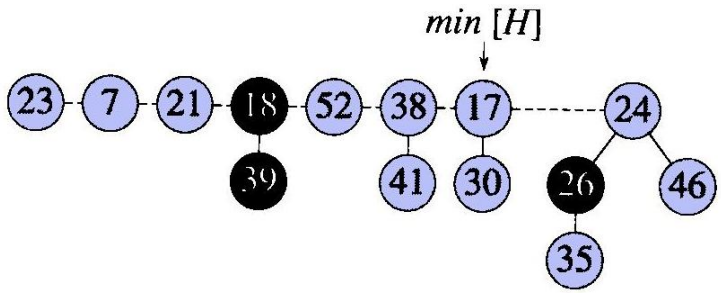

```
PROCEDURE EliminarMinimo()
 z::=this.minimo()
 IF z!=nil
   FOR cada hijo x de z
     agregar x a la foresta
   ENDFOR
   eliminar z de la foresta
   this.consolidar()
ENDIF
RETURN z
```

```
PROCEDURE consolidar()
  array grados[1..n]::=0
  FOR cada raíz w de un árbol de la foresta
    d::=w.grado(); x::=w
    WHILE grados[d]!=0
      y::=grados[d]
      IF x.clave()>y.clave()
        intercambiar(x,y)
      ENDIF
      unir x e y en x
      grados[d]::=0; d++
    ENDWHILE
    grados[d]::=x
  ENDFOR
  actualizar mínimo
```

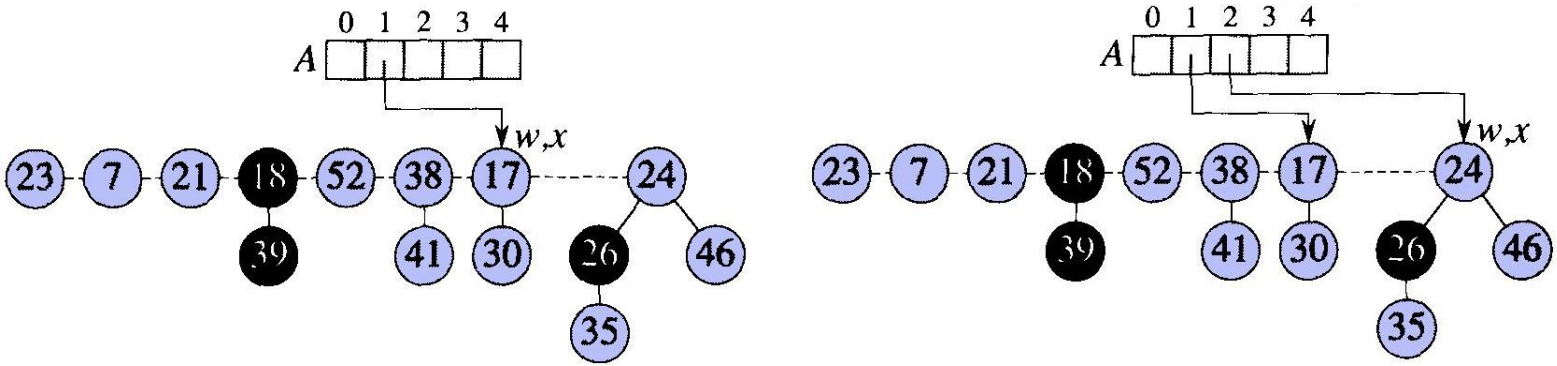


Ejemplo `eliminarMinimo()` (I)

