Colas con prioridad y montículos

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Septiembre 2015

Bibliografía

- N. Martí Oliet, Y. Ortega Mallén y A. Verdejo. Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos. Segunda edición, Garceta, 2013.
  Capítulo 8
- F. M. Carrano y T. Henry. *Data Abstraction & Problem Solving with C++: Walls and Mirrors*. Sixth edition. Pearson, 2013.
  Capítulo 17
- M. A. Weiss. *Data Structures and Algorithm Analysis in Java*. Third edition. Addison-Wesley, 2012.
  Capítulo 6

Colas con 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 con 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 con prioridad de mínimos o de máximos, respectivamente.
- Para facilitar la presentación de las propiedades de la estructura de cola con prioridad, los elementos se identifican con su prioridad, de forma que el orden total es sobre elementos.

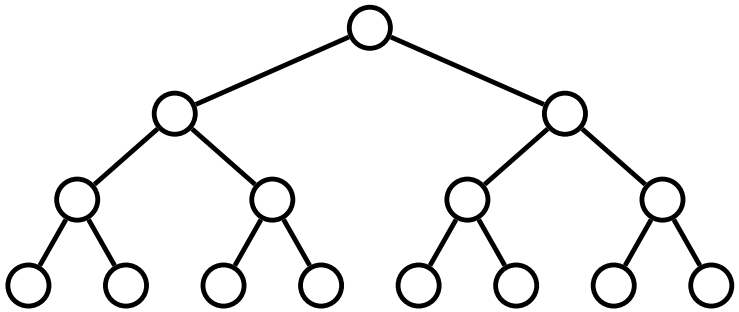
Colas con prioridad

El TAD de las colas con prioridad contiene las siguientes operaciones:

- crear una cola con prioridad vacía,
- añadir un elemento,
- consultar el primer elemento (el elemento más prioritario),
- eliminar el primer elemento, y
- determinar si la cola con prioridad es vacía.

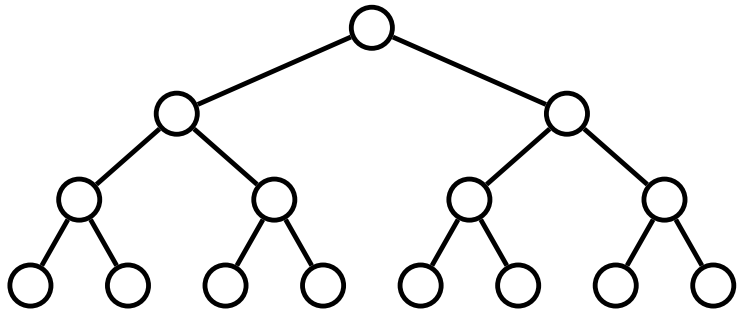
Árboles completos y semicompletos

- 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.

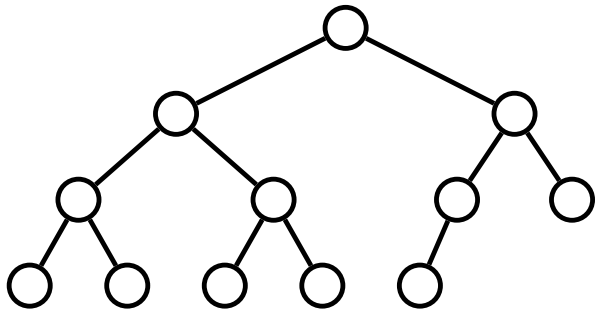


Árboles completos y semicompletos

- 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.



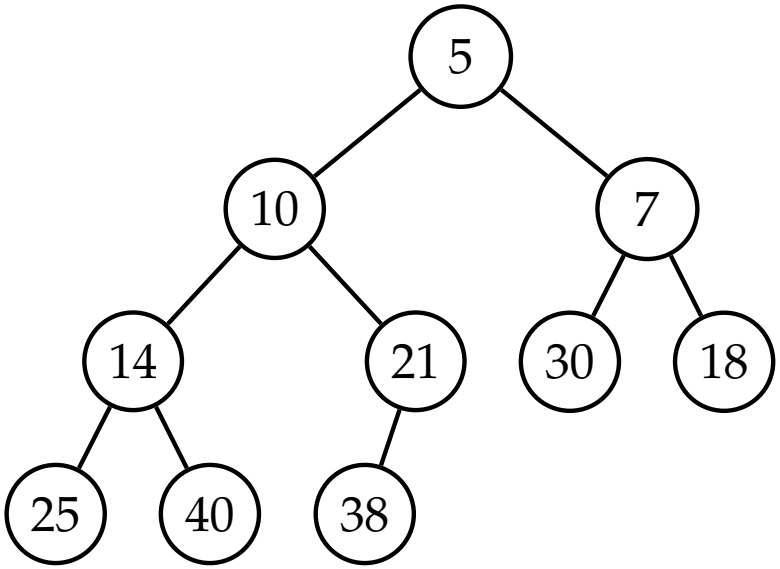
- Un árbol binario de altura h es semicompleto si o bien es completo o tiene vacantes una serie de posiciones consecutivas del nivel h empezando por la derecha, de tal manera que al rellenar dichas posiciones con nuevas hojas se obtiene un árbol completo.



