Colas de prioridad y montículos

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Bibliografía

- M. A. Weiss. *Data Structures and Algorithm Analysis in C++*. Fourth edition. Pearson, 2014.
  Capítulo 6
- R. Sedgewick y K. Wayne. *Algorithms*. Fourth Edition. Addison-Wesley, 2014.
  Sección 2.4
- N. Martí Oliet, Y. Ortega Mallén y A. Verdejo. Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos. Segunda edición, Garceta, 2013.
  Capítulo 8

Colas de prioridad

- En las colas “ordinarias” se atiende por riguroso orden de llegada (FIFO).
- También hay colas, como las de los servicios de urgencias, en las cuales se atiende según la urgencia y no según el orden de llegada: son colas de prioridad.
- Cada elemento tiene una prioridad que determina quién va a ser el primero en ser atendido; para poder hacer esto, hace falta tener un *orden total* sobre las prioridades.
- El primero en ser atendido puede ser el elemento con menor prioridad (por ejemplo, el cliente que necesita menos tiempo para su atención) o el elemento con mayor prioridad (por ejemplo, el cliente que esté dispuesto a pagar más por su servicio) según se trate de colas de prioridad de mínimos o de máximos, respectivamente.
- Para facilitar la presentación de las propiedades de la estructura de cola de prioridad, los elementos se identifican con su prioridad, de forma que el orden total es sobre elementos.

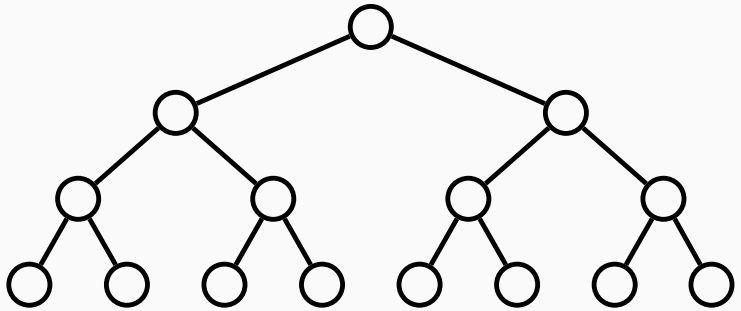
Colas de prioridad

El TAD de las colas de prioridad, `PriorityQueue<T>`, contiene las siguientes operaciones:

- crear una cola de prioridad vacía
- añadir un elemento, `void push(T const& elem)`
- consultar el primer elemento (el elemento más prioritario), `T const& top() const`
- eliminar el primer elemento, `void pop()`
- determinar si la cola de prioridad es vacía, `bool empty() const`
- consultar el número de elementos de la cola, `int size() const`

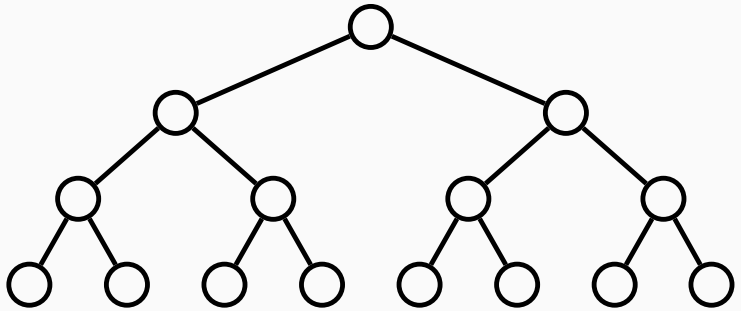
Árboles completos y semicompletos

- Un árbol binario de altura *h* es completo cuando todos sus nodos internos tienen dos hijos no vacíos, y todas sus hojas están en el nivel *h*.

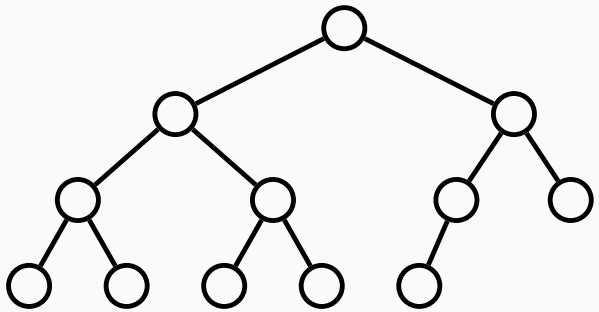


Árboles completos y semicompletos

- Un árbol binario de altura *h* es completo cuando todos sus nodos internos tienen dos hijos no vacíos, y todas sus hojas están en el nivel *h*.



- Un árbol binario de altura *h* es semicompleto si o bien es completo o tiene vacantes una serie de posiciones consecutivas del nivel *h* empezando por la derecha, de tal manera que al rellenar dichas posiciones con nuevas hojas se obtiene un árbol completo.



Árboles completos y semicompletos

Propiedades

- Un árbol binario completo de altura $h \geqslant 1$ tiene 2^{i-1} nodos en el nivel i, para todo i entre 1 y h.

  Por inducción sobre el número de nivel i.
  Cuando $i = 1$, en el primer nivel solamente hay un nodo que es la raíz, y $2^{1-1} = 1$.
  Suponiendo el resultado cierto para $i < h$, como cada nodo en el nivel i tiene exactamente dos hijos no vacíos, el número de nodos en el nivel $i + 1$ es igual a $2 * 2^{i-1} = 2^i = 2^{(i+1)-1}$.

- Un árbol binario completo de altura $h \geqslant 1$ tiene 2^{h-1} hojas.

  Las hojas son los nodos en el último nivel h.

- Un árbol binario completo de altura $h \geqslant 0$ tiene $2^h - 1$ nodos.

  Si $h = 0$, el árbol es vacío y el número de nodos es igual a $0 = 2^0 - 1$.
  Si $h > 0$, el número total de nodos es:

$$\sum_{i=1}^{h} 2^{i-1} = \sum_{j=0}^{h-1} 2^j = 2^h - 1.$$

Propiedades

- La altura de un árbol binario *semicompleto* formado por n nodos es $\lfloor \log n \rfloor + 1$.

