

Algoritmos y estructuras de datos

TAD. Montón binario. Cola de prioridad.

CEIS

Escuela Colombiana de Ingeniería

2024-1

Agenda

- ① TADS
- ② Montón binario
- ③ Cola de prioridad
- ④ Aspectos finales
Ejercicios

Agenda

- 1 TADS
- 2 Montón binario
- 3 Cola de prioridad
- 4 Aspectos finales
Ejercicios

Agenda

- 1 TADS
- 2 Montón binario
- 3 Cola de prioridad
- 4 Aspectos finales
Ejercicios

Montón binario

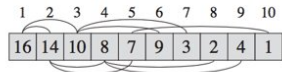
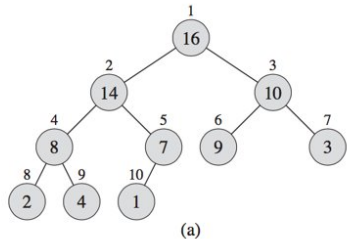
Heap

La estructura de datos *heap* es un objeto similar a un arreglo, que se puede ver como **un árbol binario casi completo**. Cada nodo del árbol corresponde a un elemento del arreglo.

Un array *A* que representa un *heap*, es un objeto con dos atributos:

- *A.length* que es el número de elementos en el array
- *A.heap_size*, que representa cuantos elementos en el *heap* están almacenados en el array *A*.

A[1..*A.length*] contiene los números, pero solo los elementos de *A*[1..*A.heap_size*], son elementos válidos del *heap*.



$$0 \leq A.\text{heap_size} \leq A.\text{length}$$

(b)

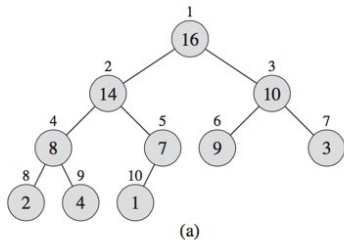
Montón binario

Heap

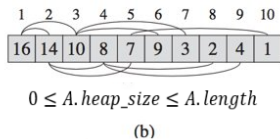
Viendo el *heap* como un árbol, se define:

- La altura de un nodo en el *heap*, como el número de arcos en la ruta hacia abajo más larga del nodo a una hoja
- La altura del *heap* como la altura de la raíz

La altura del *heap* es $\Theta(\log n)$



Varias operaciones en el *heap* corren en un tiempo proporcional a la altura del árbol



Montón binario

- La raíz del árbol es $A[1]$
- Dado el índice i de un nodo, se puede determinar el padre, el hijo izquierdo y el hijo derecho.

PARENT(i)

1 return $\lfloor i/2 \rfloor$

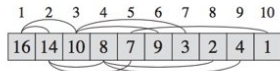
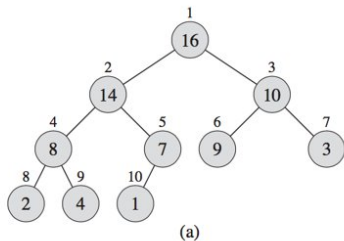
LEFT(i)

1 return $2i$

RIGHT(i)

1 return $2i + 1$

Heap



$$0 \leq A.\text{heap_size} \leq A.\text{length}$$

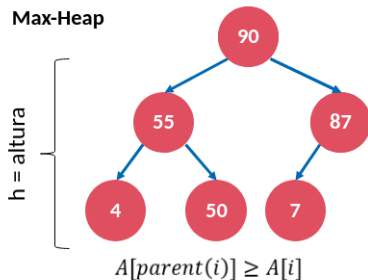
(b)

Montón binario

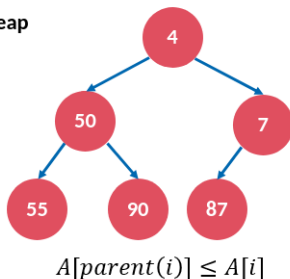
Existen dos tipos de *heaps*: *Max-Heap* y *Min-Heap*. En ambos casos los valores de los nodos satisfacen la condición del *heap*.

- *Max-Heap*: Para todo nodo i diferente a la raíz
 $A[\text{PARENT}(i)] \geq A[i]$
- *Min-Heap*: Para todo nodo i diferente a la raíz
 $A[\text{PARENT}(i)] \leq A[i]$

Max-Heap



Min-Heap



Montón binario

Las operaciones asociadas a un *max-heap*:

- MAX-HEAPIFY, es clave para mantener la propiedad de *Max-Heap*.

$O(\log n)$

- BUILD-MAX-HEAP, produce un *Max-Heap* de un array no ordenado.

