Heaps

Algoritmos y Estructuras de Datos

2^{do} cuatrimestre de 2024

Motivación

Implementar una cola de prioridad.

Motivación

Implementar una cola de prioridad.

Una cola de prioridad es un contenedor de objetos que nos permite siempre sacar el máximo de estos según alguna *relación de orden total*.

Motivación

Implementar una cola de prioridad.

Una cola de prioridad es un contenedor de objetos que nos permite siempre sacar el máximo de estos según alguna *relación de orden total*.
Operaciones necesarias para una cola de prioridad:

Máximo	Determinar el elemento más prioritario.
Agregar	Agregar un elemento.
Sacar máximo	Sacar el elemento más prioritario.
Conj→cola	Convertir un conjunto en una cola de prioridad.

Hablamos siempre del **máximo**.

Posibles implementaciones (sin usar heap)

	Máximo	**Agregar**	**Sacar máximo**	**Conj→cola**
Lista	?	?	?	?
Lista + máximo	?	?	?	?
Lista ordenada	?	?	?	?
AVL + máximo	?	?	?	?

Posibles implementaciones (sin usar heap)

	Máximo	**Agregar**	**Sacar máximo**	**Conj→cola**
Lista	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Lista + máximo	?	?	?	?
Lista ordenada	?	?	?	?
AVL + máximo	?	?	?	?

Posibles implementaciones (sin usar heap)

	Máximo	**Agregar**	**Sacar máximo**	**Conj→cola**
Lista	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Lista + máximo	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Lista ordenada	?	?	?	?
AVL + máximo	?	?	?	?

Posibles implementaciones (sin usar heap)

	Máximo	**Agregar**	**Sacar máximo**	**Conj→cola**
Lista	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Lista + máximo	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Lista ordenada	$O(1)$	$O(n)$	$O(1)$	(sorting)
AVL + máximo	?	?	?	?

Posibles implementaciones (sin usar heap)

	Máximo	**Agregar**	**Sacar máximo**	**Conj→cola**
Lista	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Lista + máximo	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Lista ordenada	$O(1)$	$O(n)$	$O(1)$	(sorting)
AVL + máximo	$O(1)$	$O(\log n)$	$O(\log n)$	$O(n \log n)$

Spoiler

¿Para qué queremos un heap si podemos usar un AVL?

Spoiler

¿Para qué queremos un heap si podemos usar un AVL?

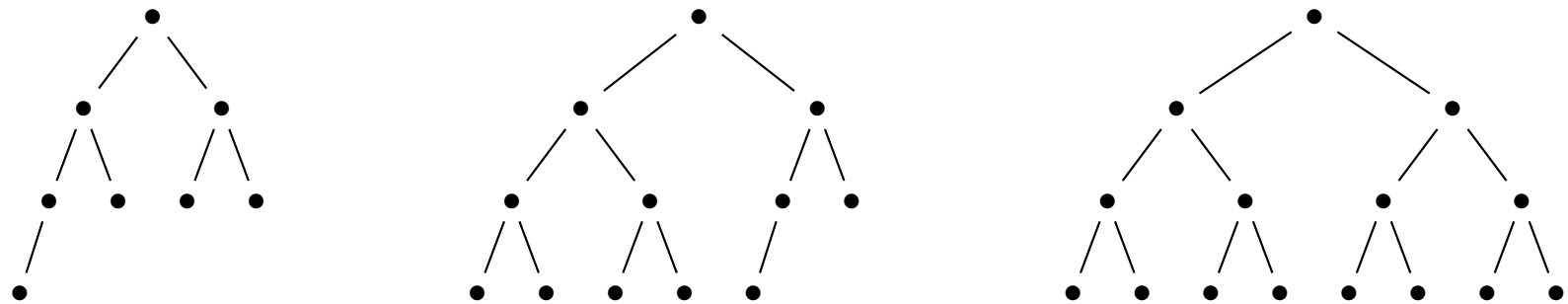
- Más sencillo de implementar.
- Mejores constantes.
- Se puede hacer sin punteros.
- La operación **Conj**$\rightarrow$**cola** es estrictamente mejor.

Heap: invariante

Un heap es un árbol binario con un invariante:

Forma

- Completo, salvo por el último nivel.
- Izquierdista.