  Supongamos un árbol binario semicompleto con n nodos y altura h.
  En el caso en que faltan más nodos en el último nivel, el árbol es un árbol binario completo de $h-1$ niveles más un nodo en el nivel h, por lo que hay en total $2^{h-1} - 1 + 1 = 2^{h-1}$ nodos.
  En el caso en que el último nivel está todo lleno, tendremos un árbol binario completo de h niveles con $2^h - 1$ nodos.
  Resumiendo, tenemos con respecto a n la siguiente desigualdad:

  $$2^{h-1} \leqslant n \leqslant 2^h - 1.$$

  Tomando logaritmos en base 2

  $$\log(2^{h-1}) \leqslant \log n \leqslant \log(2^h - 1) < \log(2^h);$$

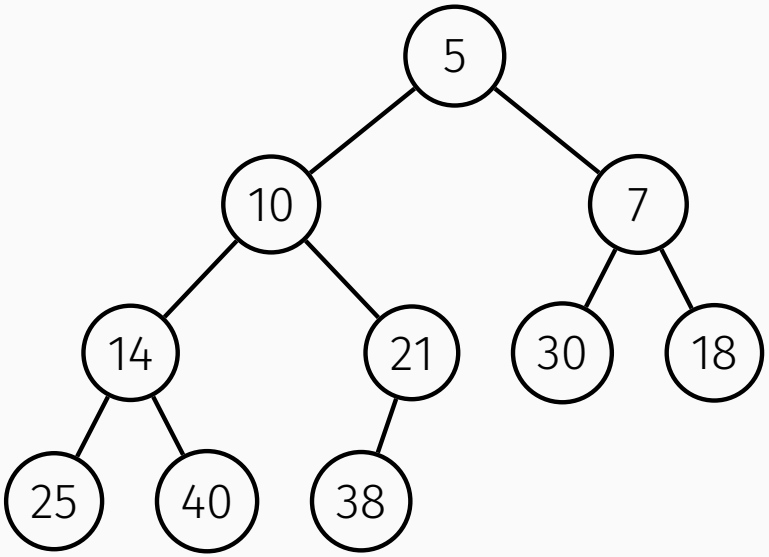
  equivalentemente,

  $$h - 1 \leqslant \log n < h,$$

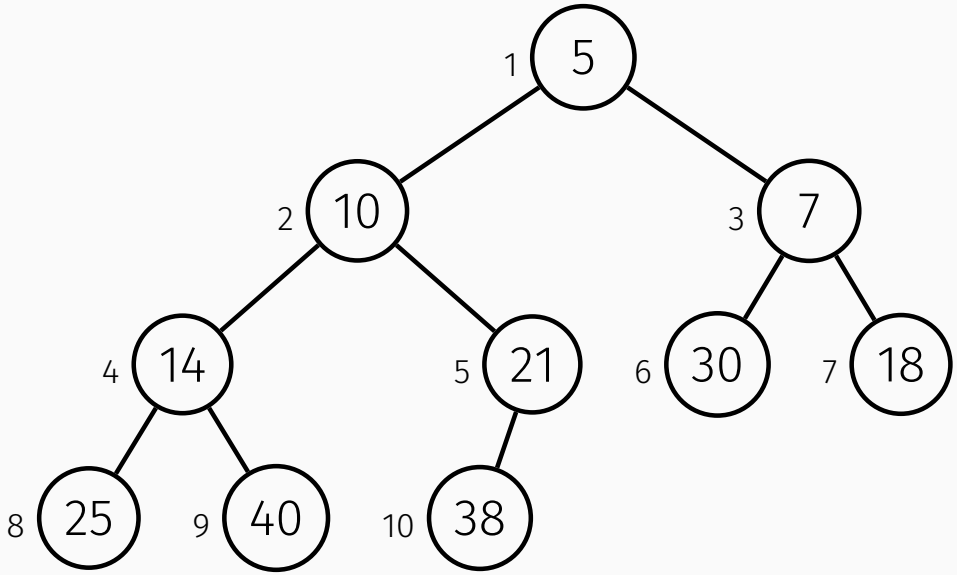
  es decir, $h - 1 = \lfloor \log n \rfloor$ y de aquí $h = \lfloor \log n \rfloor + 1$.

Montículos binarios

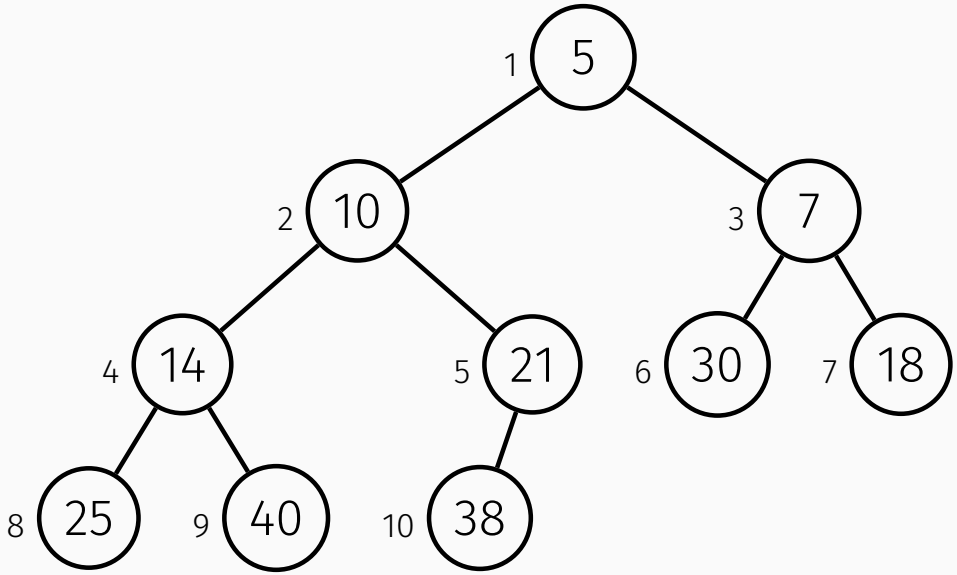
- Un montículo (binario) de mínimos es un árbol binario semicompleto donde el elemento en la raíz es menor que todos los elementos en el hijo izquierdo y en el derecho, y ambos hijos son a su vez montículos de mínimos.
- Equivalentemente, el elemento en cada nodo es menor que los elementos en las raíces de sus hijos y, por tanto, que todos sus descendientes; así, la raíz del árbol contiene el mínimo de todos los elementos en el árbol.

