Algoritmos y Estructuras de Datos

Colas de prioridad

Colas de prioridad con heaps

Colas de prioridad sobre arreglos

Algoritmos Heapify y Heapsort

Descripción del problema

Queremos implementar un tipo de datos (cola de prioridad) con las siguientes operaciones:

- Crear una nueva cola de prioridad.
- **Insertar** un elemento, indicando un nivel de *prioridad*.
- **Buscar** el elemento de mayor prioridad.
  (O alguno de ellos si hay empates).
- **Eliminar** el elemento de mayor prioridad.

¿Qué complejidades obtendríamos con las siguientes estructuras?

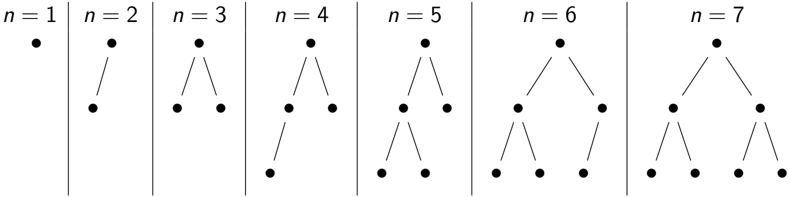
	Inserción	**Búsqueda**	**Eliminación**
Lista	$O(1)$	$O(n)$	$O(n)$
Lista ordenada	$O(n)$	$O(1)$	$O(1)$

Árboles izquierdistas

Un árbol binario es **izquierdista** si:

- Es balanceado y completo, excepto quizá por el último nivel.
  (El último nivel puede estar completo o incompleto).
- Cada vez que hay un nodo en el último nivel,
  están también todos los nodos a su izquierda.

Ejemplo — árboles izquierdistas de n nodos



Observación

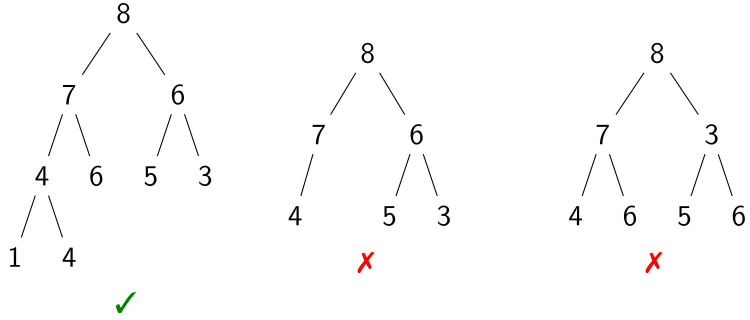
La altura de un árbol izquierdista de n nodos es $O(\log n)$.

Heaps

Un **heap** es un árbol binario con el siguiente invariante:

1. El árbol es izquierdista.
2. En todos los subárboles, el elemento de la raíz es máximo.
   (Más precisamente, es de *máxima prioridad*).

Ejemplos



Inserción en un heap

Algoritmo para insertar un elemento x en un heap:

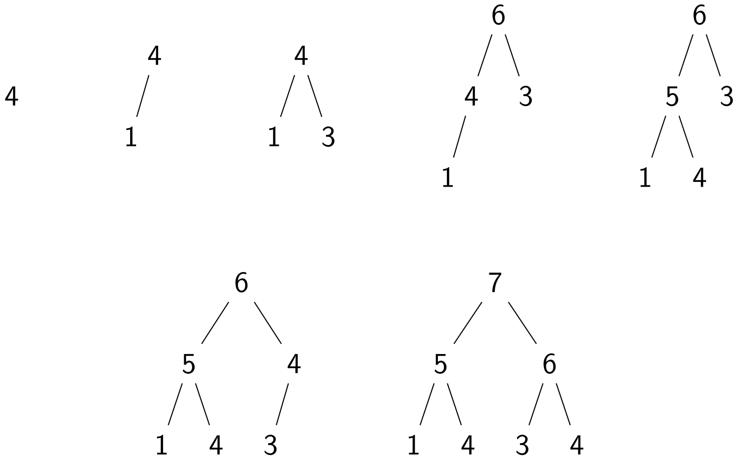
- Ubicar x en la próxima posición libre del árbol izquierdista.
  Es decir, en el último nivel a la derecha.
- Aplicar el siguiente procedimiento de **ajuste hacia arriba**.
  Mientras el elemento tenga mayor prioridad que su padre:
  - Intercambiar el elemento con su padre.

