Heaps

Algoritmos y Estructuras de Datos

1^{er} cuatrimestre de 2025

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?

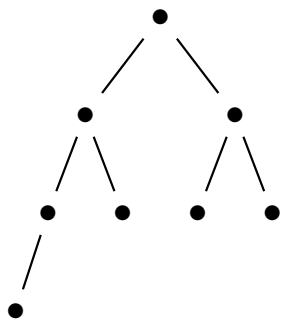
- Más sencillo de implementar.
- Mejores constantes.
- Se puede hacer sin punteros.
- La operación **Conj**$\rightarrow$**cola** es estrictamente mejor.

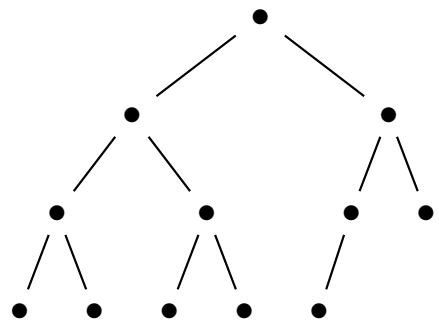
Heap: invariante

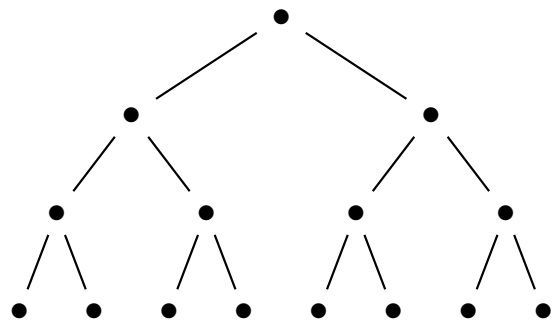
Un heap es un árbol binario con un invariante:

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* por cada uno de los niveles, yendo "hacia atrás", hasta la raíz.

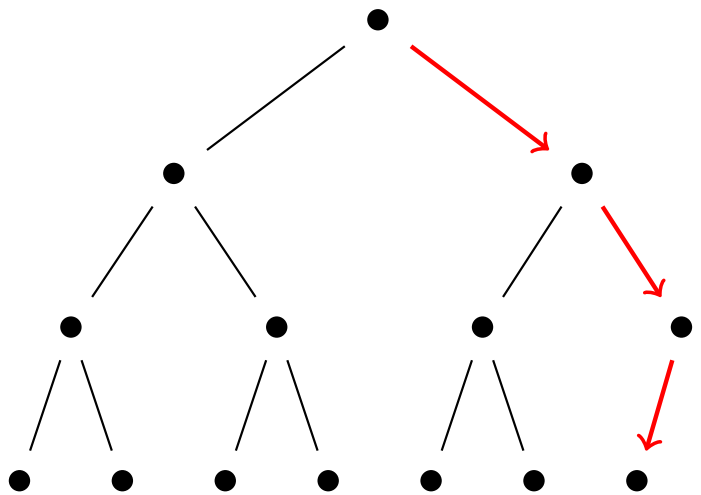
Ejemplo: *heapificar* la secuencia $6, 4, 2, 9, 3, 8, 5$.

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.

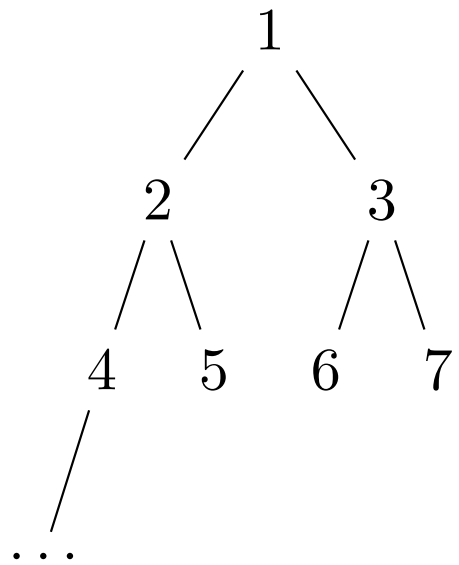
$n = 14 = (1110)_2 \quad \rightsquigarrow \quad$ [derecha, derecha, izquierda]



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:

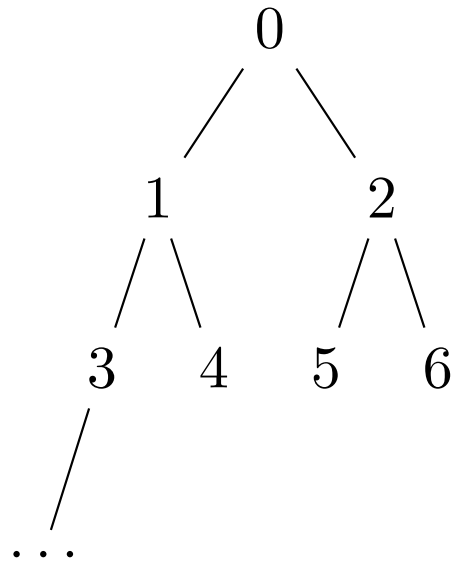
- Si el nivel n está completo, consta de exactamente 2^n nodos.
- Si los primeros n niveles están completos, constan de exactamente $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ nodos.
- El i-ésimo nodo del j-ésimo nivel está en la posición $k = 2^j - 1 + i$ en el recorrido por niveles.
- Sus hijos están en las posiciones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

¿Y cómo hacemos heaps de otras cosas?

Deberíamos poder crear un heap para cualquier tipo de clase **siempre que tengamos una relación de orden total** en las intancias de la clase.

Heap: técnicas de implementación

¿Y cómo hacemos heaps de otras cosas?

Deberíamos poder crear un heap para cualquier tipo de clase **siempre que tengamos una relación de orden total** en las intancias de la clase.

Dos ideas:

Heap: técnicas de implementación

¿Y cómo hacemos heaps de otras cosas?

Deberíamos poder crear un heap para cualquier tipo de clase **siempre que tengamos una relación de orden total** en las intancias de la clase.

Dos ideas:

- Sobreescribiendo el método *compareTo*, en Java podemos definir sobre que atributo/s de la clase nos basamos para ordenar los objetos.

Heap: técnicas de implementación

¿Y cómo hacemos heaps de otras cosas?

Deberíamos poder crear un heap para cualquier tipo de clase **siempre que tengamos una relación de orden total** en las intancias de la clase.

Dos ideas:

- Sobreescribiendo el método *compareTo*, en Java podemos definir sobre que atributo/s de la clase nos basamos para ordenar los objetos.
- Armando un Heap con las referencias de los objetos, podemos reordenarlos sin tener que mover todo el objeto, que esta guardado en memoria.

Heap: técnicas de implementación

- Sobreescribiendo el método *compareTo*, en Java podemos definir sobre qué atributo/s de la clase nos basamos para ordenar los objetos.

Heap: técnicas de implementación

- Sobreescribiendo el método *compareTo*, en Java podemos definir sobre qué atributo/s de la clase nos basamos para ordenar los objetos.

Supongamos que tenemos la siguiente clase:

```
public class CaballoDeCarrera {
    float velocidadPromedio;
    int carrerasGanadas;
    int edad;
    String nombre;

    ...
}
```

Heap: técnicas de implementación

- Sobreescribiendo el método *compareTo*, en Java podemos definir sobre qué atributo/s de la clase nos basamos para ordenar los objetos.

Supongamos que tenemos la siguiente clase:

```
public class CaballoDeCarrera {
    float velocidadPromedio;
    int carrerasGanadas;
    int edad;
    String nombre;

    ...
}
```

¿Cuántos ordenes puede haber?

Heap: técnicas de implementación

- Sobreescribiendo el método *compareTo*, en Java podemos definir sobre qué atributo/s de la clase nos basamos para ordenar los objetos.

Supongamos que tenemos la siguiente clase:

```
public class CaballoDeCarrera {
    float velocidadPromedio;
    int carrerasGanadas;
    int edad;
    String nombre;

    ...
}
```

¿Cuántos ordenes puede haber?

- Mayor velocidad promedio

Heap: técnicas de implementación

- Sobreescribiendo el método *compareTo*, en Java podemos definir sobre qué atributo/s de la clase nos basamos para ordenar los objetos.

Supongamos que tenemos la siguiente clase:

```
public class CaballoDeCarrera {
    float velocidadPromedio;
    int carrerasGanadas;
    int edad;
    String nombre;

    ...
}
```

¿Cuántos ordenes puede haber?

- Mayor velocidad promedio
- Más carreras ganadas

Heap: técnicas de implementación

- Sobreescribiendo el método *compareTo*, en Java podemos definir sobre qué atributo/s de la clase nos basamos para ordenar los objetos.

Supongamos que tenemos la siguiente clase:

```
public class CaballoDeCarrera {
    float velocidadPromedio;
    int carrerasGanadas;
    int edad;
    String nombre;

    ...
}
```

¿Cuántos ordenes puede haber?

- Mayor velocidad promedio
- Más carreras ganadas
- Mayor edad.

Heap: técnicas de implementación

- Sobreescribiendo el método *compareTo*, en Java podemos definir sobre qué atributo/s de la clase nos basamos para ordenar los objetos.