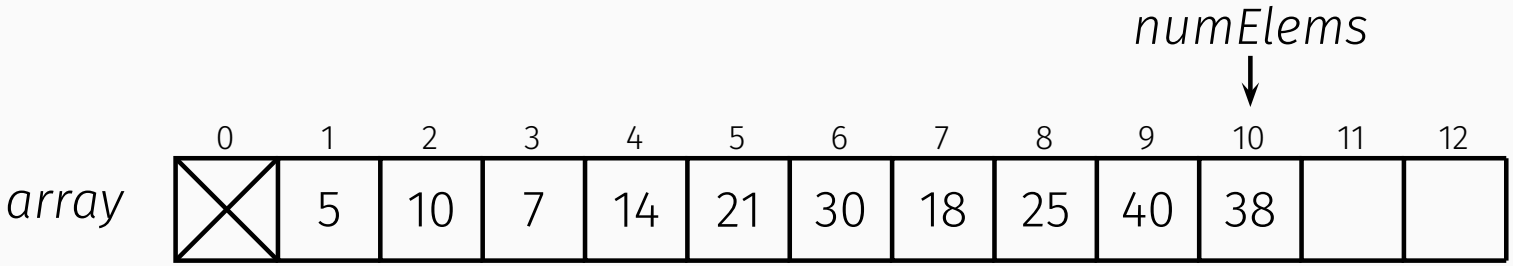


Implementación de montículos



Implementación de montículos





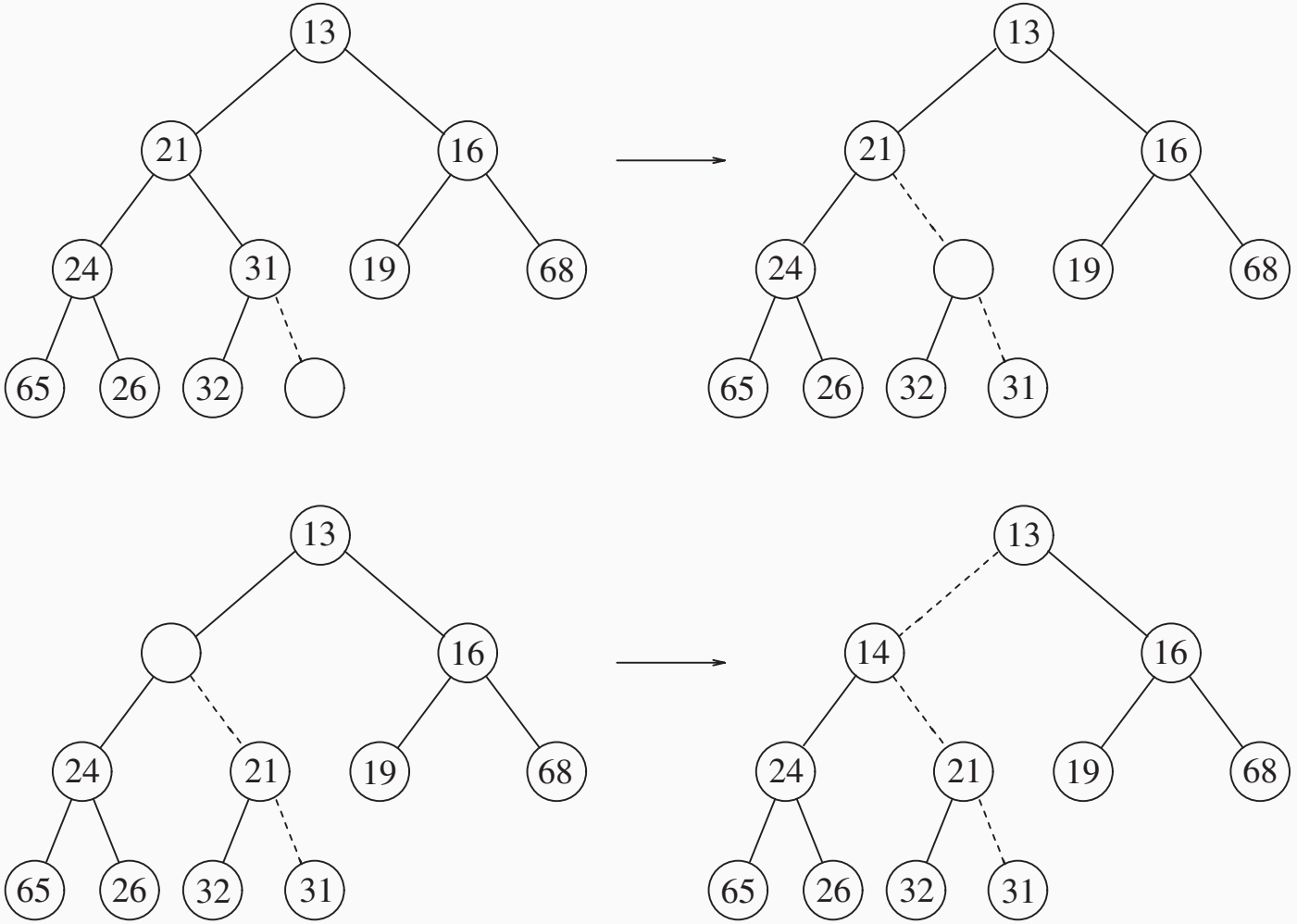
Implementación de las colas de prioridad mediante montículos

PriorityQueue.h

```
// Comparator dice cuándo un valor de tipo T es más prioritario que otro
template <typename T = int, typename Comparator = std::less<T>>
class PriorityQueue {

   // vector que contiene los datos
   std::vector<T> array;     // primer elemento en la posición 1

   /* Objeto función que sabe comparar elementos.
      antes(a,b) es cierto si a es más prioritario que b
      (a debe salir antes que b) */
   Comparator antes;

public:

   PriorityQueue(Comparator c = Comparator()) : array(1), antes(c) {}
```

Implementación de las colas de prioridad mediante montículos

- Inserción del 14:



Implementación de las colas de prioridad mediante montículos

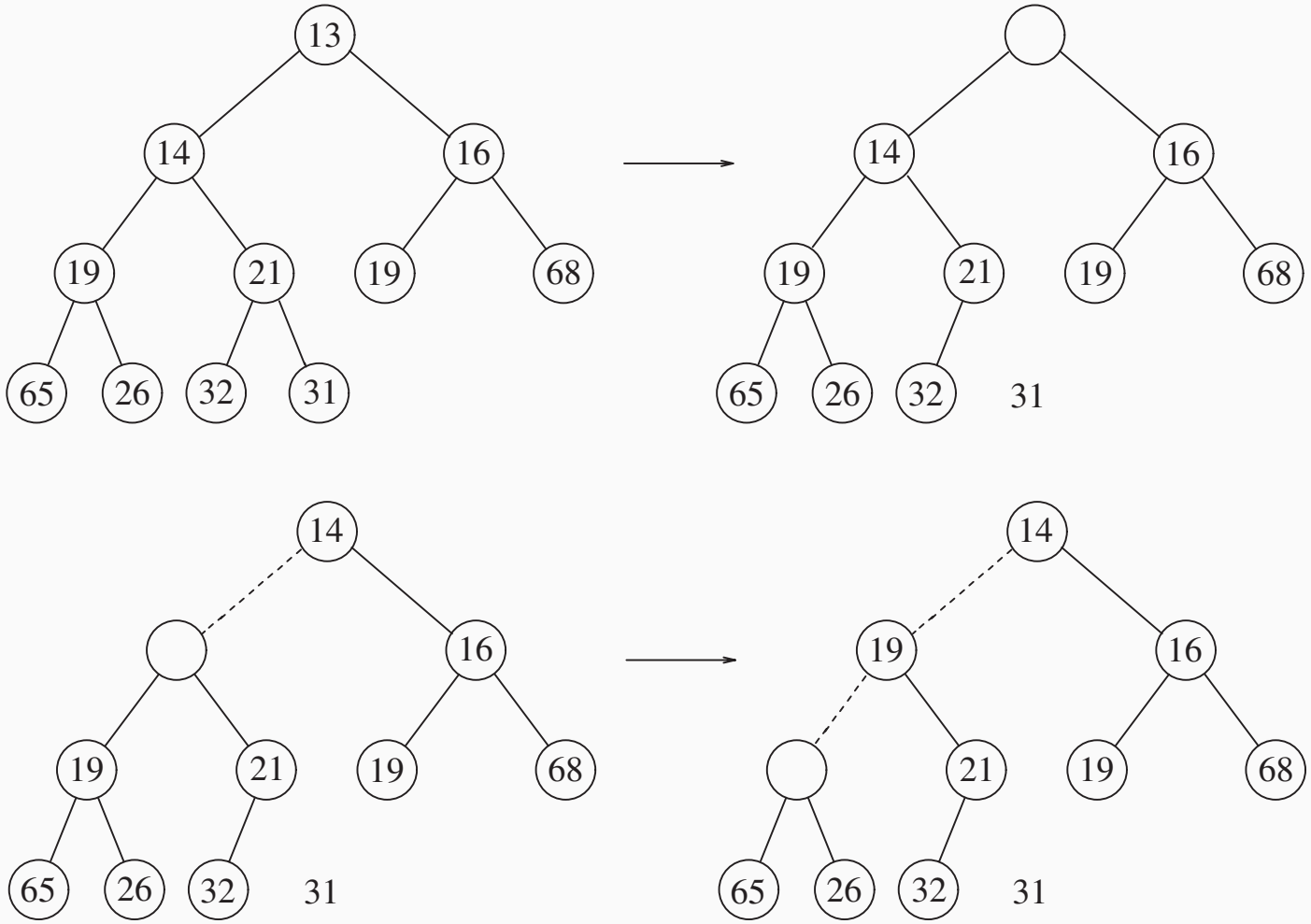
```
public:
   void push(T const& x) {
      array.push_back(x);
      flotar(array.size() - 1);
   }

private:
   void flotar(int i) {
      T elem = array[i];
      int hueco = i;
      while (hueco != 1 && antes(elem, array[hueco / 2])) {
         array[hueco] = array[hueco / 2];
         hueco /= 2;
      }
      array[hueco] = elem;
   }
```

Implementación de las colas de prioridad mediante montículos

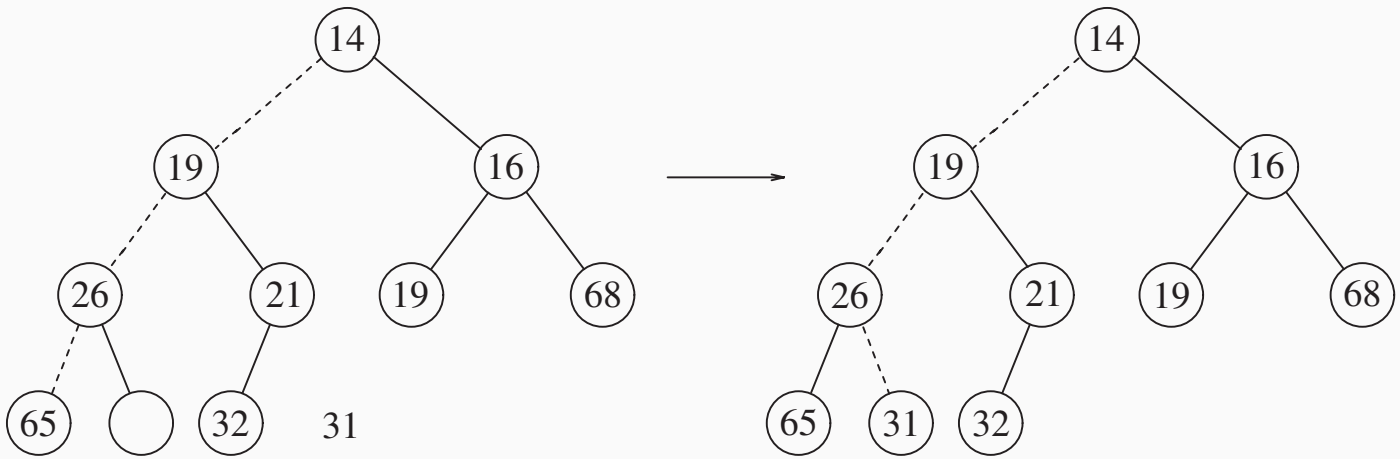
```
int size() const {
   return array.size() - 1;
}

bool empty() const {
   return size() == 0;
}

T const& top()  const {
   if (empty())
      throw std::domain_error("La cola vacia no tiene top");
   else return array[1];
}
```

Implementación de las colas de prioridad mediante montículos

- Eliminación del primero:



Implementación de las colas de prioridad mediante montículos



Implementación de las colas de prioridad mediante montículos

```
void pop() {
   if (empty())
      throw std::domain_error("Imposible eliminar de una cola vacia");
   else { array[1] = array.back(); array.pop_back();
          if (!empty()) hundir(1);
   }
}
void hundir(int i) {
  T elem = array[i];
  int hueco = i;
  int hijo = 2 * hueco; // hijo izquierdo, si existe
  while (hijo <= size())  {
     // cambiar al hijo derecho si existe y va antes que el izquierdo
     if (hijo < size() && antes(array[hijo + 1], array[hijo]))
        ++hijo;
     // flotar el hijo menor si va antes que el elemento hundiéndose
     if (antes(array[hijo], elem)) {
        array[hueco] = array[hijo];
        hueco = hijo; hijo = 2 * hueco;
     } else break;
  }
  array[hueco] = elem;
}
```

Resumen de costes de implementaciones de colas de prioridad

order-of-growth of running time for priority queue with N items

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	log N	log N	1
d-ary heap	$\log_d N$	$d \log_d N$	1
Fibonacci	1	log N †	1
impossible	1	1	1

← why impossible?