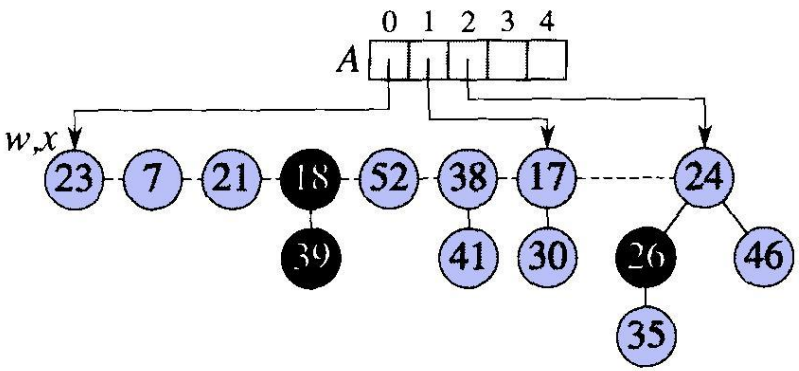


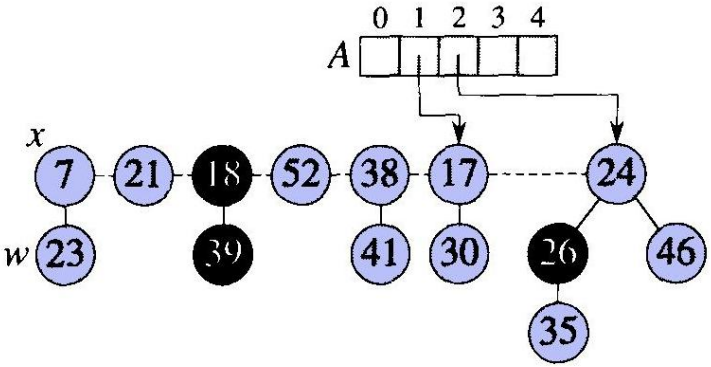


Ejemplo `eliminarMinimo()` (II) - consolidar

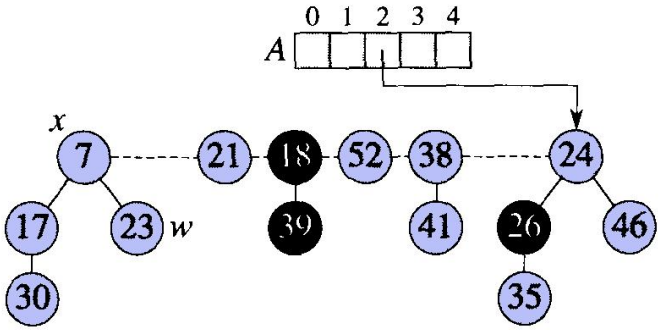


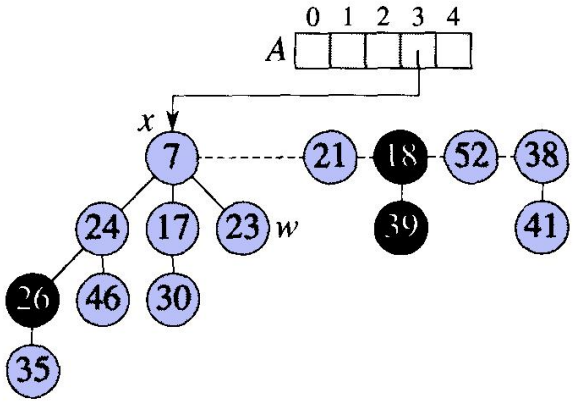
Ejemplo `eliminarMinimo()` (III) - consolidar



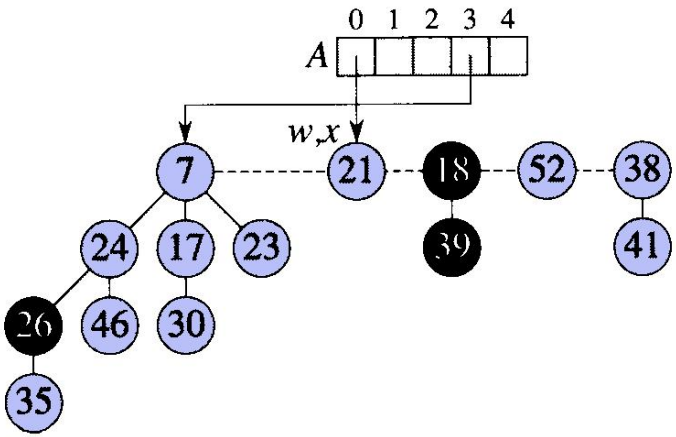


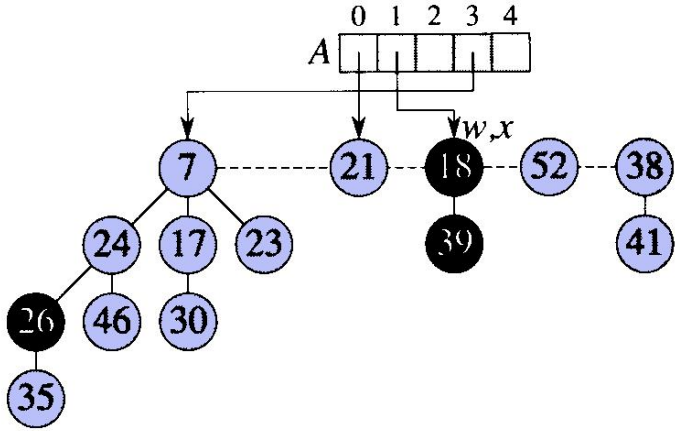
Ejemplo `eliminarMinimo()` (IV) - consolidar



