Algoritmos y Complejidad

Estructuras de Datos

Pablo R. Fillottrani

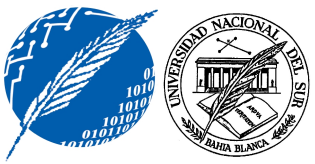
Depto. Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

primer semestre 2024

Estructuras de Datos


Heaps

- también llamados *colas con prioridad*
- pueden estar implementadas para maximizar (maxheap) o minimizar (minheap) alguna prioridad, u orden entre los elementos
- supondremos en lo que sigue que se trata de un maxheap, de lo contrario es fácil reemplazar el orden
- operaciones: Insertar(x), EliminarMax() (o EliminarMin()), CrearHeap(T)
- existen muchas implementaciones posibles, pero la más usada es la que usa árboles binarios casi completos



Implementación

- un arreglo A se usa para representación el arreglo binario casi completo, o sea que todos los niveles están llenos con la posible excepción del último
- la raíz está en $A[1]$ y los elementos del heap están almacenados desde $A[1]$ hasta $A[A.\text{tamano}]$

Implementación

- dado un elemento del heap en la posición *i* se calculan los índices de su padre y sus hijos con las funciones:

```
PADRE(i)
RETURN i div 2

HIJOIZQ(i)
RETURN 2*i

HIJODER(i)
RETURN 2*i+1
```

- para aumentar la eficiencia, estas operaciones pueden computarse con números en binario

Prioridad

- todos los elementos en posición $i > 1$ del heap (es decir todos menos la raíz) satisfacen la propiedad de heap: $A[\texttt{PADRE}(i)] \geq A[i]$
- esto implica que un elemento con la máxima prioridad se encuentra en $A[1]$ (puede haber muchos con esa prioridad)

Altura

- la altura de un nodo en un heap es el número de arcos en el camino simple hacia abajo más largo desde el nodo hasta una hoja del árbol
- la altura de un heap es la altura de la raíz del heap

Lema 1

La altura de un maxheap *de n elementos es* $\lfloor log_2 n \rfloor$

Lema 2

Para cualquier subárbol de un maxheap*, la raíz del subárbol es un elemento con la máxima prioridad en el subárbol*

(las demostraciones son simples y quedan como ejercicio)

Operaciones

- MaxHeapify(i) es un procedimiento auxiliar para mantener la propiedad de heap de un subárbol i, de $\Theta(\log n)$
- CrearHeap(A) produce un heap a partir de un arreglo desordenado inicial, de $\Theta(n)$
- Insertar(x) ingresa un nuevo elemento en el heap, manteniendo la propiedad de heap, de $\Theta(\log n)$
- ElmininarMax() retorna un elemento con prioridad máxima del heap, eliminándolo al mismo tiempo, de $\Theta(\log n)$


MaxHeapify

```
MAXHEAPIFY(A,i)
l := HIJOIZQ(i); r := HIJODER(i)
IF  l<=A.tamano and A[l]>A[i]
    max := l
ELSE max := i
IF r<=A.tamano and A[r]>A[max]
    max:=r
IF max<>i
    intercambiar A[i] con A[max]
    MAXHEAPIFY(A,max)
```

- el tiempo de ejecución en un nodo i de altura h es de $\Theta(h)$ (ejercicio: resolver recurrencia)

Construyendo un Heap

- se puede usar MAXHEAPIFY(A, i) en forma *bottom-up* para convertir un arreglo $A[1..n]$, donde $n = A$.tamano en un heap

```
CREARHEAP(A)
FOR i:= A.tamano DIV 2 downto 1
  MAXHEAPIFY(A, i)
```

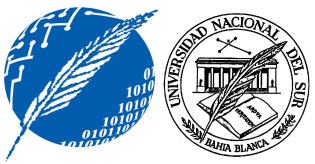
- el tiempo de ejecución es $O(n \log n)$ sabiendo que cada altura $h \in O(\log n)$ y hay $n/2$ llamadas a MAXHEAPIFY
- la correctitud del algoritmo se basa en probar por inducción el siguiente invariante de ciclo: en cada iteración, todos los nodos $i+1, i+2, \ldots, n$ son raíz de un maxheap (ejercicio)

Cota estricta

Lema 3

Si A es un maxheap *con n elementos, entonces hay a lo sumo* $\lceil n/2^{h+1} \rceil$ *elementos de altura h*

- Ejercicio: probar este lema



Cota estricta

Teorema 4

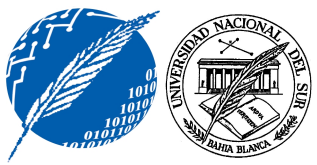
CrearHeap(A) *tiene un tiempo de ejecución* $T(n) \in \Theta(n)$ *donde n es la cantidad de elementos de A*

Demostración.