† amortized

Convertir un vector en un montículo

```
void monticulizar1() {
   for (int i = 2; i <= size(); ++i) {
      flotar(i);
   }
}
```

Convertir un vector en un montículo

```
void monticulizar1() {
   for (int i = 2; i <= size(); ++i) {
      flotar(i);
   }
}
```

nivel	nodos	flotan
2	2	cada uno 1
3	4	cada uno 2
	$\vdots$	
i	2^{i-1}	cada uno $i-1$
	$\vdots$	
h	2^{h-1}	cada uno $h-1$

Convertir un vector en un montículo

```
void monticulizar1() {
   for (int i = 2; i <= size(); ++i) {
      flotar(i);
   }
}
```

nivel	nodos	flotan
2	2	cada uno 1
3	4	cada uno 2
	$\vdots$	
i	2^{i-1}	cada uno $i-1$
	$\vdots$	
h	2^{h-1}	cada uno $h-1$

$$\sum_{i=2}^{h}(i-1)2^{i-1} = \sum_{j=1}^{h-1} j2^j = (h-2)2^h + 2 = (\lfloor \log N \rfloor - 1)2^{\lfloor \log N \rfloor + 1} + 2 \in \Theta(N \log N)$$

Convertir un vector en un montículo

```
void monticulizar2() {
   for (int i = size()/2; i >= 1; --i)
      hundir(i);
}
```

Convertir un vector en un montículo

```
void monticulizar2() {
   for (int i = size()/2; i >= 1; --i)
      hundir(i);
}
```

nivel	nodos	hunden
h	2^{h-1}	nada
$h-1$	2^{h-2}	cada uno 1
$h-2$	2^{h-3}	cada uno 2
	$\vdots$	
i	2^{i-1}	cada uno $h-i$
	$\vdots$	
1	1	$h-1$

Convertir un vector en un montículo

```
void monticulizar2() {
   for (int i = size()/2; i >= 1; --i)
      hundir(i);
}
```

nivel	nodos	hunden
h	2^{h-1}	nada
$h-1$	2^{h-2}	cada uno 1
$h-2$	2^{h-3}	cada uno 2
	$\vdots$	
i	2^{i-1}	cada uno $h-i$
	$\vdots$	
1	1	$h-1$

$$\sum_{i=1}^{h-1}(h-i)2^{i-1} = \sum_{j=2}^{h}(j-1)2^{h-j} < \sum_{j=1}^{h} j2^{h-j} = 2^h \sum_{j=1}^{h} \frac{j}{2^j}$$

$$= 2^h(2 - \frac{h+2}{2^h}) \leqslant 2^{h+1} = 2^{\lfloor \log N \rfloor + 2} \in O(N)$$

Heapsort

Método de ordenación basado en la utilización de un montículo.

```
void heapsort_abstracto(std::vector<int> & v) {
    PriorityQueue<int> colap;
    for (int e : v)
        colap.push(e);
    for (int i = 0; i < v.size(); ++i) {
        v[i] = colap.top();
        colap.pop();
    }
}
```

El coste en tiempo está en $\Theta(N \log N)$, y en espacio adicional en $\Theta(N)$.

Heapsort

- Podemos ahorrarnos este espacio adicional si utilizamos el mismo vector para representar el montículo auxiliar.
- Primero el vector se convierte en un montículo.
- Después se recorren las posiciones del vector de derecha a izquierda extrayendo cada vez el primero del montículo para colocarlo al principio de la parte de la derecha ya ordenada.

Heapsort

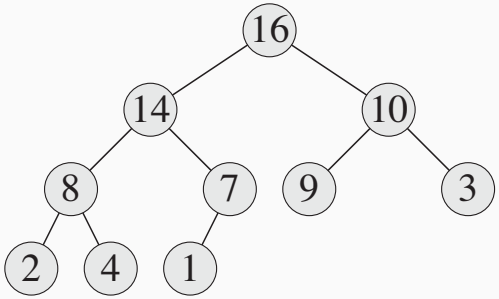
- Podemos ahorrarnos este espacio adicional si utilizamos el mismo vector para representar el montículo auxiliar.
- Primero el vector se convierte en un montículo.
- Después se recorren las posiciones del vector de derecha a izquierda extrayendo cada vez el primero del montículo para colocarlo al principio de la parte de la derecha ya ordenada.


```
template <typename T, typename Comparador = std::less<T>>
void heapsort(std::vector<T> & v, Comparador cmp = Comparador()) {
    // monticulizar
    for (int i = (v.size() - 1) / 2; i >= 0; --i)
        hundir_max(v, v.size(), i, cmp);
    // ordenar
    for (int i = v.size() - 1; i > 0; --i) {
        std::swap(v[i], v[0]);
        hundir_max(v, i, 0, cmp);
    }
}
```

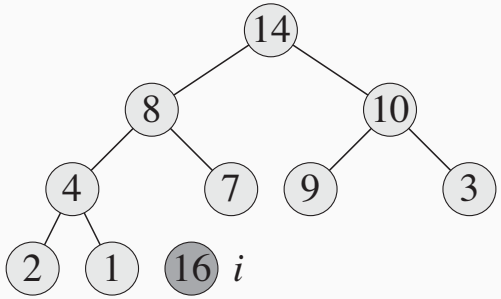
Heapsort

```
template <typename T, typename Comparador>
void hundir_max(std::vector<T> & v, int N, int j, Comparador cmp) {
   // montículo en v en posiciones de 0 a N-1
   T elem = v[j];
   int hueco = j;
   int hijo = 2*hueco + 1; // hijo izquierdo, si existe
   while (hijo < N) {
      // cambiar al hijo derecho si existe y va antes que el izquierdo
      if (hijo + 1 < N && cmp(v[hijo], v[hijo + 1]))
         hijo = hijo + 1;
      // flotar el hijo menor si va antes que el elemento hundiéndose
      if (cmp(elem, v[hijo])) {
         v[hueco] = v[hijo];
         hueco = hijo; hijo = 2*hueco + 1;
      } else break;
   }
   v[hueco] = elem;
}
```