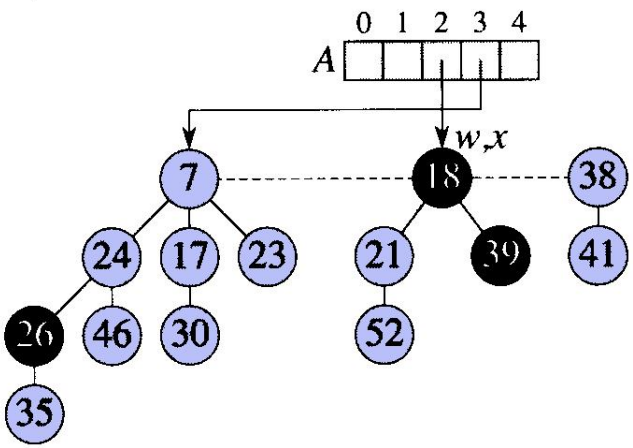


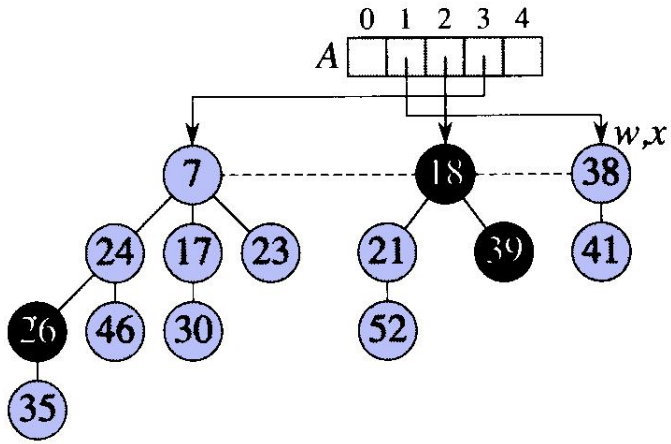
Ejemplo `eliminarMinimo()` (V) - consolidar



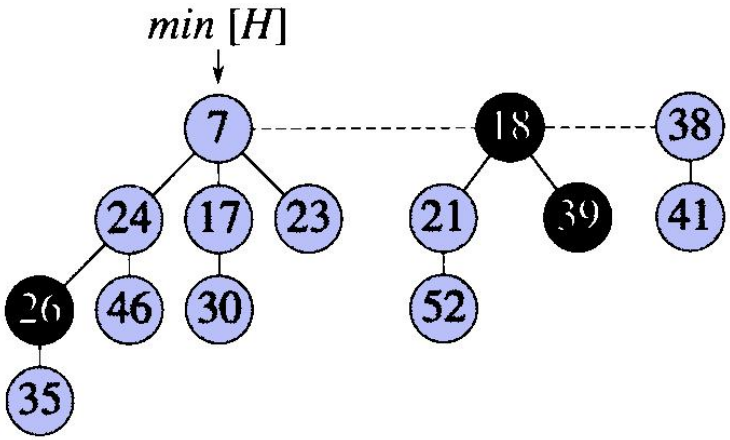


Ejemplo `eliminarMinimo()` (VI) - consolidar





Ejemplo `eliminarMinimo()` (VII) - consolidar



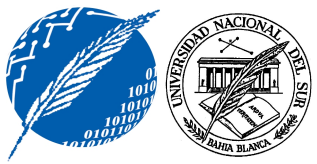
- la operación `disminuirClave` debe implementarse en tiempo amortizado constante, por lo que no es posible hacer un percolate por el árbol
- se supone que el nodo x ya viene dado. Si cuando se disminuye la clave se viola la propiedad de *heap*, entonces el nodo es cortado del árbol al que pertenece y se inserta como un árbol independiente en la foresta
- para asegurar que un nodo no pierda demasiados descendientes (y por lo tanto asegurar el tiempo amortizado del `eliminarMinimo`), entonces se marca el nodo que pierde un hijo por primera vez
- si un nodo pierde un segundo hijo, entonces también es cortado, se agrega su subárbol como árbol independiente en la foresta, y se procede con el padre

```
PROCEDURE DisminuirClave(x,k)
 clave[x]::=k; y::=padre[x]
 IF y!=nil y clave[x]<clave[y]
   H.corte(x,y)
   H.corteCascada(y)
 ENDIF
 IF clave[x]<clave[H.minimo()]
   H.setMinimo(x)
 ENDIF
```

```
PROCEDURE corte(x,y)
 eliminar x de la lista de hijos de y,
   actualizando su rango
 agregar x a la foresta
 marcado[x]::=false
```

