Algoritmos y estructuras de datos

TAD. Montón binario. Cola de prioridad.

CEIS

Escuela Colombiana de Ingeniería

2024-2

Agenda

Agenda

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

Agenda

Montón binario

Heap

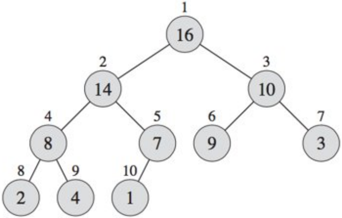
La estructura de datos *heap* es un objeto similar a un arreglo, que se puede ver como **un árbol binario casi completo**. Cada nodo del árbol corresponde a un elemento del arreglo.

.

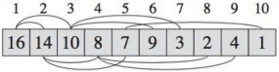
Un array `A` que representa un *heap*, es un objeto con dos atributos:

- `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*.



(a)



$0 \leq A.heap\_size \leq A.length$

(b)

Montón binario

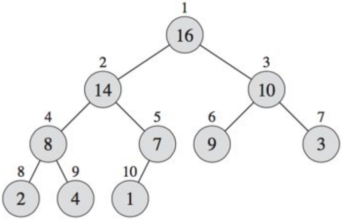
Heap

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

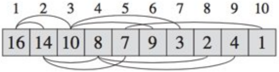
- La altura de un nodo en el *heap*, como el número de arcos en la ruta hacia abajo más larga del nodo a una hoja
- La altura del *heap* como la altura de la raíz

  La altura del *heap* es $\Theta(logn)$

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



(a)



$0 \leq A.heap\_size \leq A.length$

(b)

Montón binario

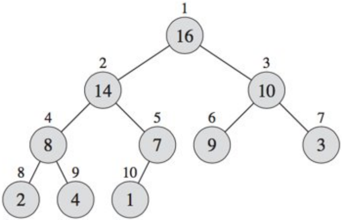
- La raíz del árbol es `A[1]`
- Dado el índice `i` de un nodo, se puede determinar el padre, el hijo izquierdo y el hijo derecho.

PARENT(i)
    return $\lfloor i/2 \rfloor$

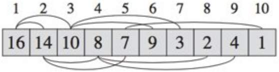
LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

Heap



(a)



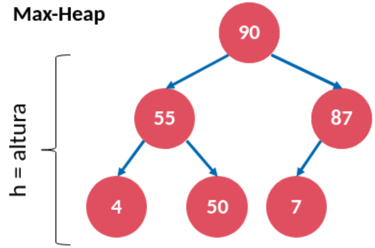
$0 \leq A.heap\_size \leq A.length$

(b)

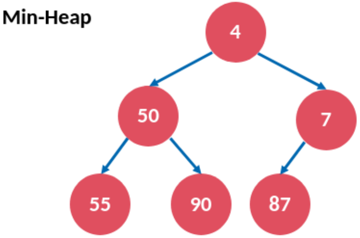
Montón binario

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[PARENT(i)] >= A[i]`
- *Min-Heap*: Para todo nodo `i` diferente a la raíz `A[PARENT(i)] <= A[i]`



$$A[parent(i)] \geq A[i]$$



$$A[parent(i)] \leq A[i]$$

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)

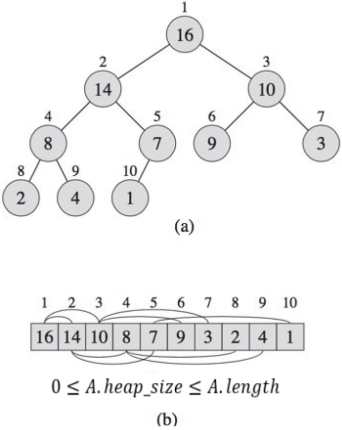
- `HEAPSORT` , ordena un array en tiempo

  O(nlogn)

- `MAX-HEAP-INSERT`
- `HEAP-EXTRACT-MAX`
- `HEAP-INCREASE-KEY`
- `HEAP-MAXIMUM`, permiten a un *heap* implementar una cola de pioridad.