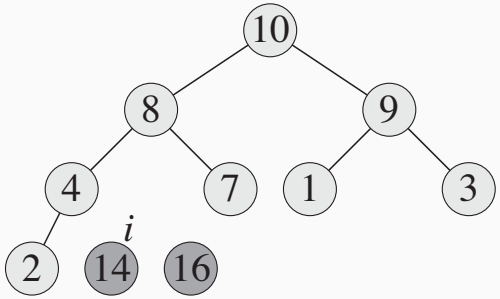
Heapsort



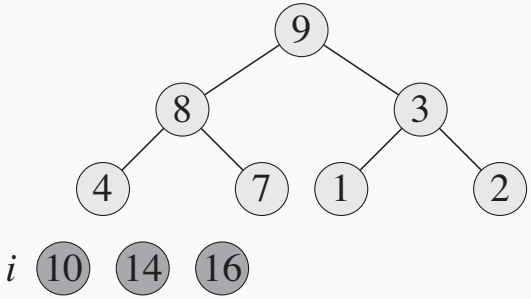
(a)



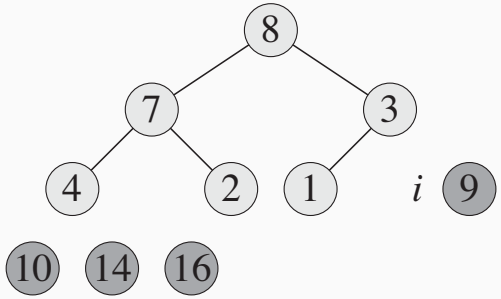
(b)



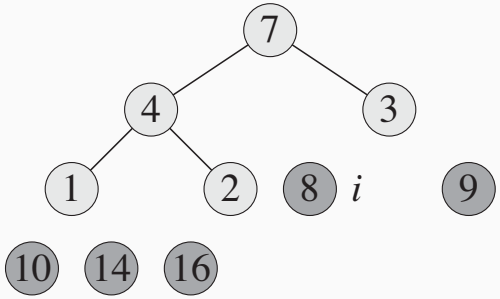
(c)



(d)

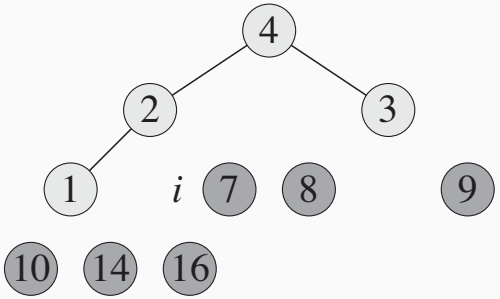


(e)

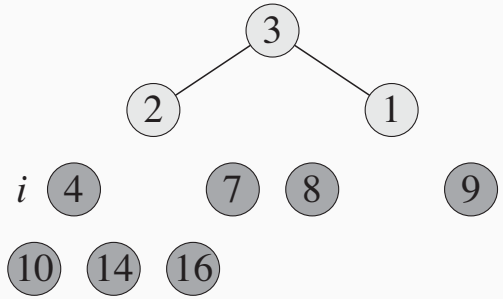


(f)

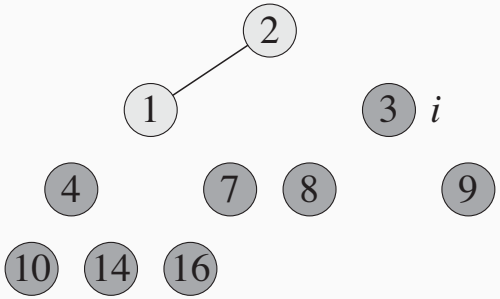
Heapsort



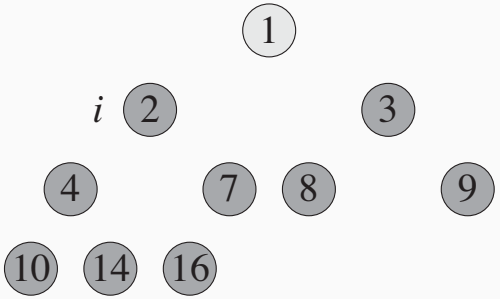
(g)



(h)



(i)



(j)

(k)

Heapsort

```
vector<string> datos {"Zorro", "Lobo", "abeja", "leon", "perro", "gato"};

heapsort(datos);
```

Heapsort

```
vector<string> datos {"Zorro", "Lobo", "abeja", "leon", "perro", "gato"};

heapsort(datos);
```

→ Lobo Zorro abeja gato leon perro

Heapsort

```
vector<string> datos {"Zorro", "Lobo", "abeja", "leon", "perro", "gato"};

heapsort(datos);

 → Lobo Zorro abeja gato leon perro

class ComparaString {
public:
    bool operator()(string const& a, string const& b) const {
        return aMinusculas(a) < aMinusculas(b);
    }
};

heapsort(datos, ComparaString());
```

Heapsort

```
vector<string> datos {"Zorro", "Lobo", "abeja", "leon", "perro", "gato"};

heapsort(datos);

 → Lobo Zorro abeja gato leon perro

class ComparaString {
public:
    bool operator()(string const& a, string const& b) const {
        return aMinusculas(a) < aMinusculas(b);
    }
};

heapsort(datos, ComparaString());

 → abeja gato leon Lobo perro Zorro
```

Heapsort

```
vector<string> datos {"Zorro", "Lobo", "abeja", "leon", "perro", "gato"};

heapsort(datos);

 → Lobo Zorro abeja gato leon perro

class ComparaString {
public:
    bool operator()(string const& a, string const& b) const {
        return aMinusculas(a) < aMinusculas(b);
    }
};

heapsort(datos, ComparaString());

 → abeja gato leon Lobo perro Zorro

heapsort(datos, [](string const& a, string const& b) {
                    return aMinusculas(a) < aMinusculas(b); } );
```

Unidad Curiosa de Monitorización

La *Unidad Curiosa de Monitorización* (UCM) se encarga de leer los datos proporcionados por una serie de sensores y enviar con cierta periodicidad los datos obtenidos y procesados a los usuarios que se han registrado previamente.

