Programación Avanzada y Estructuras de Datos

7. Cola de prioridad

Víctor M. Sánchez Cartagena

Grado en Ingeniería en Inteligencia Artificial
Dep. Lenguajes y Sistemas Informáticos
Universidad de Alicante

4 de diciembre de 2024

Índice

Ejemplo introductorio

Imagina que queremos gestionar la cola virtual de un sitio web de venta de entradas. Cada petición se codifica en un objeto `Peticion` que guarda el nombre de usuario, la dirección IP, el tipo de usuario (no registrado, registrado, premium, etc.), el gasto total que ha efectuado en el sitio web de venta de entradas, y la hora a la que ha accedido al sitio web. Todos estos criterios deben tenerse en cuenta para ordenar las peticiones en la cola virtual. La clase `Peticion` dispone de un `operator<` que nos indica si la primera petición tiene menos prioridad que la segunda.

Ejemplo introductorio

- Cada vez que un nuevo usuario accede la web de venta de entradas, debe agregarse su petición a la cola virtual
- Cada vez que el servidor cuenta con capacidad de cómputo suficiente, debe sacar de la cola virtual la petición con más prioridad y darle acceso al servicio de venta

Ejemplo introductorio

```
class Peticion{
private:
string nombre_usuario;
int IP;
/*Tipo usuario: 0: sin registrar; 1: registrado; 2: premium*/
int tipo_usuario;
float gasto_total;
time_t hora_acceso;
public:
Peticion();
Peticion(const string &, int, int, float);
Peticion(const Peticion &);
Peticion& operator=(const Peticion&);
~Peticion();
bool operator<(const Peticion& ) const;
//Getters y setters
string getNombreUsuario() const;
//....
};
```

Ejemplo introductorio

Pregunta

¿En qué estructura de datos almacenarías las peticiones para minimizar el coste temporal de: insertar elementos; saber cuál es el elemento con máxima prioridad; sacar de la cola el elemento con máxima prioridad?

Ejemplo introductorio

Pregunta

¿En qué estructura de datos almacenarías las peticiones para minimizar el coste temporal de: insertar elementos; saber cuál es el elemento con máxima prioridad; sacar de la cola el elemento con máxima prioridad?

- Lista enlazada ordenada descendentemente según `operator<`
  - Insertar: $\Omega(1)$; $O(n)$
  - Obtener elemento con máxima prioridad: $O(1)$
  - Borrar elemento con máxima prioridad: $O(1)$
- Árbol AVL ordenado según `operator<`
  - Insertar: $\Theta(\log(n))$
  - Obtener elemento con máxima prioridad: $\Theta(\log(n))$
  - Borrar elemento con máxima prioridad: $\Theta(\log(n))$

Ejemplo introductorio

```
//Añadir una petición a la cola
void add_to_queue(list<Peticion> & q, const Peticion & p){
    bool inserted=false;
    for(auto it= q.begin(); it != q.end() ;++it){
        if(*it < p ){
            //Insert before it
            q.insert(it,p);
            inserted=true;
            break;
        }
    }
    if(! inserted){
        q.push_back(p);
    }
}

list<Peticion> priority_queue;
//Obtener petición con máxima prioridad
cout << priority_queue.front().getNombreUsuario() << endl;
//Sacarla de la cola
priority_queue.pop_front();
```

TAD cola de prioridad

Especificación del TAD cola de prioridad

Definición: Colección de n elementos almacenados sin un orden definido. Cada elemento tiene asociada una prioridad. Sólo se puede elimininar y acceder al elemento con máxima prioridad

TAD cola de prioridad

Especificación del TAD cola de prioridad

Operaciones:

- Obtiene el número de elementos almacenados en la cola

```
int size() const;
```

- Añade el elemento `e` a la cola

```
void insert(const Elem &e);
```

- Devuelve el elemento con máxima prioridad

```
Elem max() const;
```

- Elimina el elemento con máxima prioridad. Devuelve `false` si la cola estaba vacía y `true` en caso contrario

```
bool removeMax();
```

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`		
`insert(9)`		
`insert(2)`		
`max()`		
`removeMax()`		
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`		
`insert(2)`		
`max()`		
`removeMax()`		
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	Salida	Contenido de la cola
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`		
`max()`		
`removeMax()`		
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`		
`removeMax()`		
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`		
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`		
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`		
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	Salida	Contenido de la cola
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`	5	{5,2}
`removeMax()`		
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`	5	{5,2}
`removeMax()`	true	{2}
`removeMax()`		
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`	5	{5,2}
`removeMax()`	true	{2}
`removeMax()`	true	{}
`size()`		
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	Salida	Contenido de la cola
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`	5	{5,2}
`removeMax()`	true	{2}
`removeMax()`	true	{}
`size()`	0	{}
`removeMax()`		

TAD cola de prioridad

Especificación del TAD cola prioridad

Ejemplos (cola inicialmente vacía):

Operación	**Salida**	**Contenido de la cola**
`insert(5)`	-	{5}
`insert(9)`	-	{9,5}
`insert(2)`	-	{9,5,2}
`max()`	9	{9,5,2}
`removeMax()`	true	{5,2}
`size()`	2	{5,2}
`max()`	5	{5,2}
`removeMax()`	true	{2}
`removeMax()`	true	{}
`size()`	0	{}
`removeMax()`	false	{}

Implementación del TAD cola de prioridad

- Lista ordenada: `insert`: $O(n)$, `max`: $O(1)$, `removeMax`: $O(1)$
- Árbol AVL: `insert`: $O(\log n)$, `max`: $O(\log n)$, `removeMax`: $O(\log n)$