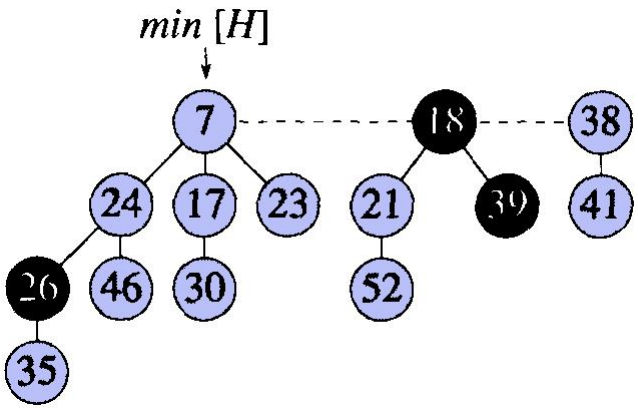
```
PROCEDURE corteCascada(y)
 z::=padre[y]
 IF z!=nil
   IF no marcado[y]
     marcado[y]::=true
   ELSE
     H.corte(y,z)
     H.corteCascada(z)
   ENDIF
 ENDIF
```

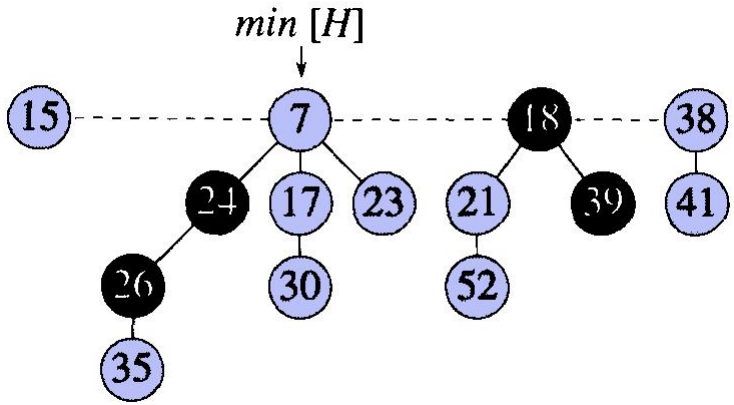
- una llamada a `DisminuirClave` tiene como costo real 1 más la cantidad de cortes en cascada



Ejemplos (I)