La UCM admite que los usuarios se registren proporcionando un *Identificador*, un número que identifica de forma única al usuario, y un *Periodo*, el intervalo de tiempo que transcurrirá entre dos envíos consecutivos de información a ese usuario. Es decir, cuando hayan pasado *Periodo* segundos desde que el usuario se registró, este recibirá la información de la UCM por primera vez; y después recibirá la información cada *Periodo* segundos.

Acaban de registrarse varios usuarios. ¿Podrías decir a quiénes irán dirigidos los *K* primeros envíos de información? Si dos o más usuarios tienen que recibir la información al mismo tiempo, los envíos se realizan en orden creciente de sus identificadores de usuario.

Unidad Curiosa de Monitorización

```
#include "PriorityQueue.h"

struct registro {
   int momento;  // cuándo le toca
   int id;       // identificador (se utiliza en caso de empate)
   int periodo;  // tiempo entre consultas
};

bool operator<(registro const& a, registro const& b) {
   return a.momento < b.momento ||
          (a.momento == b.momento && a.id < b.id);
}

bool resuelveCaso() {
   int N;
   cin >> N; // número de usuarios registrados

   if (N == 0) // no hay más casos
      return false;
```

Unidad Curiosa de Monitorización

```
    PriorityQueue<registro> cola;

    // leemos los registros
    for (int i = 0; i < N; ++i) {
        int id_usu, periodo;
        cin >> id_usu >> periodo;
        cola.push({periodo, id_usu, periodo});
    }

    int envios; // número de envíos a mostrar
    cin >> envios;

    while (envios--) {
        registro e = cola.top(); cola.pop();
        cout << e.id << '\n';
        e.momento += e.periodo;
        cola.push(e);
    }
    return true;
}
```

Unidad Curiosa de Monitorización

```
    PriorityQueue<registro> cola;

    // leemos los registros
    for (int i = 0; i < N; ++i) {
        int id_usu, periodo;
        cin >> id_usu >> periodo;
        cola.push({periodo, id_usu, periodo});
    }

    int envios; // número de envíos a mostrar
    cin >> envios;

    while (envios--) {
        registro e = cola.top(); cola.pop();
        cout << e.id << '\n';
        e.momento += e.periodo;
        cola.push(e);
    }
    return true;
}
```

$O(N \log N + K \log N)$, donde N es el número de usuarios y K el número de consultas

Colas de prioridad en la STL

- La librería `queue` de la STL contiene la clase `priority_queue` que implementa colas de prioridad de máximos.
- Dado un orden, como $<$, la operación `top` devuelve el elemento mayor, el que se encuentra más a la derecha en el orden

$$a_1 < a_2 < \cdots < a_n$$

- Podemos utilizar esas colas como colas de mínimos si cambiamos el objeto comparador, utilizando el operador $>$:

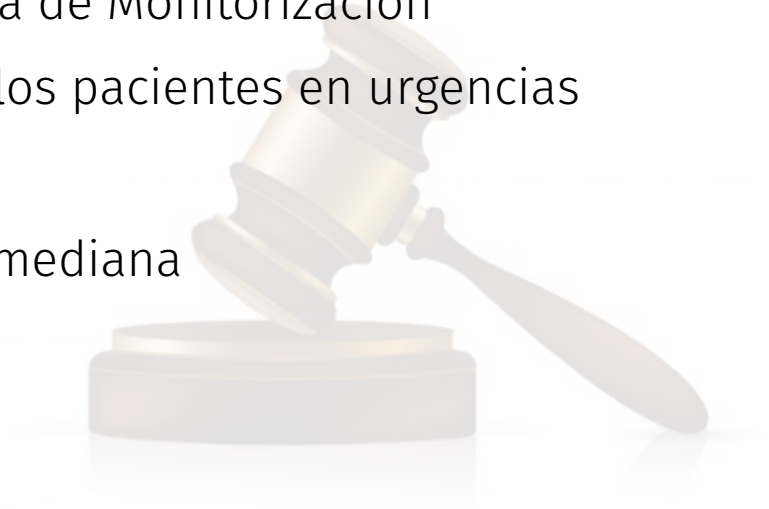
$$b_1 > b_2 > \cdots > b_n$$

- El comparador es el tercer argumento de la plantilla:

```
priority_queue<int, vector<int>, std::greater<int>> cola_min;
```

Problemas

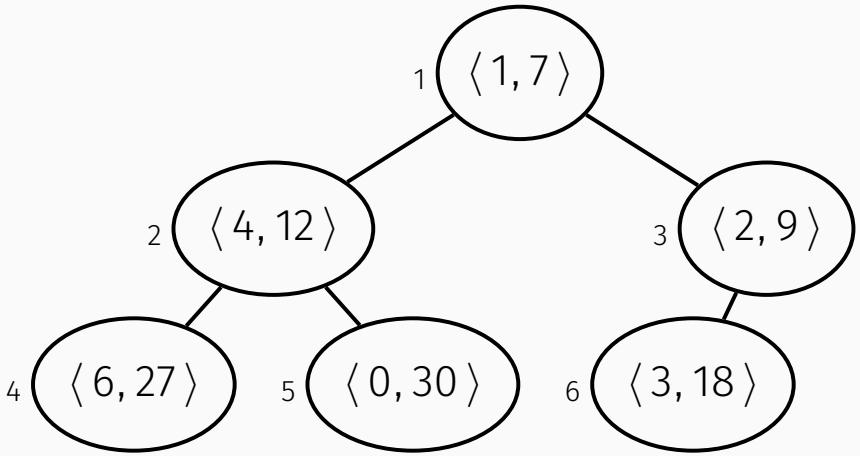
- 20 - Lo que cuesta sumar
- 21 - Unidad Curiosa de Monitorización
- 22 - Ordenando a los pacientes en urgencias
- 23 - Multitarea
- 24 - Cálculo de la mediana