Implementación del TAD cola de prioridad

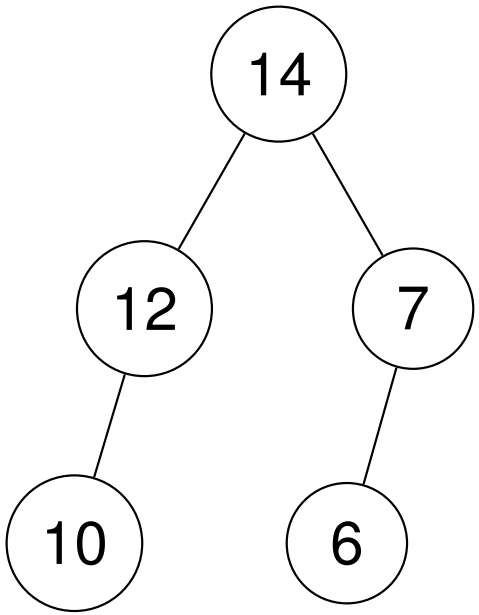
- Lista ordenada: `insert`: $O(n)$, `max`: $O(1)$, `removeMax`: $O(1)$
- Árbol AVL: `insert`: $O(\log n)$, `max`: $O(\log n)$, `removeMax`: $O(\log n)$
- Montículo o *heap*: `insert`: $O(\log n)$, `max`: $O(1)$, `removeMax`: $O(\log n)$

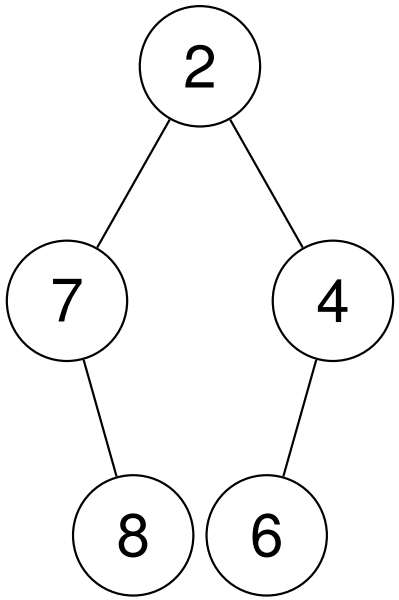
Índice

Heap

Árbol mínimo (o máximo)

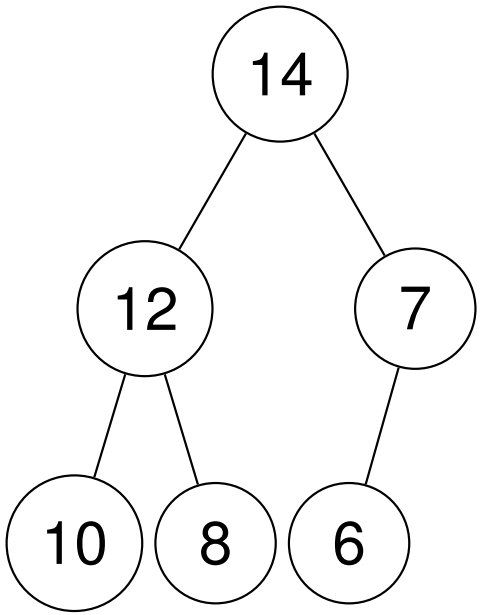
Árbol en el que la etiqueta de cada nodo es menor (o mayor) que la de sus hijos

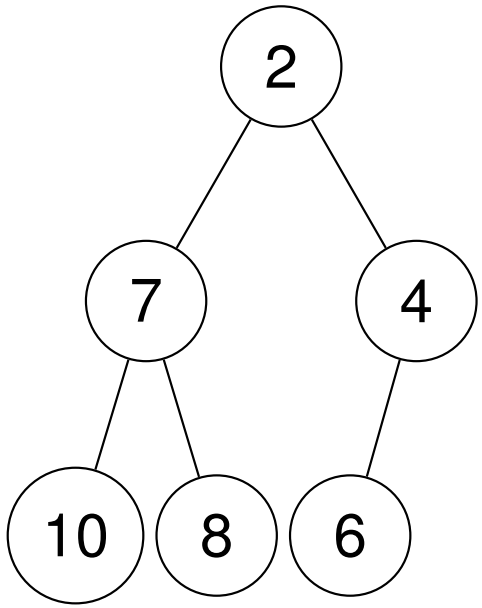




Heap mínimo (o máximo)

Árbol mínimo (o máximo) que además es completo





Heap

Pregunta

¿Dónde está el elemento máximo de un heap máximo?

Heap

Pregunta

¿Dónde está el elemento máximo de un heap máximo?

En la raíz, por eso la complejidad de la operación `max` es $O(1)$

Inserción en un heap

Algoritmo de inserción en un heap:

1. Insertar el elemento en la posición libre de más a la izquierda del último nivel, para que el árbol continúe siendo completo
2. Repetir la siguiente operación mientras que el árbol no cumpla las propiedades de heap:
   - Árbol mínimo: intercambiar el elemento insertado con su padre si el elemento es menor que su padre
   - Árbol máximo: intercambiar el elemento insertado con su padre si el elemento es mayor que su padre

Inserción en un heap

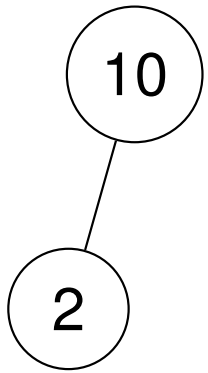
Ejemplo

Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21

Inserción en un heap

Ejemplo

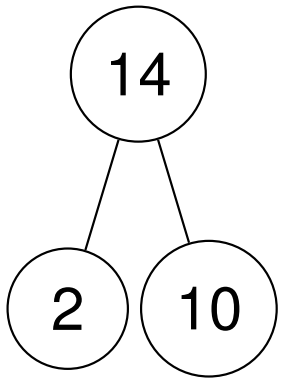
Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21