Como el tiempo de MAXHEAPIFY(A, i) es $\Theta(\log h)$ y el lema anterior:

$$T(n) = \Sigma_{h=0}^{\lfloor \log n \rfloor} \lceil n/2^{h+1} \rceil \Theta(h) = \Theta(n \Sigma_{h=o}^{\lfloor \log n \rfloor} (h/2^h))$$

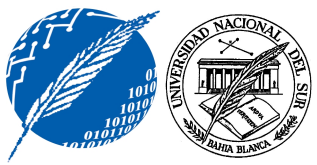
Acotando $\Sigma_{h=0}^{\lfloor \log n \rfloor} h/2^h \leq \Sigma_{h=0}^{\infty} h/2^h = 2$ se llega a $T(n) \in \Theta(n)$. □



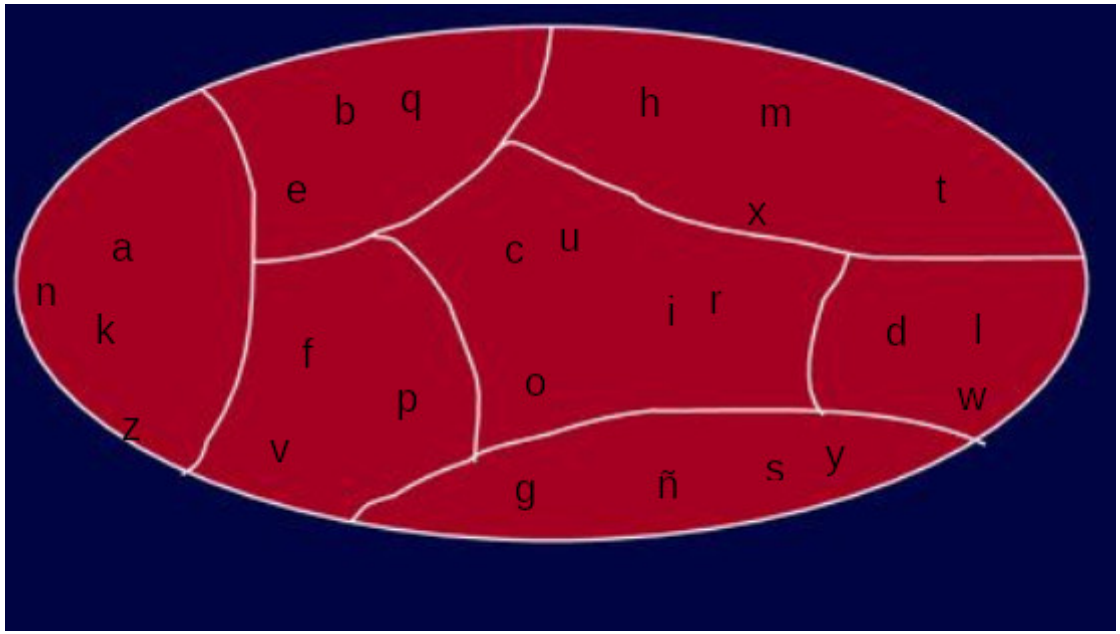
Heapsort

	costo	veces
`FUNCTION Heapsort(A)`		
`CrearHeap(A)`	$\Theta(n)$	1
`FOR i ::= n DOWNTO 2`	c	$\sum_{i=2}^{n} 1$
`  intercambiar A[1] y A[i]`	c	$\sum_{i=2}^{n} 1$
`  A.tamano- -`	c	$\sum_{i=2}^{n} 1$
`  MAXHEAPIFY(A,1)`	$\Theta(\log n)$	$\sum_{i=2}^{n} 1$

$$
\begin{aligned}
T_H(n) &= \Theta(n) + \sum_{i=2}^{n} (c_1 + c_2 + c_3 + \Theta(\log n)) = \\
&= \Theta(n) + \Theta(n \log n) \in \Theta(n \log n)
\end{aligned}
$$

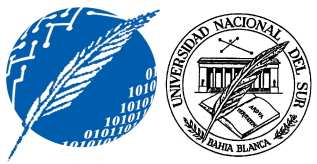


Particiones dinámicas



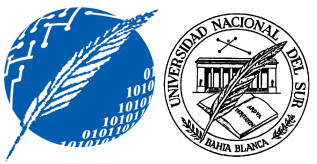
Particiones dinámicas

- la estructura de datos Particiones Dinámicas, o *disjoint sets*, se usa para almacenar n elementos agrupados en particiones que pueden cambiar durante la ejecución
- cada partición es disjunta a todas las otras (no se solapan), y todo elemento pertenece a una partición
- el cambio que se puede producir es una unión, o mezcla, de dos particiones
- al inicializar, en general, cada elemento pertence a una partición singleton, que solo contiene ese elemento