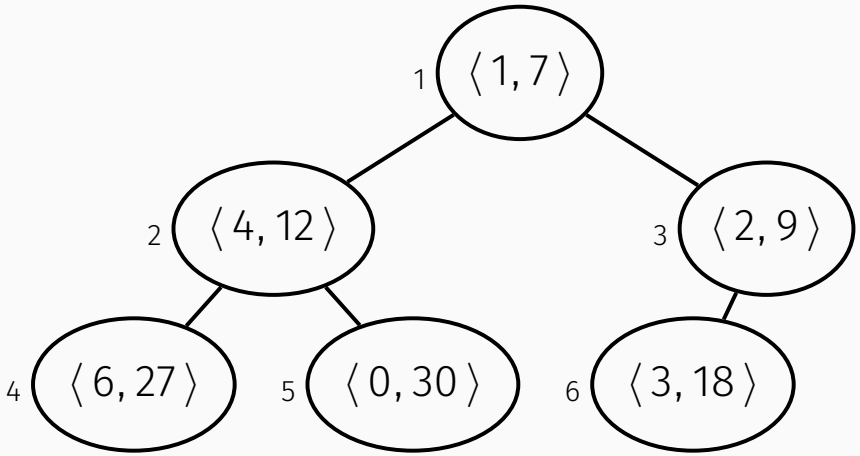
Prioridades variables asociadas a elementos con identificador

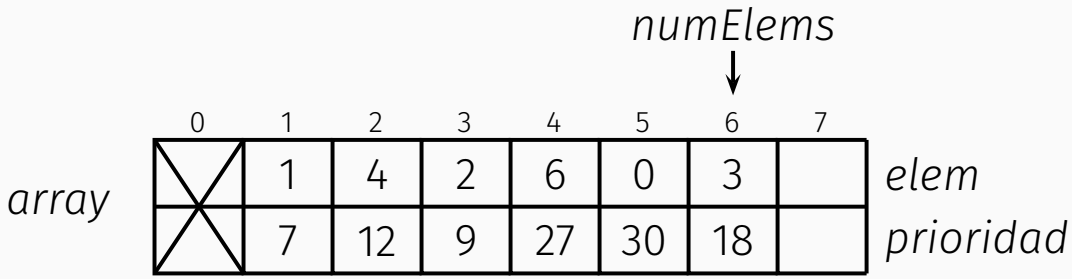
- Queremos una cola de prioridad que almacene elementos en el intervalo $[0..N)$ cada uno con una prioridad asociada.
- Y queremos poder modificar la prioridad asociada a un elemento en tiempo logarítmico.
- Utilizaremos un montículo de pares de la forma $\langle \mathit{elem}, \mathit{prioridad} \rangle$ donde *elem* es un número natural en el intervalo $[0..N)$ y todos son diferentes.
- El orden entre los pares viene inducido por el orden entre las prioridades.

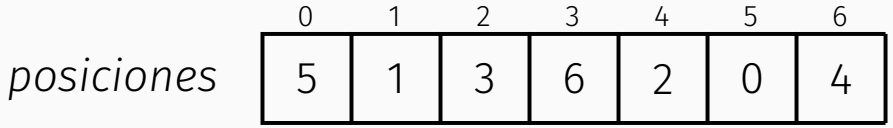
Montículo con prioridades variables



Montículo con prioridades variables







$$array[posiciones[i]].elem = i$$

Montículo con prioridades variables

IndexPQ.h

```
template <typename T = int, typename Comparator = std::less<T>>
class IndexPQ {
public:
   struct Par { // registro para las parejas < elem, prioridad >
      int elem;
      T prioridad;
   };
private:
   // vector que contiene los datos (pares < elem, prio >)
   std::vector<Par> array;     // primer elemento en la posición 1

   // vector que contiene las posiciones en array de los elementos
   std::vector<int> posiciones; // un 0 indica que el elemento no está

   /* Objeto función que sabe comparar prioridades.
      antes(a,b) es cierto si a es más prioritario que b */
   Comparator antes;
```

Montículo con prioridades variables

```
public:
   /** Constructor */
   IndexPQ(int N, Comparator c = Comparator()) :
     array(1), posiciones(N, 0), antes(c) {};

   Par const& top() const {
       if (size() == 0) throw std::domain_error("Error cola vacia.");
       else return array[1];
   }

   void pop() {
      if (size() == 0) throw std::domain_error("Error cola vacia.");
      else {
         posiciones[array[1].elem] = 0; // para indicar que no está
         if (size() > 1) {
            array[1] = std::move(array.back());
            posiciones[array[1].elem] = 1;
            array.pop_back();
            hundir(1);
         } else array.pop_back();
      }
   }
```

Montículo con prioridades variables

```
private:
  void hundir(int i) {
    Par mov = array[i];
    int hueco = i;
    int hijo = 2*hueco; // hijo izquierdo, si existe
    while (hijo <= size())  {
        // cambiar al hijo derecho si existe y va antes que el izquierdo
        if (hijo < size() &&
             antes(array[hijo + 1].prioridad, array[hijo].prioridad))
            ++hijo;
        // flotar el hijo menor si va antes que el elemento hundiéndose
        if (antes(array[hijo].prioridad, mov.prioridad)) {
            array[hueco] = array[hijo];
            posiciones[array[hueco].elem] = hueco;
            hueco = hijo; hijo = 2*hueco;
        } else break;
    }
    array[hueco] = mov;
    posiciones[array[hueco].elem] = hueco;
  }
```

Montículo con prioridades variables

```
public:
 void push(int e, T const& p) {
   if (posiciones.at(e) != 0)
       throw std::invalid_argument("Elementos repetidos.");
   else {
       array.push_back({e, p});
       posiciones[e] = array.size() - 1;
       flotar(array.size() - 1);
   }
 }

private:
 void flotar(int i) {
   Par mov = array[i];
   int hueco = i;
   while (hueco != 1 && antes(mov.prioridad, array[hueco/2].prioridad)) {
      array[hueco] = array[hueco/2];
      posiciones[array[hueco].elem] = hueco;
      hueco /= 2;
   }
   array[hueco] = mov;
   posiciones[array[hueco].elem] = hueco;
 }
```

Montículo con prioridades variables

```
void update(int e, T const& p) {
    int i = posiciones.at(e);
    if (i == 0) // el elemento e se inserta por primera vez
        push(e, p);
    else {
        array[i].prioridad = p;
        if (i != 1 && antes(array[i].prioridad, array[i/2].prioridad))
            flotar(i);
        else // puede hacer falta hundir a e
            hundir(i);
    }
}
```