Inserción en un heap

Ejemplo

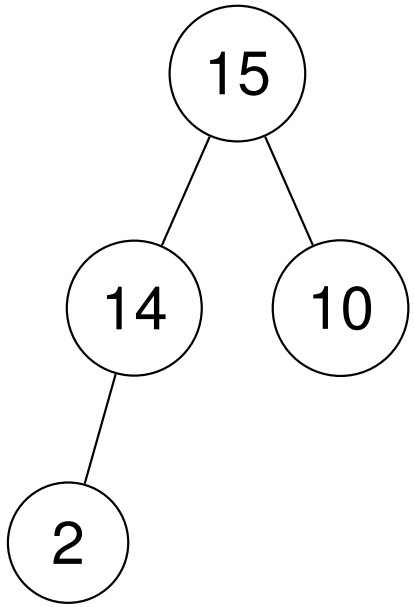
Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21



Inserción en un heap

Ejemplo

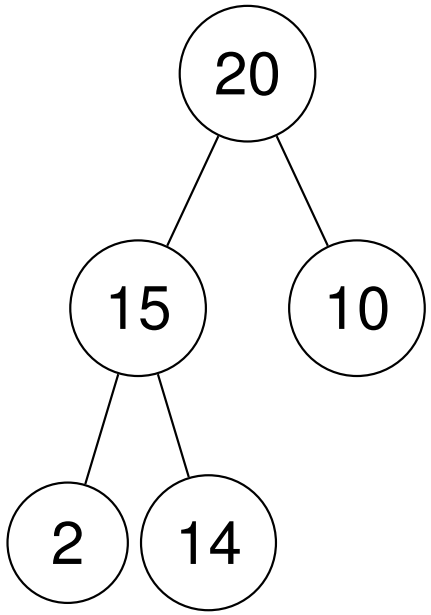
Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21



Inserción en un heap

Ejemplo

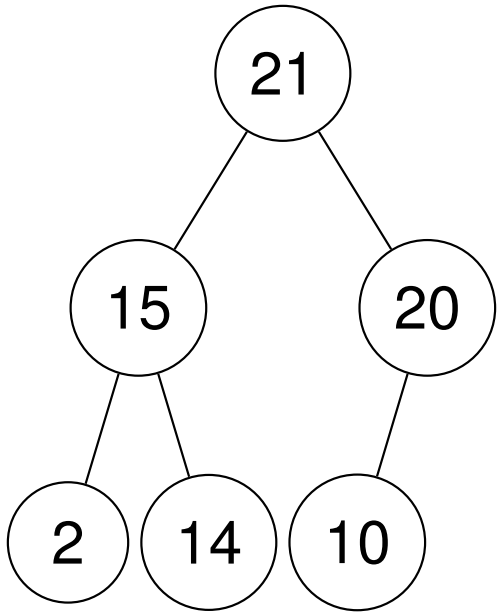
Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21



Inserción en un heap

Ejemplo

Inserta en un heap máximo inicialmente vacío los siguientes valores: 2, 10, 14, 15, 20 y 21



Inserción en un heap

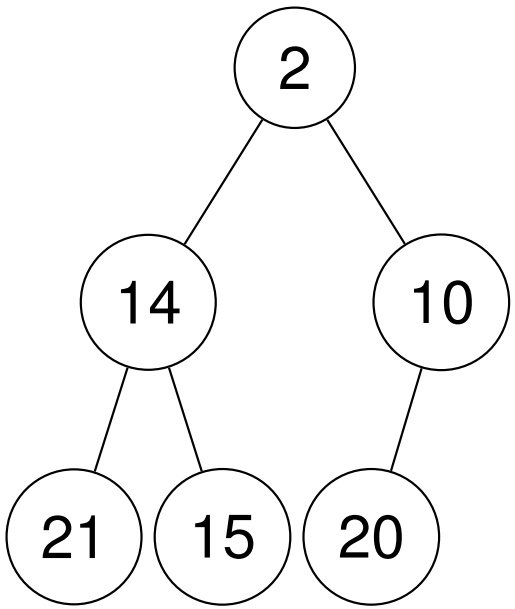
Ejercicio

Inserta en un heap mínimo inicialmente vacío los siguientes valores: 21, 20, 15, 2, 14, 10

Inserción en un heap

Ejercicio

Inserta en un heap mínimo inicialmente vacío los siguientes valores: 21, 20, 15, 2, 14, 10



Inserción en un heap

Pregunta

¿Cuándo se producen el mejor y peor caso en la inserción en un heap máximo? ¿Cuál es la complejidad asintótica respecto al número de elementos almacenados?

Inserción en un heap

Pregunta

¿Cuándo se producen el mejor y peor caso en la inserción en un heap máximo? ¿Cuál es la complejidad asintótica respecto al número de elementos almacenados?

- Mejor caso: el elemento a insertar es menor que su padre. $\Omega(1)$ (con implementación como vector)
- Peor caso: el elemento a insertar es mayor que sus ascendentes (y que el resto de elementos). $O(\log n)$

Borrado en un heap

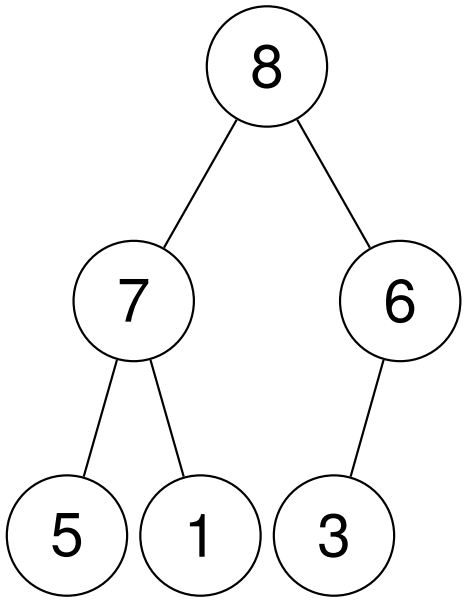
Algoritmo de borrado en un heap:

1. Mover el elemento más a la derecha del último nivel a la raíz
2. Repetir la siguiente operación mientras que el árbol no cumpla las propiedades de heap:
   - Árbol mínimo: intercambiar el elemento movido con el menor de sus hijos
   - Árbol máximo: intercambiar el elemento movido con el mayor de sus hijos

Borrado en un heap

Ejemplo

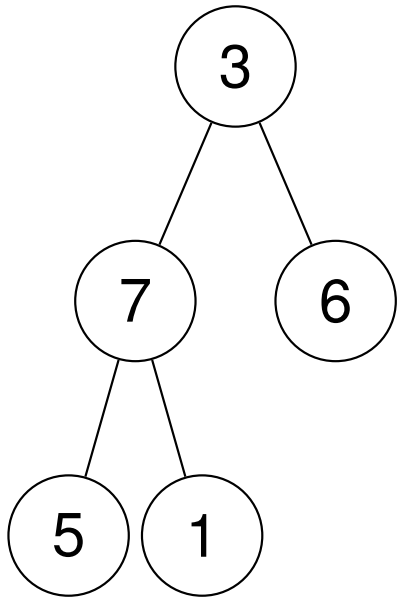
Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Ejemplo

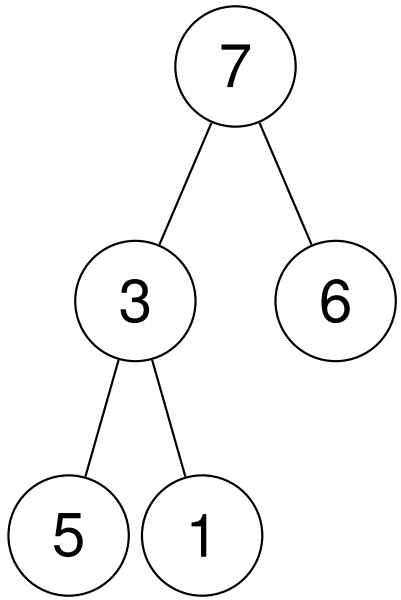
Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Ejemplo

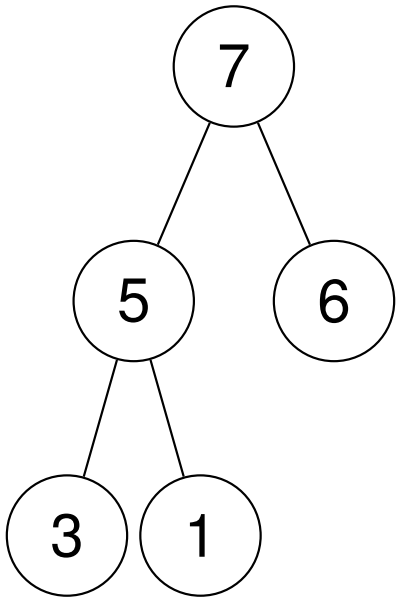
Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Ejemplo

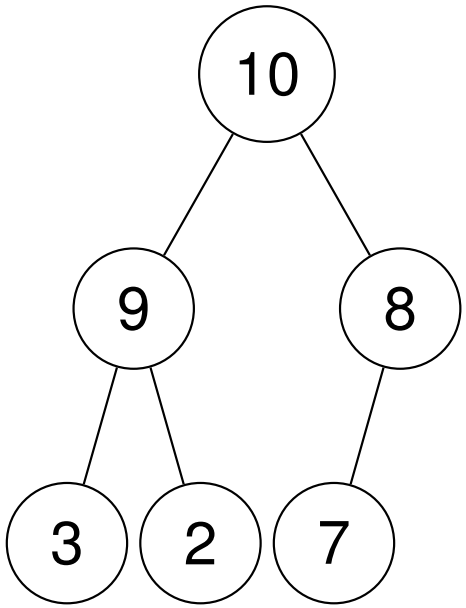
Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Ejercicio

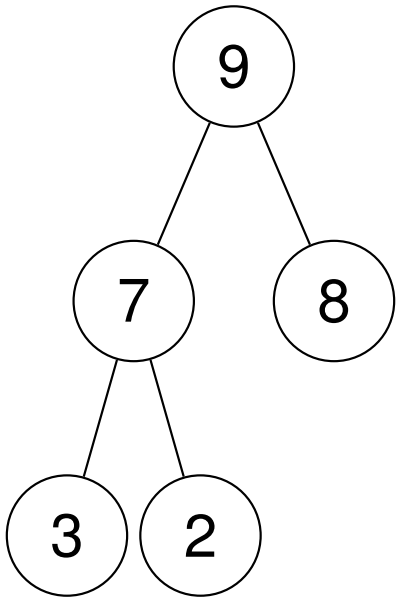
Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Ejercicio

Realiza un borrado en el siguiente heap máximo



Borrado en un heap

Pregunta

¿Cuándo se producen el mejor y peor caso en el borrado en un heap máximo? ¿Cuál es la complejidad asintótica respecto al número de elementos almacenados?