Supongamos que tenemos la siguiente clase:

```
public class CaballoDeCarrera {
    float velocidadPromedio;
    int carrerasGanadas;
    int edad;
    String nombre;

    ...
}
```

¿Cuántos ordenes puede haber?

- Mayor velocidad promedio
- Más carreras ganadas
- Mayor edad.
- etc

Todas tienen su uso

Heap: técnicas de implementación

Ejemplo

```
public class CaballoDeCarrera {
    ...
    @Override
    public int compareTo(CaballoDeCarrera otro){
       if(otro == null){
         String mensajeDeError = "No puede compararse con null";
         throw new IllegalArgumentException(mensajeDeError)
       }
       return otro.carrerasGanadas - this.carrerasGanadas;;
    }
}
```

Heap: técnicas de implementación

Ejemplo

```
public class CaballoDeCarrera {
    ...
    @Override
    public int compareTo(CaballoDeCarrera otro){
       if(otro == null){
         String mensajeDeError = "No puede compararse con null";
         throw new IllegalArgumentException(mensajeDeError)
       }
       return otro.carrerasGanadas - this.carrerasGanadas;;
    }
}
```

- Con esta implementacion es resposabilidad de la clase definir qué orden va a usarse en el heap y no podemos usar otros criterios.

Heap: técnicas de implementación

Ejemplo

```
public class CaballoDeCarrera {
    ...
    @Override
    public int compareTo(CaballoDeCarrera otro){
       if(otro == null){
         String mensajeDeError = "No puede compararse con null";
         throw new IllegalArgumentException(mensajeDeError)
       }
       return otro.carrerasGanadas - this.carrerasGanadas;;
    }
}
```

- Con esta implementacion es resposabilidad de la clase definir qué orden va a usarse en el heap y no podemos usar otros criterios.
- Una alternativa seria abstraer la relación de orden utilizando una interfaz *Comparator*

Heap: técnicas de implementación

- Armando un Heap con las referencias de los objetos, podemos reordenarlos sin tener que mover todo el objeto, que está guardado en memoria.

Heap			
ref_1	ref_2	ref_3	ref_4

posicion	**Memoria**
ref_3	obj_3
	...
ref_4	obj_4
ref_2	obj_2
ref_1	obj_1

Heap: técnicas de implementación

- Armando un Heap con las referencias de los objetos, podemos reordenarlos sin tener que mover todo el objeto, que está guardado en memoria.

Heap			
ref_1	ref_2	ref_3	ref_4

posicion	**Memoria**
ref_3	obj_3
	...
ref_4	obj_4
ref_2	obj_2
ref_1	obj_1

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

Heap: técnicas de implementación

- Armando un Heap con las referencias de los objetos, podemos reordenarlos sin tener que mover todo el objeto, que está guardado en memoria.

Heap			
ref_1	ref_2	ref_3	ref_4

posicion	**Memoria**
ref_3	obj_3
	...
ref_4	obj_4
ref_2	obj_2
ref_1	obj_1

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

¡Podría romperse la invariante de orden que tiene que satisfacer!

Heap: técnicas de implementación

Para que no suceda esto, necesitamos implementar dos cosas:

- Un *Handle* para cada objeto en el Heap, con una referencia o índice.
- Un método que dado un *Handle* nos permita modificar la ubicación de un objeto.

Handle: recordemos

Los *Handles* son **referencias abstractas** a un recurso manejado por otro sistema.

Handle: recordemos

Los *Handles* son **referencias abstractas** a un recurso manejado por otro sistema.

Los implementaremos como un objeto que, en su representación interna, guarda un puntero al Nodo de interés. Y nos permiten acceder a la ubicación de un objeto dentro del Heap en $O(1)$.

Handle: recordemos

Los *Handles* son **referencias abstractas** a un recurso manejado por otro sistema.

Los implementaremos como un objeto que, en su representación interna, guarda un puntero al Nodo de interés. Y nos permiten acceder a la ubicación de un objeto dentro del Heap en $O(1)$.

Una vez que tenemos el Handle podemos usar el método para modificarlo y luego reordenar el heap.

Handle: recordemos

Los *Handles* son **referencias abstractas** a un recurso manejado por otro sistema.

Los implementaremos como un objeto que, en su representación interna, guarda un puntero al Nodo de interés. Y nos permiten acceder a la ubicación de un objeto dentro del Heap en $O(1)$.

Una vez que tenemos el Handle podemos usar el método para modificarlo y luego reordenar el heap.

¡Pero ojo! **Hay que asegurarnos que cuando se mueve algo en el heap, se modifique el índice del *Handle* correspondiente**.