Orden

- La raíz es el máximo.
- El invariante se cumple recursivamente para los hijos.
- (yapa) Todos los caminos de la raíz a una hoja son secuencias ordenadas.

Heap: algoritmos

Máximo $O(1)$

- Está en la raíz del árbol.

Agregar $O(\log n)$

- Ubicar el elemento respetando la forma del heap.
- Mientras sea mayor que su padre, intercambiarlo con el padre. (*Sift up*).

Sacar máximo $O(\log n)$

- Reemplazar la raíz del árbol por el "último" elemento, respetando la forma del heap.
- Mientras sea menor que uno de sus hijos, intercambiarlo con el mayor de sus hijos.
(*Sift down*).

Heap: algoritmos

Ejemplo: insertar en secuencia $6, 4, 2, 9, 3, 8, 5$ y sacar el máximo.

Heap: algoritmos

Conj→cola (*heapify*) $O(n)$

- Armar un árbol con los elementos respetando la forma.
- Hacer *Sift down* para cada uno de los elementos, yendo "hacia atrás", desde el último hasta la raíz.

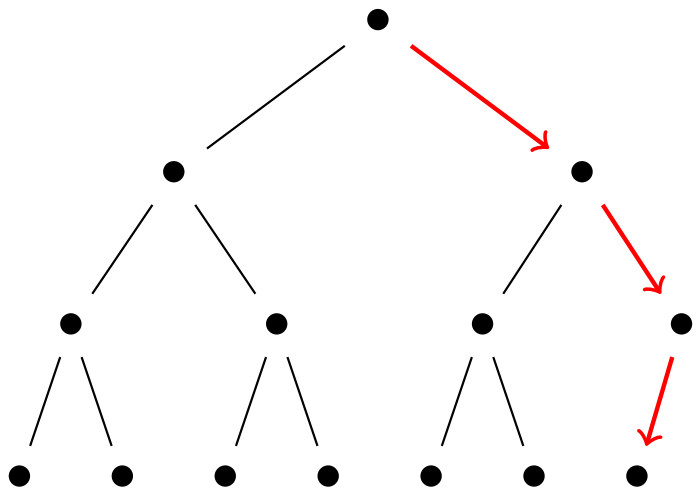
Ejemplo: *heapificar* la secuencia $6, 4, 2, 9, 3, 8, 5$.

Heap: técnicas de implementación

Técnica de implementación con referencias (punteros)

Si el heap tiene n elementos, la posición del último se puede encontrar a partir de la representación en binario de n, ignorando el dígito 1 más significativo.

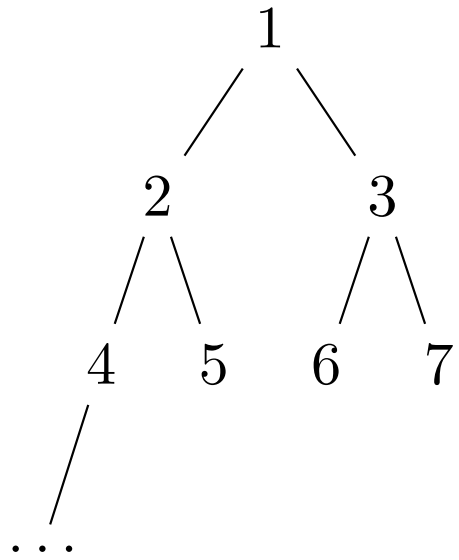
$n = 14 = (1110)_2 \quad \rightsquigarrow \quad$ [derecha, derecha, izquierda]



Heap: técnicas de implementación

¿Y por qué funciona esta técnica?

Enumeremos los nodos del árbol en el orden de recorrido por niveles:



Al usar referencias, los nodos se numeran empezando en 1.

Heap: técnicas de implementación

Técnica de implementación con arreglos

Los elementos se pueden guardar en un arreglo de tamaño N.

Las siguientes funciones sirven para navegar el árbol:

$$\begin{aligned} \texttt{HIJO\_IZQ}(i) &= 2 * i + 1 \\ \texttt{HIJO\_DER}(i) &= 2 * i + 2 \\ \texttt{PADRE}(i) &= \lfloor \tfrac{i-1}{2} \rfloor \end{aligned}$$

Usando índices $0 \leq i < N$.