Borrado en un heap

Pregunta

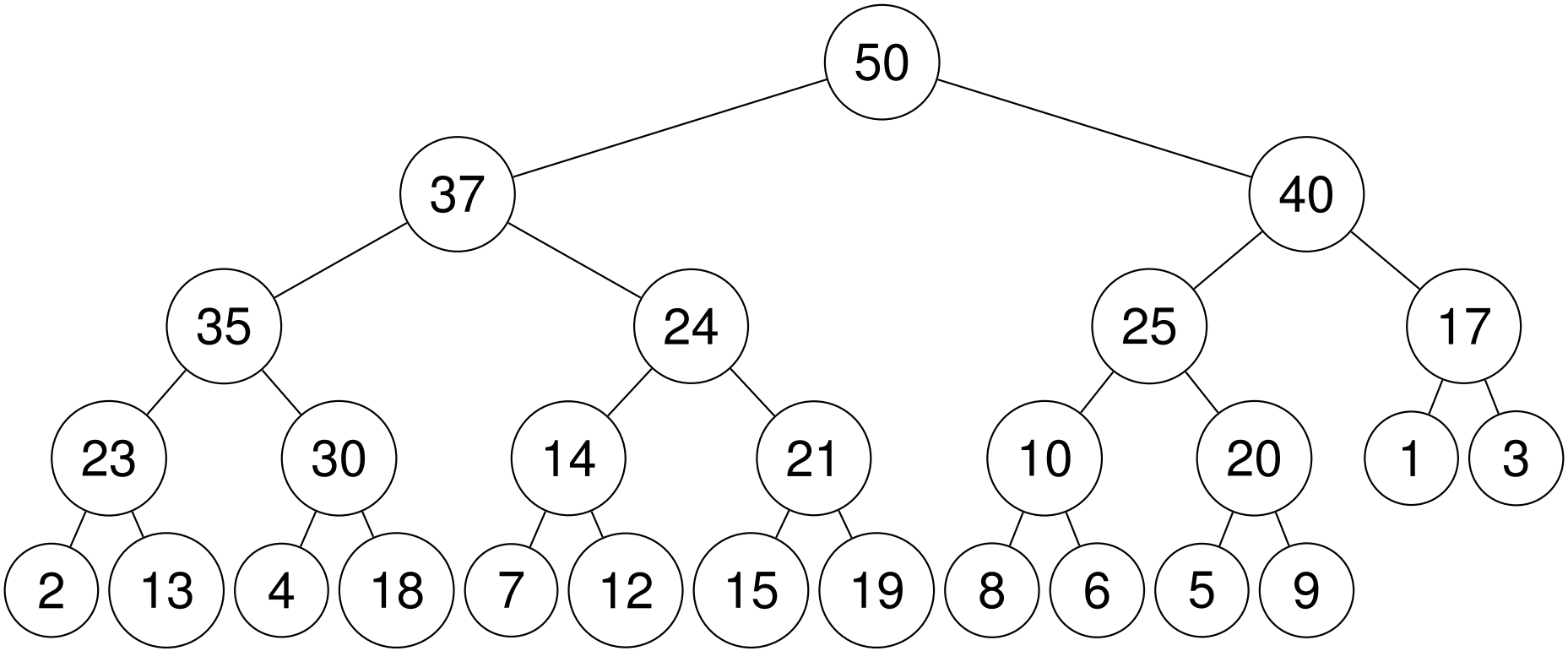
¿Cuándo se producen el mejor y peor caso en el borrado en un heap máximo? ¿Cuál es la complejidad asintótica respecto al número de elementos almacenados?

- Mejor caso: sólo se hace un intercambio de la raíz. $\Omega(1)$ (con implementación como vector)
- Peor caso: el elemento que se mueve a la reíz es menor que el resto de elementos del heap$\rightarrow$la raíz se intercambia hasta llegar a las hojas. $O(\log n)$

Ejercicios

Ejercicio

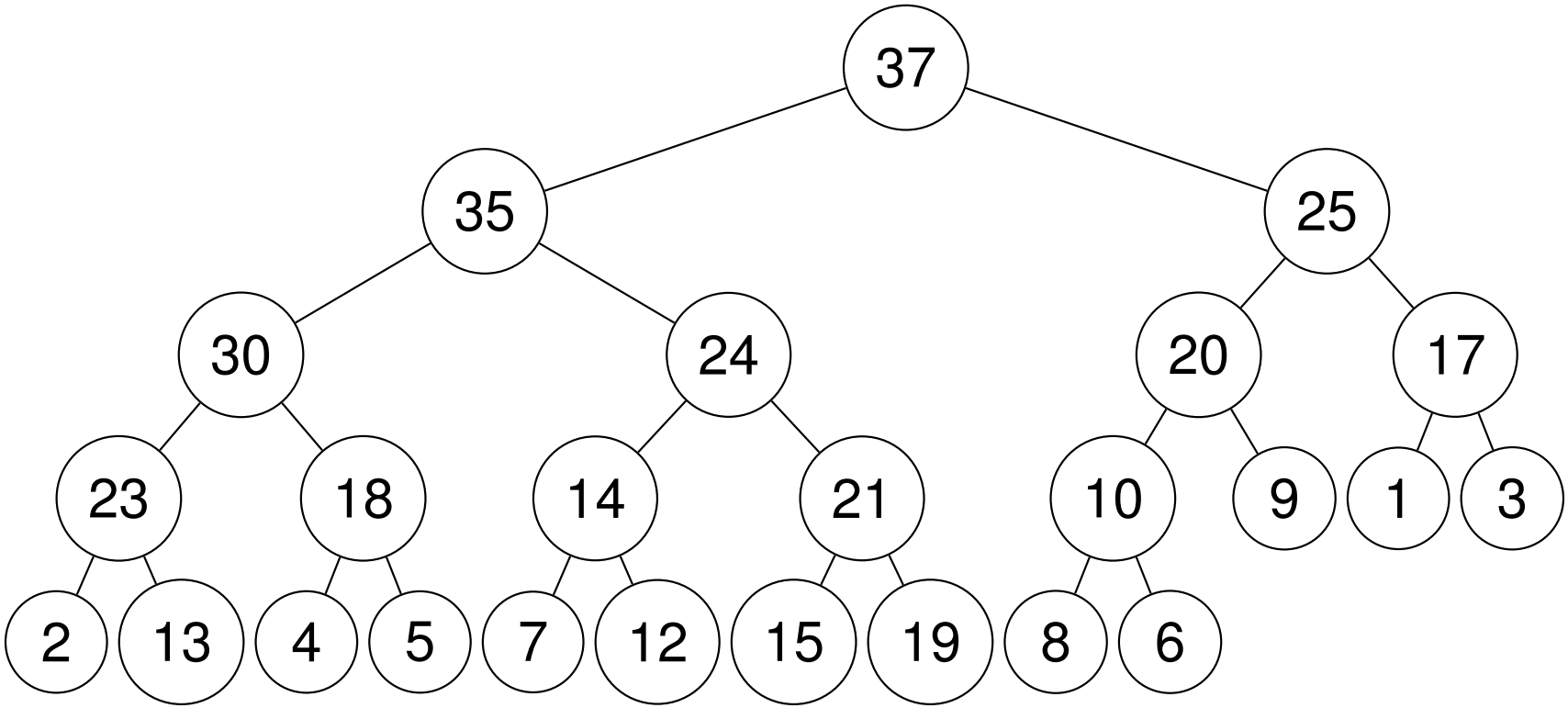
Realiza dos borrados sobre el siguiente montículo máximo:



Ejercicios

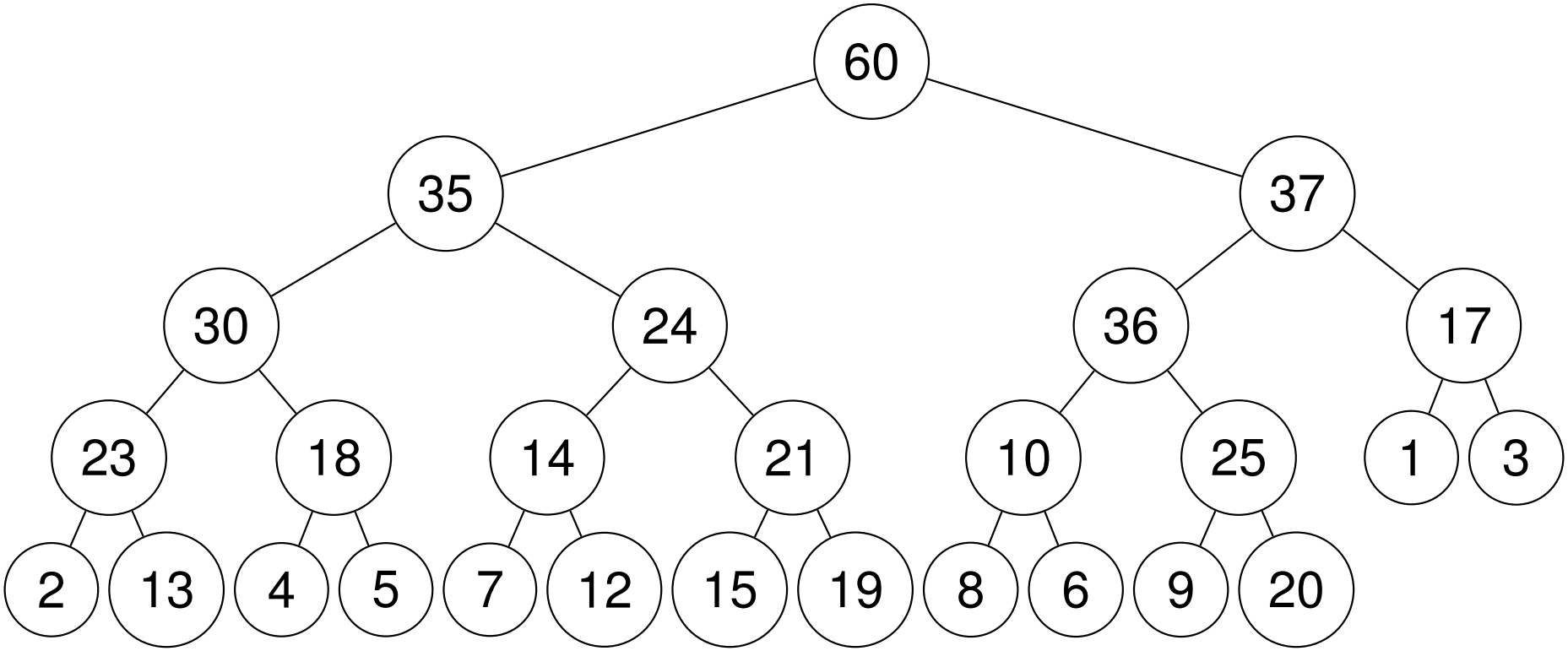
Ejercicio

Inserta 60, 36 en el siguiente montículo máximo:



Ejercicios

Resultado:



Representación

Pregunta

Teniendo en cuenta que un heap siempre es un árbol completo y el tipo de intercambios que se hacen, ¿qué representación es más adecuada: vector o enlazada?

Representación

Pregunta

Teniendo en cuenta que un heap siempre es un árbol completo y el tipo de intercambios que se hacen, ¿qué representación es más adecuada: vector o enlazada?

Las características del heap hacen que las grandes desventajas de la implementación vector desaparezcan: no hay desperdicio de memoria al ser un árbol completo, y los intercambios no implican desplazamientos de elementos en el vector. Además, insertar un elemento al final del último nivel tiene coste constante.