$O(n)$

- HEAPSORT , ordena un array en tiempo

$O(n \log n)$

- MAX-HEAP-INSERT

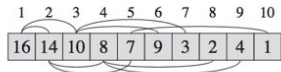
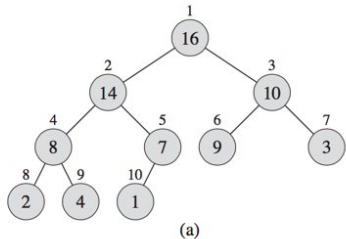
- HEAP-EXTRACT-MAX

- HEAP-INCREASE-KEY

- HEAP-MAXIMUM, permiten a un *heap* implementar una cola de prioridad.

$O(n \log n)$

Max-Heap



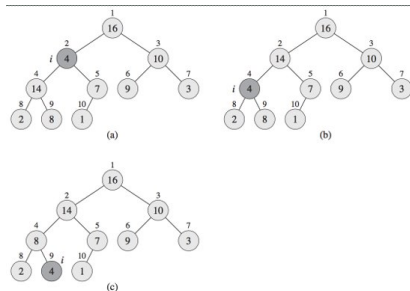
Montón binario. MAX-HEAPIFY.

Para mantener la propiedad de *Max-Heap*, se usa el método MAX-HEAPIFY.

Las entradas son un array A y un índice i dentro del array. Cuando se llama al método, se asume que los árboles con raíz en $\text{LEFT}(i)$ y $\text{RIGHT}(i)$ son *Max-Heaps*, pero $A[i]$ puede ser menor que sus hijos, violando la condición.

MAX-HEAPIFY(A, i)

```
1   $l = \text{LEFT}(i)$ 
2   $r = \text{RIGHT}(i)$ 
3  if  $l \leq A.\text{heap-size}$  and  $A[l] > A[i]$ 
4       $\text{largest} = l$ 
5  else  $\text{largest} = i$ 
6  if  $r \leq A.\text{heap-size}$  and  $A[r] > A[\text{largest}]$ 
7       $\text{largest} = r$ 
8  if  $\text{largest} \neq i$ 
9      exchange  $A[i]$  with  $A[\text{largest}]$ 
10     MAX-HEAPIFY( $A, \text{largest}$ )
```

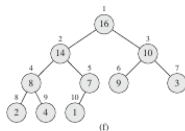
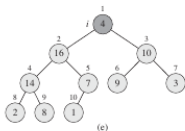
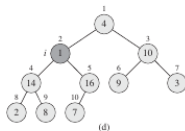
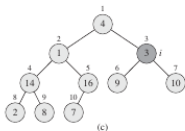
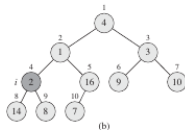
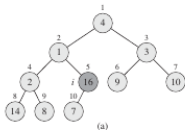


Montón binario. BUILD-MAX-HEAP.

Se puede usar el procedimiento MAX-HEAPIFY de una manera bottom-up para convertir un arreglo $A[1..n]$, donde $n = A.length$, en un *Max-Heap*. Todos los elementos $A[(n/2)+1..n]$ son hojas del árbol, por ende un heap de un solo elemento.

A

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



BUILD-MAX-HEAP(A)

- 1 $A.heap-size = A.length$
- 2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
- 3 MAX-HEAPIFY(A, i)

Montón binario. HEAPSORT

HEAPSORT(*A*)

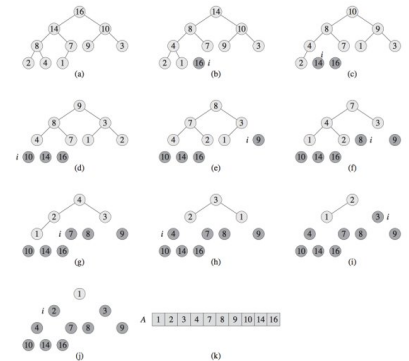
```

1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size - 1
5      MAX-HEAPIFY(A, 1)
    
```

El algoritmo heapsort inicia usando BUILD-MAX-HEAP para construir un *Max-Heap* del arreglo de entrada $A[1..n]$, donde $n = A.length$.

Como el máximo elemento del arreglo es almacenado en la raíz $A[1]$, se puede poner en la posición final correcta al cambiarla con $A[n]$.

Si se descarta el nodo n del heap, podemos decrementar el tamaño del *heap* (*heap-size*) y verificar que el nuevo elemento raíz no viole la propiedad del *Max-Heap*.