Complejidad temporal en peor caso: $O(\log n)$.

Inserción en un heap

Ejemplo

Insertemos: 4, 1, 3, 6, 5, 4, 7.



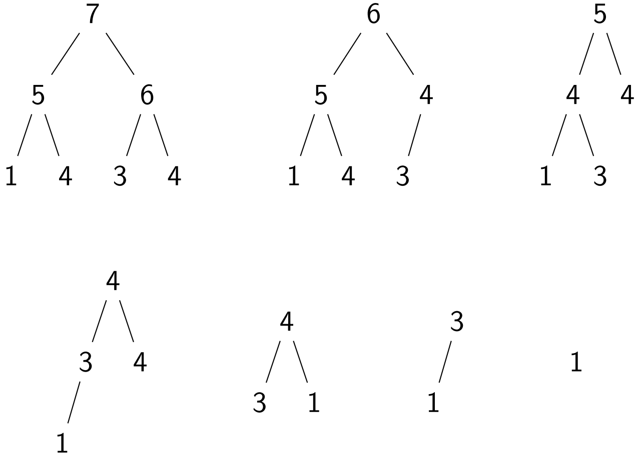
Eliminación del máximo de un heap

Algoritmo para eliminar el máximo elemento de un heap:

- Reemplazar la raíz por el elemento en la última posición.
- Aplicar el siguiente procedimiento de **ajuste hacia abajo**.
  Mientras el elemento sea menor que alguno de sus hijos:
  - Intercambiarlo con el hijo de mayor prioridad.

Complejidad temporal en peor caso: $O(\log n)$.

Eliminación del máximo de un heap



Colas de prioridad con heaps

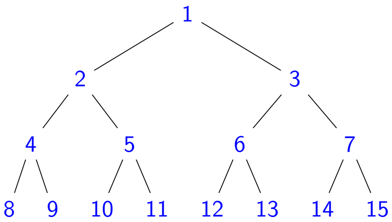
Colas de prioridad sobre arreglos

Algoritmos Heapify y Heapsort

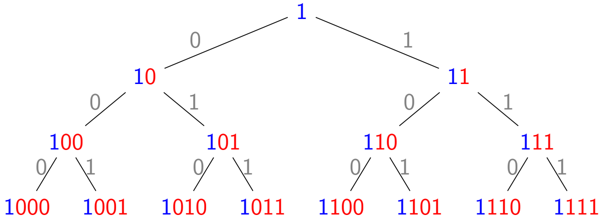
Árboles izquierdistas

Los algoritmos de inserción y eliminación en un heap de tamaño n requieren encontrar la posición del n-ésimo elemento del árbol.

¿Cómo encontramos el n-ésimo nodo en un árbol izquierdista?



La codificación binaria de n indica el camino hacia el nodo:

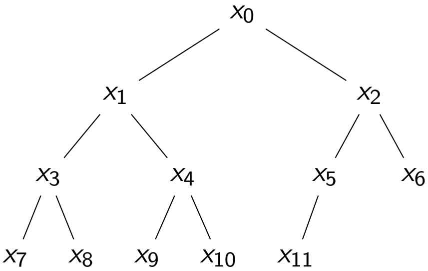


Árboles izquierdistas sobre arreglos

Los árboles izquierdistas se pueden representar como **arreglos**.

Ejemplo

El árbol izquierdista con 12 nodos enumerados de 0 a 11:



se puede representar con el siguiente arreglo de largo 12:

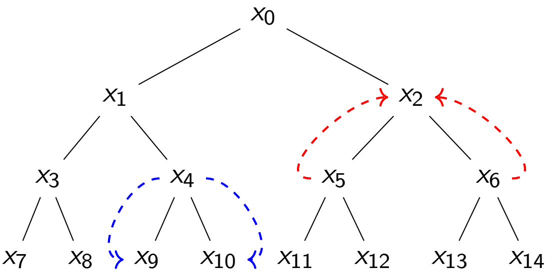
$$[x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}, x_{11}]$$

Árboles izquierdistas sobre arreglos

Dado un índice $0 \leq i < n$ de un árbol izquierdista:

1. El hijo izquierdo se encuentra en el índice $2i + 1$.
2. El hijo derecho se encuentra en el índice $2i + 2$.
3. El padre se encuentra en el índice $\lfloor \frac{i-1}{2} \rfloor$.