Heap: técnicas de implementación

Observemos que:

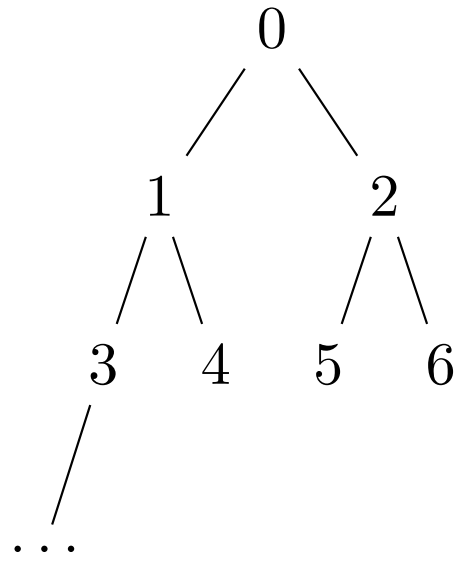
- Si el nivel n está completo, consta de exactamente 2^n nodos.
- Si los primeros n niveles están completos, constan de exactamente $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ nodos.
- El i-ésimo nodo del j-ésimo nivel está en la posición $k = 2^j - 1 + i$ en el recorrido por niveles.
- Sus hijos están en las posiciones $2i$ y $2i + 1$ del $(j + 1)$-ésimo nivel, es decir:

  Hijo izquierdo: $2^{j+1} - 1 + 2i = 2k + 1$
  Hijo derecho: $2^{j+1} - 1 + 2i + 1 = 2k + 2$

Heap: técnicas de implementación

¿Y por qué funciona esta técnica?

Enumeremos los nodos del árbol en el orden de recorrido por niveles:

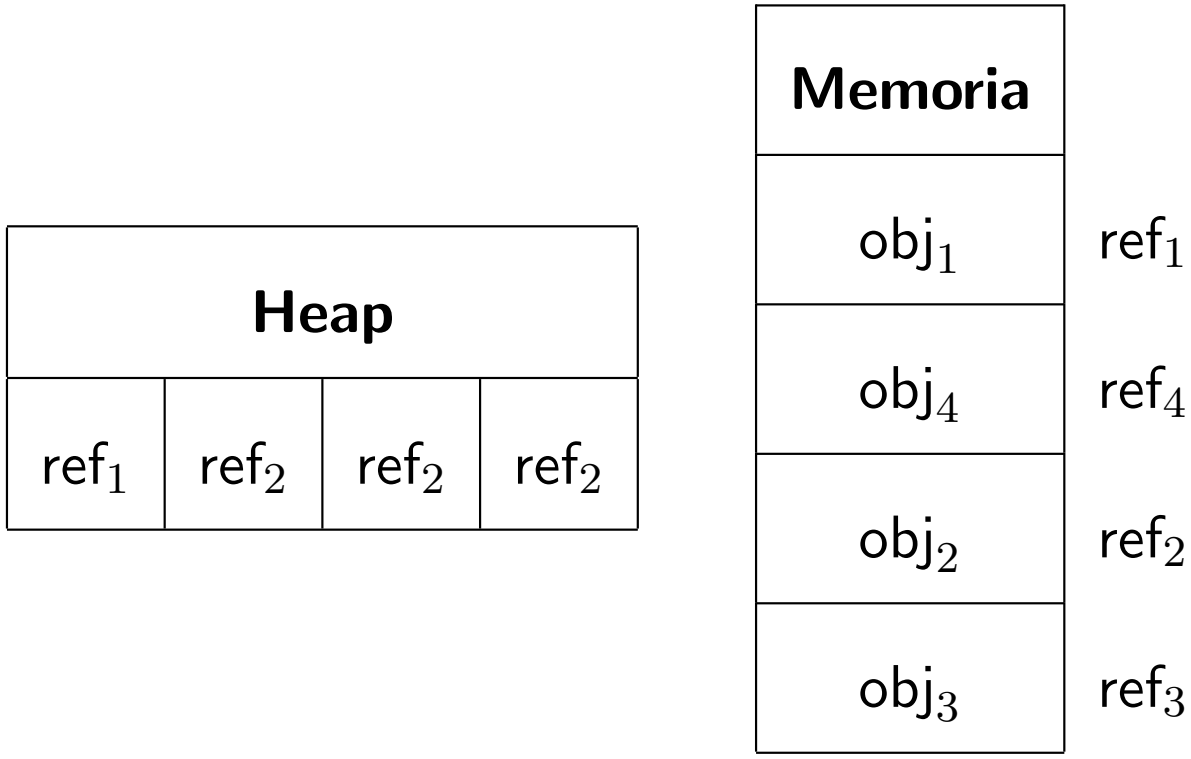


Usando un arreglo, tanto los niveles como los nodos se numeran empezando en 0.