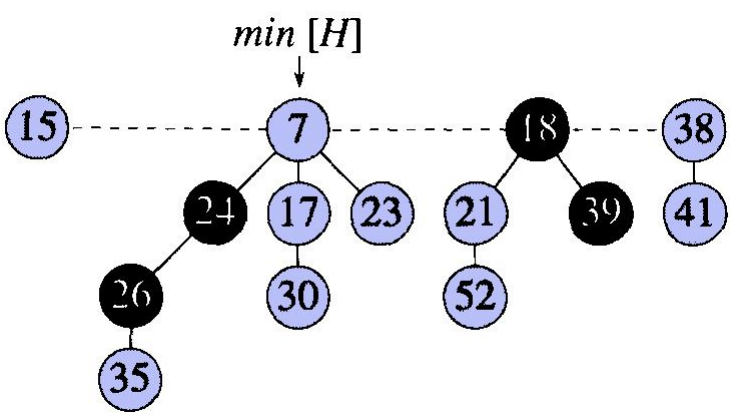
- `disminuirClave(46,15)`

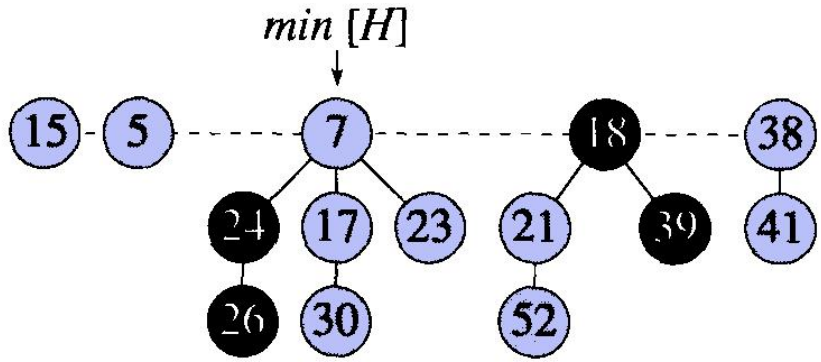




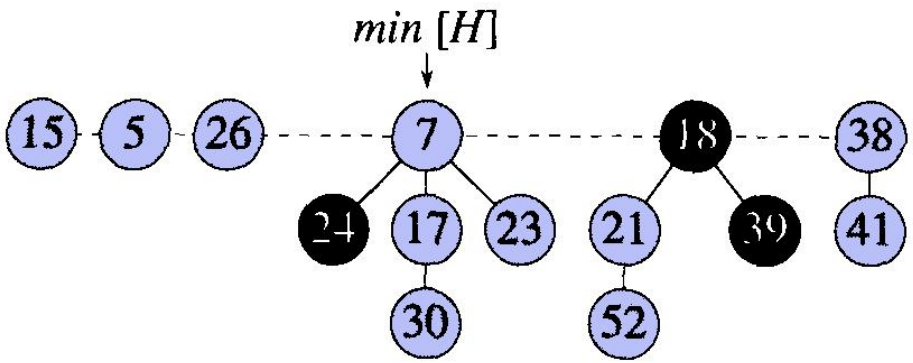
Ejemplos (II)

- `disminuirClave(35,5)`

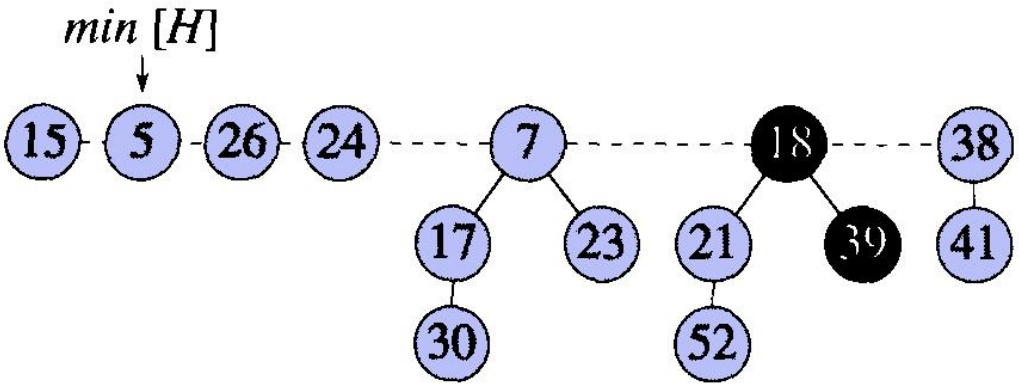




Ejemplos (III)



Ejemplos (IV)



Análisis amortizado de las operaciones

- para realizar el análisis amortizado de esta E.D. se usa como función potencial:

$$\Phi(H) = \mathit{arboles}(H) + 2 \times \mathit{marcados}(H)$$

- esta función satisface los requisitos para una función potencial comenzando del *heap* vacío

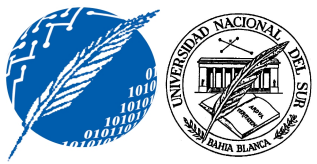
- en la operación `insertar` se agrega un árbol y los nodos marcados no cambian, luego:

$$\hat{c}_i = c_i + \Phi(H_i) - \Phi(H_{i-1}) = \Theta(1) + 1 \in \Theta(1)$$

- en la operación `mezcla` la cantidad de árboles y nodos marcados no cambian:

$$\hat{c}_i = c_i + \Phi(H_i) - \Phi(H_{i-1}) = \Theta(1) + 0 \in \Theta(1)$$

- para la operación `eliminarMinimo` serán necesarias algunas propiedades, que veremos a continuación

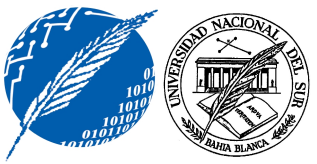


Lema 2

Sea x un nodo en un heap *de Fibonacci tal que* $\texttt{grado}[x] = k$. *Entonces para todo hijo* $y_i, 2 \leq i \leq k$ *vale que* $\texttt{grado}[y_i] \geq i - 2$.