Índice

Algoritmo heapsort

- Puede emplearse la eficiencia de los heaps para ordenar un vector rápidamente
- Los algoritmos más simples para ordenación de vectores tienen complejidad $O(n^2)$
- *Heapsort*, el algoritmo de ordenación basado en heaps tiene una complejidad de $\Theta(n \cdot \log(n))$, la menor complejidad conocida para ordenación de vectores

Algoritmo heapsort

Primera fase:

- Parte izquierda del vector: heap
- Parte derecha del vector: elementos sin ordenar
- Se insertan los elementos de la parte desordenada de uno en uno

Segunda fase:

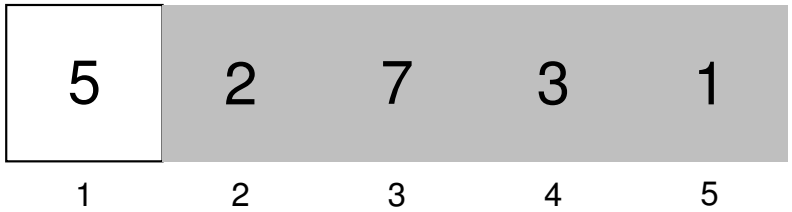
- Parte izquierda del vector: heap
- Parte derecha del vector: elementos ordenados
- Se borra iterativamente la raíz del heap y se pone en la parte derecha

Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 1:

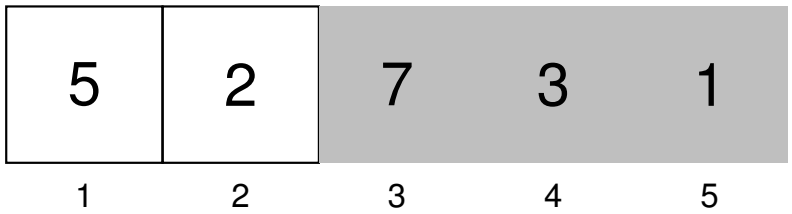


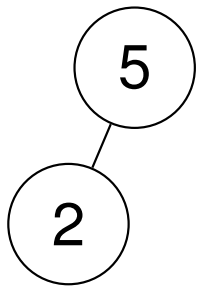
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 1:



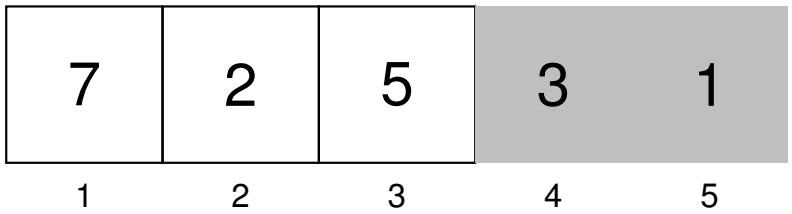


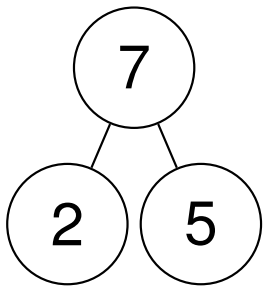
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 1:



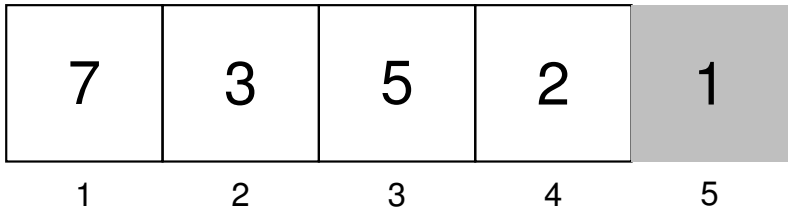


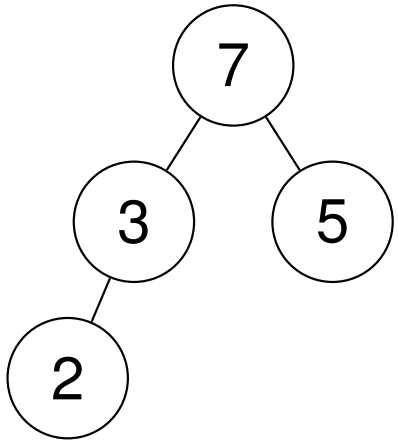
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 1:



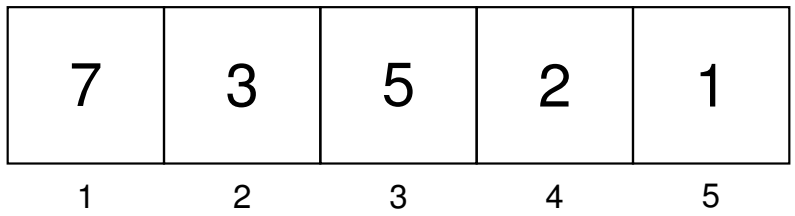


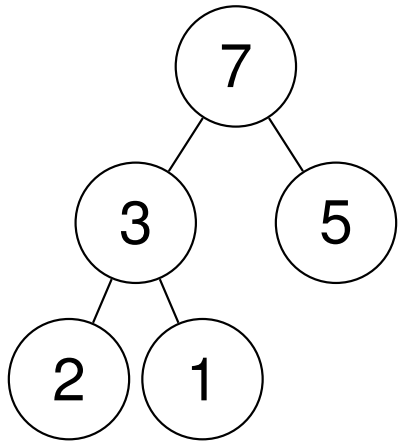
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 1:



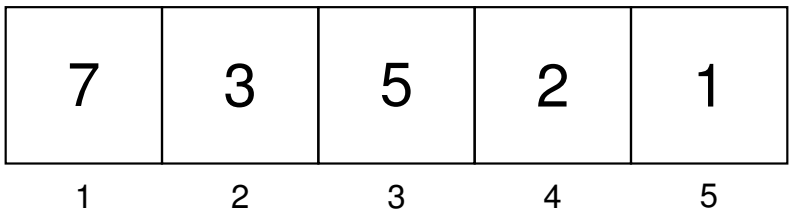


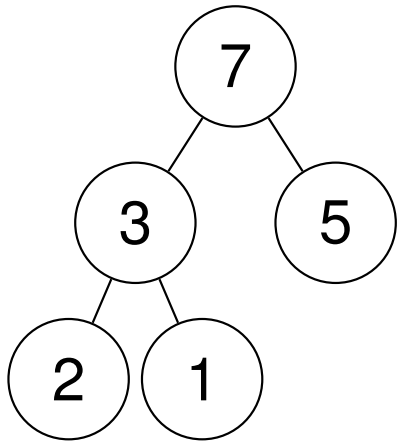
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 2:



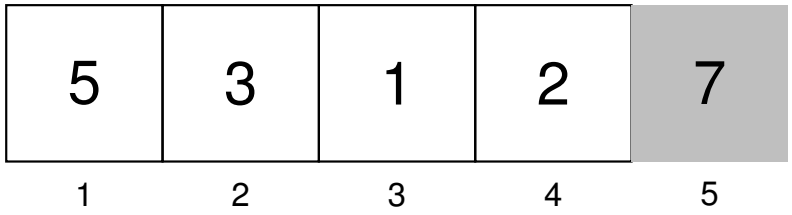


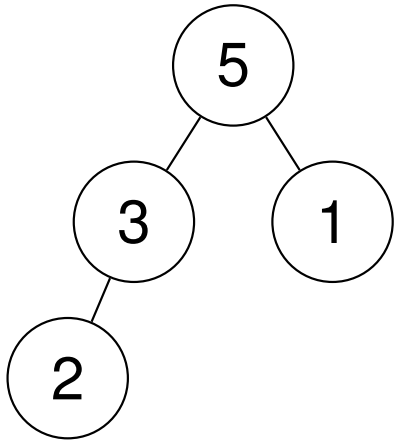
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 2:



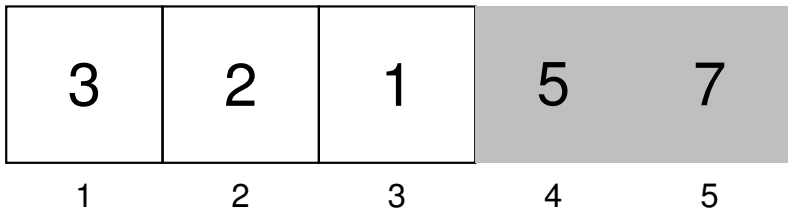


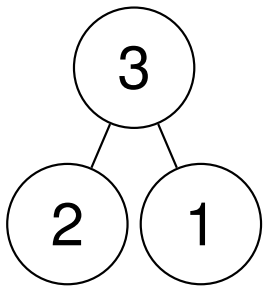
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 2:



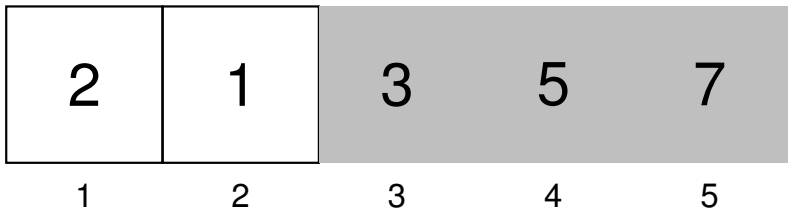


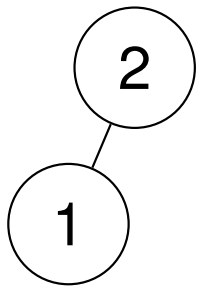
Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 2:



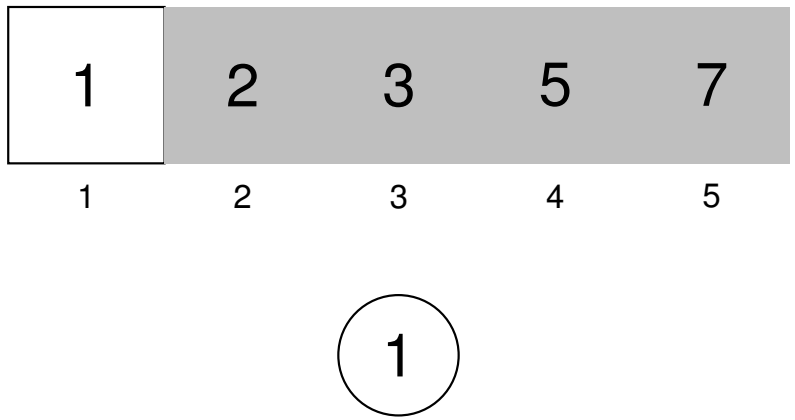


Algoritmo heapsort

Ejemplo

Ordena el siguiente vector empleando un heap máximo

Fase 2:



Algoritmo heapsort

Ejercicio

Ordena el siguiente vector empleando un heap mínimo (de mayor a menor): 9 5 7 4 8 6 2 1

Índice

- Implementada como un heap almacenado en vector

```
#include<queue>
using namespace std;
//...
priority_queue<int> q;

//Inserción en la cola con "push"
q.push(20);
q.push(100);
q.push(30);

//Obtención del elemento con máxima prioridad
cout << q.top() << endl; //100

//Eliminación del elemento con máxima prioridad
q.pop();

//Número de elementos en la colsa
cout << q.size() << endl;
```

Almacenando objetos

- Cualquier objeto que implemente un `operator<` puede almacenarse en la cola
- Se ordenarán de mayor a menor prioridad

```
#include<queue>
using namespace std;
//...
priority_queue<Peticion> q;


//Inserción en la cola con "push"
q.push(...);

//Obtención del elemento con máxima prioridad
cout << q.top().getNombreUsuario(); << endl;

//Eliminación del elemento con máxima prioridad
q.pop();
```