Heap: técnicas de implementación

Podremos crear un heap para cualquier tipo de clase mientras tengamos una relacion de orden total que los ordene.

El Heap contendra referencias a estos objetos, que reordenaremos sin tener que trasladar el objeto guardado en memoria.



Heap: técnicas de implementación

En Java, sobreescribiendo el método *compareTo* podemos definir sobre que atributo/s nos basamos para ordenar estos objetos.

Pero ¿Que pasa si se modifican los valores internos de estos objetos que estamos referenciando mientras aún están en el heap?

Heap: técnicas de implementación

En Java, sobreescribiendo el método *compareTo* podemos definir sobre que atributo/s nos basamos para ordenar estos objetos.

Pero ¿Que pasa si se modifican los valores internos de estos objetos que estamos referenciando mientras aún están en el heap? ¡Podría **romperse** la invariante del mismo si no lo cuidamos!

Heap: técnicas de implementación

Para que no suceda esto, necesitamos implementar dos cosas:

- *handles* para cada objeto en el heap, una referencia o indice.
- un método que dado un *handle* nos permita o modificar o simplemente revisar la ubicación de un objeto.

Los handles nos permiten acceder a la ubicación de un objeto dentro del Heap en $O(1)$.

Una forma simple de implementar esto es que dentro del Heap el *handle* se guarde junto al objeto y que al agregar un objeto nuevo al Heap, se te devuelva el handle del mismo.

Comparator: Motivación

Supongamos que tenemos la siguiente clase **CaballoDeCarrera** con los atributos:

- int carrerasGanadas;
- float velocidadPromedio;
- int Edad;
- String Nombre

¿Cuantos relaciones de orden podemos crear para esta clase?

Comparator: Motivación

Supongamos que tenemos la siguiente clase **CaballoDeCarrera** con las clases

- int carrerasGanadas;
- float velocidadPromedio;
- int Edad;
- String Nombre

¿Cuantos relaciones de orden podemos crear para esta clase?

- Mayor cantidad de carreras ganadas
- Mayor velocidad promedio
- Mayor edad.
- etc

Todas tienen su uso

Comparator

¿Habra que implementar un nuevo heap por cada relacion de orden que querramos seguir?

Comparator

¿Habra que implementar un nuevo heap por cada relacion de orden que querramos seguir?

NO!!!

Una solución es que Heap tome un **número de prioridad** junto con el objeto y que guarde y utilize este para ordenarlo. Pero en Java, podemos abstraer la relacion de orden utilizando *Comparator*

Comparator

La interfaz Comparator definira un metodo **compare**(arg1, arg2) que representara la relacion de orden que querremos definir.

$$\textbf{compare}(arg1, arg2) = \begin{cases} < 0 & \text{si } arg1 < arg2 \\ 0 & \text{si } arg1 = arg2 \\ > 0 & \text{si } arg1 > arg2 \end{cases} \quad (1)$$

Comparator: implementacion

Si quisieramos construir un comparador por **Carreras Ganadas**, definiriamos:

```
public class CarrerasComparator implements Comparator
<CaballoDeCarrera>{
   @Override
   public int compare(CaballoDeCarrera cab1, CaballoDeCarrera cab2){
      return Integer.compare(cab1.carrerasGanadas,
cab2.carrerasGanadas);
   }
}
```

Podemos escribir esto mucho mas corto haciendo:

```
Comparator<CaballoDeCarrera> CarrerasComparator =
Comparator.comparing(CaballoDeCarrera::carrerasGanadas);
```

Lo cual crea una instancia de Comparator equivalente.

Comparator: Fin

Ahora nuestro constructor de Heap recibira un Comparator, pudiendo usar una misma implementacion para varias relaciones de orden.