Ejemplo



$9 = 2 \cdot 4 + 1$ $\qquad$ $10 = 2 \cdot 4 + 2$ $\qquad$ $2 = \lfloor \frac{5-1}{2} \rfloor$ $\qquad$ $2 = \lfloor \frac{6-1}{2} \rfloor$

Cola de prioridad sobre arreglos en Python

Operaciones auxiliares

```
def esRaiz(i):
    return i == 0

def enRango(heap, i):
    return 0 <= i < len(heap)

def hijoIzq(i):
    return 2 * i + 1

def hijoDer(i):
    return 2 * i + 2

def padre(i):
    return (i - 1) // 2
```

Cola de prioridad sobre arreglos en Python

Algoritmo de inserción

```python
def insertar(heap, x):
    heap.append(x)
    ajustarHaciaArriba(heap, len(heap) - 1)

def ajustarHaciaArriba(heap, i):
    while not esRaiz(i) and heap[i] > heap[padre(i)]:
        heap[i], heap[padre(i)] = heap[padre(i)], heap[i]
        i = padre(i)
```

Cola de prioridad sobre arreglos en Python

Algoritmo de eliminación del máximo

```
def eliminarMaximo(heap):
    ultimo = heap.pop()
    if len(heap) == 0:
        return
    heap[0] = ultimo
    ajustarHaciaAbajo(heap, 0)

def ajustarHaciaAbajo(heap, i):
    while tieneHijoMayor(heap, i):
        j = indiceDelHijoMayor(heap, i)
        heap[i], heap[j] = heap[j], heap[i]
        i = j
```

Cola de prioridad sobre arreglos en Python

Algoritmo de eliminación del máximo — operaciones auxiliares

```
def tieneHijoMayor(heap, i):
    izq = hijoIzq(i)
    der = hijoDer(i)
    return (enRango(heap, izq) and heap[izq] > heap[i]) \
        or (enRango(heap, der) and heap[der] > heap[i])

def indiceDelHijoMayor(heap, i):
    # Precondición: tieneHijoMayor(heap, i)
    izq = hijoIzq(i)
    der = hijoDer(i)
    if enRango(heap, der) and heap[der] > heap[izq]:
        return der
    else:
        return izq
```

Colas de prioridad con heaps

Colas de prioridad sobre arreglos

Algoritmos Heapify y Heapsort

Heapify de Floyd

Supongamos que tenemos un arreglo de n elementos.
¿Cuál es el costo de construir un heap con dichos elementos?

Si hacemos n inserciones, el costo es $O(n \log n)$.

Se puede hacer de manera más eficiente con el algoritmo $\textsc{Heapify}$.

Heapify de Floyd

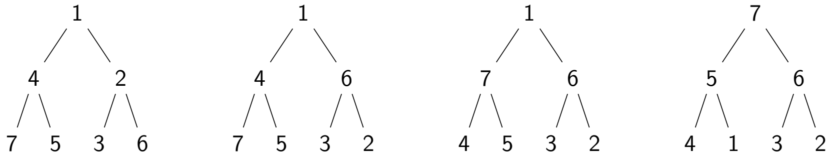
Entrada: un arreglo A de n elementos.
Salida: un heap que contiene a los elementos de A.

Método.

- Para cada índice i desde $n-1$ hasta 0: (descendentemente)
  - Aplicar el algoritmo de **ajuste hacia abajo** a partir de i.

El método es *in-place*.

Ejemplo



La complejidad temporal es $O(n)$ en peor caso.
(*Cf.* Sec. 6.3 del Cormen *et al.*).

Heapsort

La estructura de datos nos da un nuevo algoritmo de ordenamiento.

Algoritmo Heapsort

Entrada: un arreglo A.

Salida: una permutación ordenada de A.

Método.

- Heapify(A)
- Repetir $|A|$ veces:
  - Producir el máximo elemento de A en la salida.
  - Eliminar el máximo elemento de A.

La complejidad temporal en peor caso es $O(n \log n)$.

Se puede hacer *in-place*.