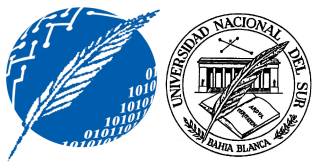
Particiones dinámicas

- entonces una partición dinámica caracteriza una serie de conjuntos $S_1, S_2, \ldots, S_k$
- se identifica cada conjunto S_i con un elemento representante, que pertenece al conjunto
- en general no hay condiciones específicas que el representante debe cumplir



Operaciones

- CrearConjunto(x) crea un nuevo conjunto cuyo único elemento es x
- Unir(x,y) mezcla las particiones que contienen a x y a y. El nuevo representante puede ser cualquier elemento del nuevo conjunto
- Encontrar(x) retorna el representante del conjunto al que x pertence

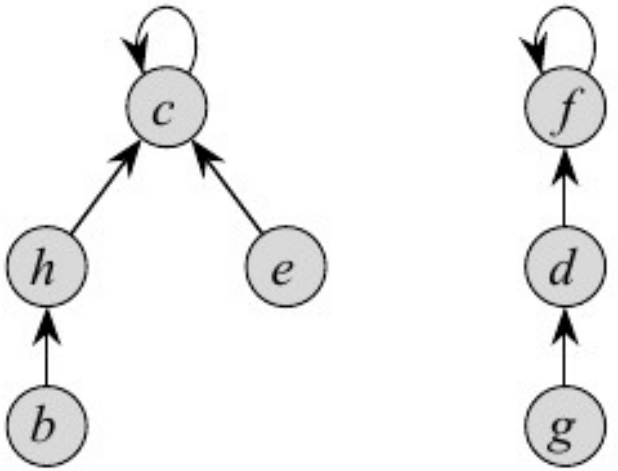


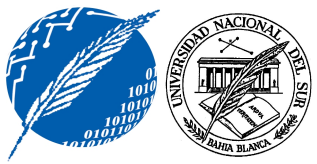
Aplicación

- Ejercicio: podría pensarse un algoritmo para CFC que inicialice una partición dinámica con todos los nodos de un grafo, y luego recorra todos los arcos uniendo los conjuntos de los extremos
- este algoritmo involucraría $\Theta(n+a)$ llamadas a operaciones de esta ED

Implementación: forestas de árboles

- se representan cada conjunto con un árbol con sus miembros, cada miembro apunta a su padre en el árbol y la raíz de cada árbol es padre de sí mismo y el representante del conjunto
- se implementa CrearConjunto(x) creando un árbol de un solo nodo; Unir(x,y) haciendo que una raíz apunte a la otra y Encontrar(x) haciendo el recorrido desde el nodo hasta su raíz



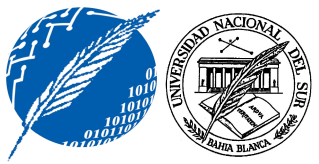


Heurísticas para las operaciones

- las operaciones Unir y Encontrar con la implementación vista son de $O(n)$
- se agregan dos heurísticas para mejorar este tiempo
- unión por rango: se mantiene un *rango* en cada nodo que es una cota superior de su altura, y en Unir el nodo de rango inferior es colocado como hijo del nodo con rango superior
- compresión de caminos cada vez que se ejecuta un Encontrar, todos los nodos visitados son puestos como hijos de la raíz, sin actualizar el rango

Implementación de Encontrar con compresión de caminos

```
FUNCTION Encontrar(x)
IF x<>x.padre THEN
    x.padre ::= Encontrar(x.padre)
RETURN x.padre
```



Implementación de Encontrar con compresión de caminos

- Encontrar es un procedimiento de dos pasos: primero se hacen llamadas recursivas hasta encontrar la raíz; luego al deshacerse la pila de recursión se actualizan los padres de los nodos visitados
- el tiempo es de $O(n)$


Implementación de Unir con unión por rango

```
PROCEDURE Link(x,y)
IF x.rango > y.rango THEN
   y.padre ::= x
ELSE
   x.padre ::= y
   IF x.rango == y.rango THEN y.rango++

PROCEDURE Unir(x,y)
Link(Encontrar(x),Encontrar(y))
```

Implementación de Unir con unión por rango

- el procedimiento Unir simplemente realiza una búsqueda de los representantes de cada nodo, y llama al procedimiento auxiliar Link
- Link realiza el control el rango y el enlace
- por las llamadas a Encontrar, Unir también es de tiempo de $O(n)$