Montón binario. HEAPSORT

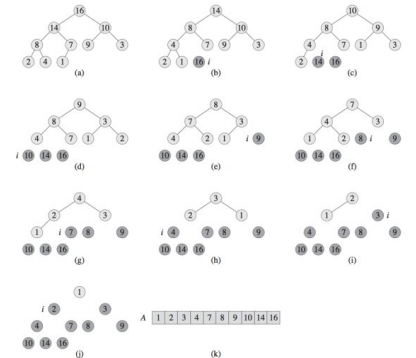
HEAPSORT(A)

```

1  BUILD-MAX-HEAP( $A$ )
2  for  $i = A.length$  downto 2
3      exchange  $A[1]$  with  $A[i]$ 
4       $A.heap-size = A.heap-size - 1$ 
5      MAX-HEAPIFY( $A, 1$ )
    
```

Reúne las dos mejores características de *insertion-sort* y *merge-sort*. Ordena en el lugar y su tiempo de ejecución es el mismo de *merge-sort*.

$O(n \log n)$



Ejemplos

Max-Priority Queue

Programar la ejecución de tareas en un computador compartido:

- Cada tarea se guarda con su prioridad para ejecutarse
- Cuando una tarea termina, el computador toma la siguiente más importante (mayor prioridad)

INSERT(S, x)

MAXIMUM(S)

EXTRACT-MAX(S)

INCREASE-KEY(S, x, k)

Min-Priority Queue

Simulador orientado por eventos:

- Cada evento se guarda con la fecha en que debe ocurrir (su *key*)
- Cada evento debe ejecutarse según su tiempo de ocurrencia, por lo que se toman los más próximos para correr primero (menor prioridad)

INSERT(S, x)

MINIMUM(S)

EXTRACT-MIN(S)

DECREASE-KEY(S, x, k)

Agenda

- 1 TADS
- 2 Montón binario
- 3 Cola de prioridad
- 4 Aspectos finales
Ejercicios

Colas de prioridad

Datos

Una **cola de prioridad** es una estructura de datos para mantener un conjunto S de elementos, cada uno asociado con un valor llamado llave. Reúne las dos mejores características de *insertion-sort* y *merge-sort*. Ordena en el lugar y su tiempo de ejecución es el mismo de *merge-sort*.

Operaciones

- $\text{INSERT}(S, x)$ inserta el elemento x en el conjunto S .
- $\text{MAXIMUM}(S)$ retorna el elemento de S con la llave mas grande.
- $\text{EXTRACT-MAX}(S)$ remueve y retorna el elemento de S con la llave mas grande
- $\text{INCREASE-KEY}(S, x, k)$ incrementa el valor de la llave del elemento x a un nuevo valor k .

Operaciones

HEAP-MAXIMUM(A)

```
1 return  $A[1]$ 
```

HEAP-EXTRACT-MAX(A)

```
1 if  $A.heap-size < 1$ 
2   error "heap underflow"
3  $max = A[1]$ 
4  $A[1] = A[A.heap-size]$ 
5  $A.heap-size = A.heap-size - 1$ 
6 MAX-HEAPIFY( $A, 1$ )
7 return  $max$ 
```

HEAP-INCREASE-KEY(A, i, key)

```
1 if  $key < A[i]$ 
2   error "new key is smaller than current key"
3  $A[i] = key$ 
4 while  $i > 1$  and  $A[PARENT(i)] < A[i]$ 
5   exchange  $A[i]$  with  $A[PARENT(i)]$ 
6    $i = PARENT(i)$ 
```

MAX-HEAP-INSERT(A, key)

```
1  $A.heap-size = A.heap-size + 1$ 
2  $A[A.heap-size] = -\infty$ 
3 HEAP-INCREASE-KEY( $A, A.heap-size, key$ )
```

Agenda

- 1 TADS
- 2 Montón binario
- 3 Cola de prioridad
- 4 Aspectos finales
Ejercicios

Ejercicios

1. Ilustre el paso a paso de heapsort sobre el arreglo:
 $A = [5, 13, 2, 25, 7, 17, 20, 8, 4]$
2. Ilustre el paso a paso de heap_extract_max sobre el heap:
 $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$
3. Ilustre el paso a paso de max_heap_insert(10) sobre el heap:
 $A = [15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1]$
4. Implemente el código para las siguientes operaciones sobre un min-heap:
 - heap_mínimum
 - heap_extract_min
 - heap_decrease_key
 - min_heap_insert
5. Desarrolle un algoritmo para determinar si un árbol binario es un max-heap:
 - **Entrada:** Arbol binario

```
class TreeNode:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

class BinaryTree:
    def __init__(self, root=None):
        self.root = root
```