  O(nlogn)

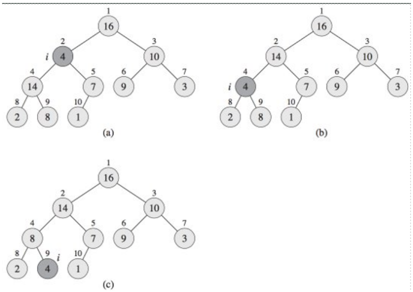
Max-Heap



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 `LEFT(i)` y `RIGHT(i)` son *Max-Heaps*, pero `A[i]` puede ser menor que sus hijos, violando la condición.

```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
      largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
      largest = r
if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```

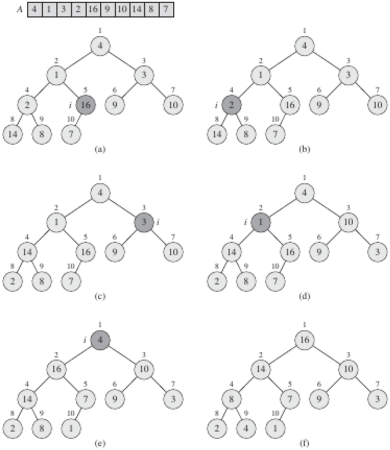


Montón binario. `BUILD-MAX-HEAP`.

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.

BUILD-MAX-HEAP(A)

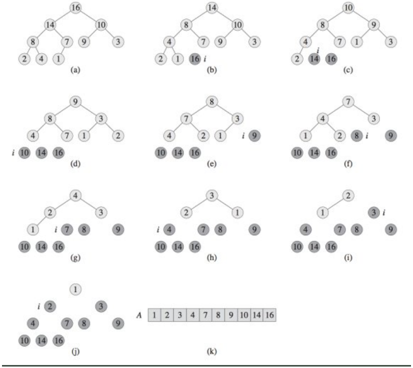
$A.\mathit{heap\text{-}size} = A.\mathit{length}$
for $i = \lfloor A.\mathit{length}/2 \rfloor$ **downto** 1
    MAX-HEAPIFY(A, i)



Montón binario. HEAPSORT

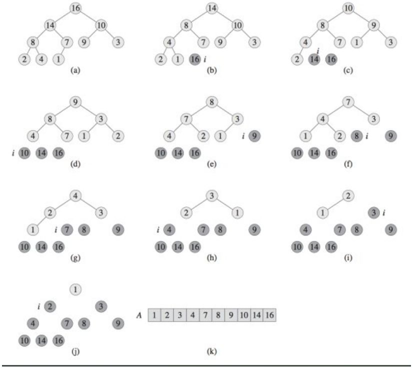
```
HEAPSORT(A)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size − 1
        MAX-HEAPIFY(A, 1)
```

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)
    BUILD-MAX-HEAP(A)
    for i = A.length downto 2
        exchange A[1] with A[i]
        A.heap-size = A.heap-size − 1
        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

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

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

Operaciones

```
HEAP-MAXIMUM(A)
   return A[1]
```

```
HEAP-EXTRACT-MAX(A)
   if A.heap-size < 1
       error "heap underflow"
   max = A[1]
   A[1] = A[A.heap-size]
   A.heap-size = A.heap-size - 1
   MAX-HEAPIFY(A, 1)
   return max
```

```
HEAP-INCREASE-KEY(A, i, key)
   if key < A[i]
       error "new key is smaller than current key"
   A[i] = key
   while i > 1 and A[PARENT(i)] < A[i]
       exchange A[i] with A[PARENT(i)]
       i = PARENT(i)
```

```
MAX-HEAP-INSERT(A, key)
   A.heap-size = A.heap-size + 1
   A[A.heap-size] = -∞
   HEAP-INCREASE-KEY(A, A.heap-size, key)
```

Agenda

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

```python
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
```