Demostración.

En un *Heap* de Fibonacci la única posibilidad de y_i sea colocado como hijo de x es que $\texttt{grado}[x] = \texttt{grado}[y_i]$. Y como y_i es el i-ésimo hijo, en ese momento tanto x como y_i tenían $i-1$ hijos. Luego y_i pierde a lo sumo un hijo, con lo que $\texttt{grado}[y_i] \geq i - 2$. □



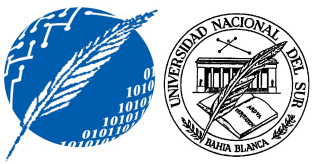
Lema 3

Sean F_k los números de Fibonacci, luego $\sum_{i=0}^{k} F_i = F_{k+2} - 1$.

Demostración.

Por inducción sobre k. Si $k = 0$ es trivial. Suponiendo que vale para $k-1$, entonces
$\sum_{i=0}^{k} F_i = \sum_{i=0}^{k-1} F_i + F_k = (F_{k+1} - 1) + F_k = F_{k+2} - 1$. □

- se nota con $des(x)$ la cantidad de descendientes de un nodo x



Teorema 4

Sea x un nodo de un heap *de Fibonacci tal que* $\texttt{grado}[x] = k$. *Encontes la cantidad de descendientes de x es al menos F_{k+2}.*

Demostración.

Por inducción sobre el rango k de x. Si $k = 0$ vale. Si $k > 1$, sean $y_1, \ldots, y_k$ los hijos de x en el orden de inserción. Entonces

$$\begin{aligned} des(x) &= 1 + 1 + \sum_{i=2}^{k} des(y_i) \geq 1 + 1 + \sum_{i=2}^{k} F_i = \\ &= 1 + \sum_{i=0}^{k} F_i = 1 + F_{k+2} - 1 = F_{k+2} \end{aligned}$$

□

Corolario 5

El rango de un nodo en un heap *de Fibonacci de n nodos es siempre* $O(\log n)$.

Demostración.

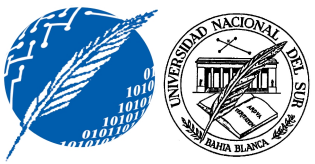
Sea x un nodo de un *heap* de Fibonacci tal que `rango`$[x] = k$. Por el teorema 4, $n \geq s(x) \geq F_{k+2} \geq \phi^k$. Luego $k \leq \log_\phi \in O(\log n)$. □

- retomando el ánalisis amortizado de `eliminarMinimo()`, sea r el grado de la raíz que contiene la menor etiqueta, T la cantidad de árboles en el momento $i-1$ y n la cantidad de elementos almacenados

- el costo real de la operación es $T + r$, y los nodos marcados no cambian

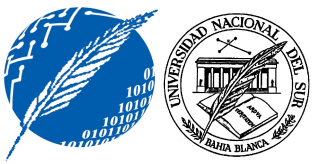
$$\begin{aligned} \hat{c}_i &= c_i + \Phi(H_i) - \Phi(H_{i-1}) = T + r + O(\log n) - T \\ &\leq r + O(\log n) \leq O(\log n) + O(\log n) \in O(\log n) \end{aligned}$$

usando el corolario anterior para acotar r y la cantidad de árboles después de la consolidación (ya que el rango máximo de un nodo es una cota de la cantidad máxima de árboles)



- en la operación `disminuirClave` el costo real es 1 más la cantidad de cortes en cascada C
- sea M la cantidad de nodos marcados antes de la operación. La cantidad de árboles aumenta en $1 + C$, C nodos marcados dejan de serlo, y un nodo no marcado pasa a marcado

$$\begin{aligned} \hat{c}_i &= c_i + \Phi(H_i) - \Phi(H_{i-1}) \\ &\leq 1 + C + (T + 1 + C + 2(M - C + 1)) - (T + 2M) = \\ &= 4 \in \Theta(1) \end{aligned}$$



Comparación heaps con mezcla

operación	*Heaps* binarios (peor caso)	*Heaps* binomiales (peor caso)	*Heaps* de Fibonacci (amortizado)
`crearHeap()`	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
`minimo()`	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
`insertar(x)`	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
`eliminarMinimo()`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
`mezclar(H1, H2)`	$\Theta(n)$	$O(\log n)$	$\Theta(1)$
`disminuirClave(x,k)`	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
`eliminar(x)`	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$