Cota del tiempo de una secuencia de operaciones

- cada ejecución de la operación Encontrar hace que las siguientes operaciones sean más eficientes
- esto no puede reflejarse en el tiempo en el peor caso de una sola llamada a la operacion; se usa entonces una análisis amortizado
- en el análisis amortizado se analiza el tiempo en el peor de los casos de una secuencia de K llamadas a las operaciones de la estructura de datos
- el detalle de este análisis puede verse en la [CLRS22, sección 19.4]

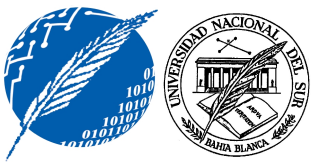
Cota del tiempo de una secuencia de operaciones con las heurísticas

- se puede demostrar que una secuencia de K llamadas a las operaciones CrearConjunto, Encontrar y Unir es de tiempo de $O(K \log^*(n))$ en el peor de los casos
- $\log^*(n)$ es el logaritmo iterado (la cantidad de veces que se puede aplicar logaritmo a un número) y es una función de crecimiento muy lento, mucho menor que $\log(n)$, y a efectos prácticos se puede considerar constante

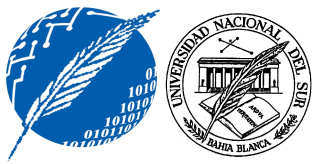

Cota del tiempo de un secuencia de operaciones con las heurísticas

$$\begin{aligned} \log^* 2 &= 1 \\ \log^* 4 &= 2 \\ \log^* 16 &= 3 \\ \log^* 65536 &= 4 \\ \log^*(2^{65536}) &= 5 \end{aligned}$$

- dado que el número de átomos en el universo observable está estimado en aproximadamente $10^{80} << 2^{65536}$, es raro encontrar un valor de n tal que $\log^* n > 5$



- son una ED de mergeable heap, o sea un heap que incluye la operación de mezcla
- un heap 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



Propiedades árboles Binomiales (I)

Lema 5

Sea B_k el árbol binomial de grado k, entonces

1. *B_k tiene 2^k nodos*
2. *B_k tiene altura k*
3. *en B_k existen exactamente $\binom{k}{i}$ nodos de profundidad i*
4. *la raíz de B_k es de grado k (cantidad de hijos), y sus hijos son (de izquierda a derecha) de grados $0, 1, \ldots, k-2, k-1$*

Demostración.

Queda como ejercicio, usar inducción sobre el grado k en todos los casos. □

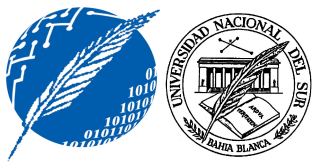
Propiedades árboles Binomiales (II)

Lema 6

El máximo grado de un nodo en un árbol binomial de n nodos es $\log n$.

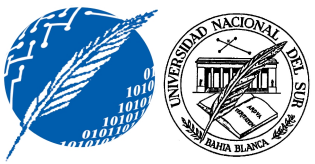
Demostración.

Inmediato de las propiedades 1 y 4 del lema 5. □

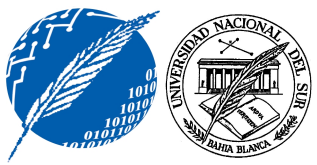


Operaciones *Heap Binomial*

- cada árbol de un *heap binomial* es un árbol binomial que además cumpled con la propiedad de *heap*, o sea que el nodo con menor clave de un árbol está en la raíz, y todos sus subárboles también cumplen la propiedad de *heap*
- además también satisfacen que si el *heap* tiene n nodos entonces existen a lo sumo $\lfloor \log n \rfloor + 1$ árboles



- la implementación de las operaciones es la siguiente:
  - `crearHeap()` produce una foresta vacía, y es de $\Theta(1)$.
  - `minimo()` se puede implementar manteniendo un puntero al árbol con menor clave en tiempo $\Theta(1)$
  - `mezclar(H1, H2)` se puede realizar mediante a un proceso análogo a la suma binaria, componiendo árboles binomiales de rangos repetidos en un árbol de rango mayor. Esto lleva tiempo de $\Theta(\log n)$
  - `eliminarMinimo()` e `insertar(x)` se implementan en base a la operación de mezcla
  - `disminuirClave(x, k)` debe recorrer a lo sumo la máxima altura del árbol más alto en la foresta, lo que es de $O(\log n)$ de acuerdo a las propiedades vistas.
  - `eliminar(x)` se implementa disminuyendo la clave del elemento al mínimo posible, y luego llamando a `eliminarMinimo()`



Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.

Introduction To Algorithms.

The MIT Press, 4th edition, 2022.