Árboles completos y semicompletos

Montículos

- Un montículo 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.



Propiedades

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

Propiedades

- Un árbol binario completo de altura $h \geq 1$ 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 $2 * 2^{i-1} = 2^i = 2^{(i+1)-1}$.

Propiedades

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

Propiedades

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

  Las hojas son los nodos en el último nivel h.

Propiedades

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

  Las hojas son los nodos en el último nivel h.

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

Propiedades

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

  Las hojas son los nodos en el último nivel h.

- Un árbol binario completo de altura $h \geq 0$ 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 $\lfloor \log n \rfloor + 1$.

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:

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

  Tomando logaritmos en base 2

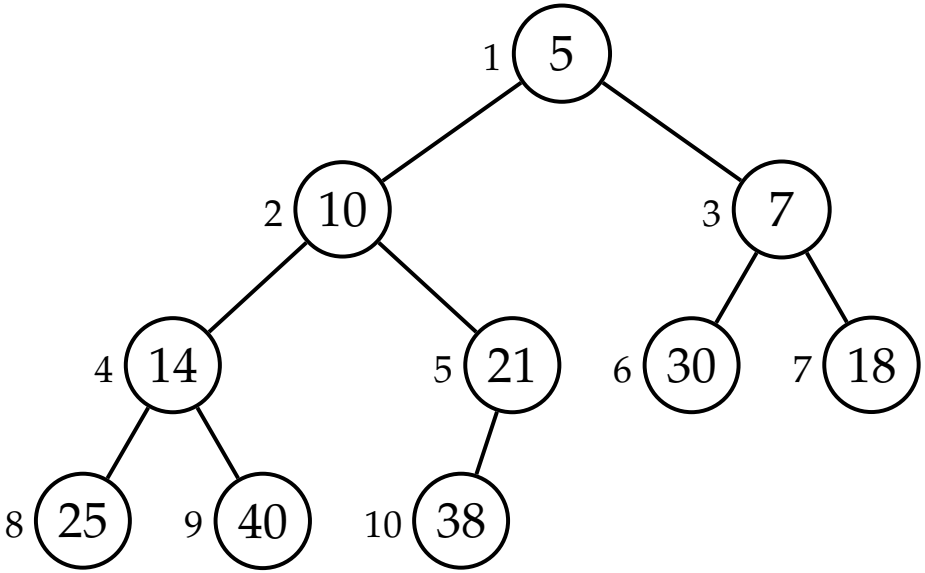
  $$\log(2^{h-1}) \leq \log n \leq \log(2^h - 1) < \log(2^h);$$

  equivalentemente,

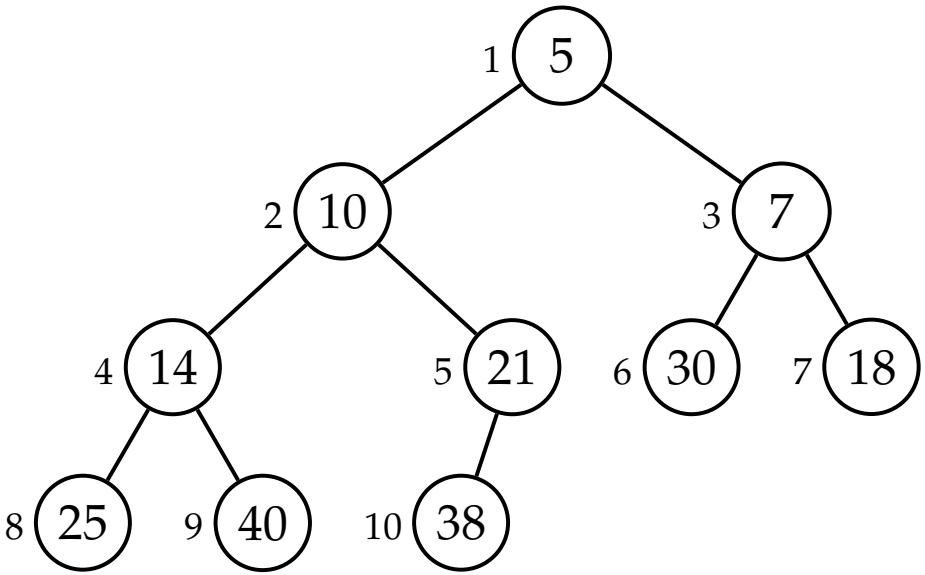
  $$h - 1 \leq \log n < h,$$

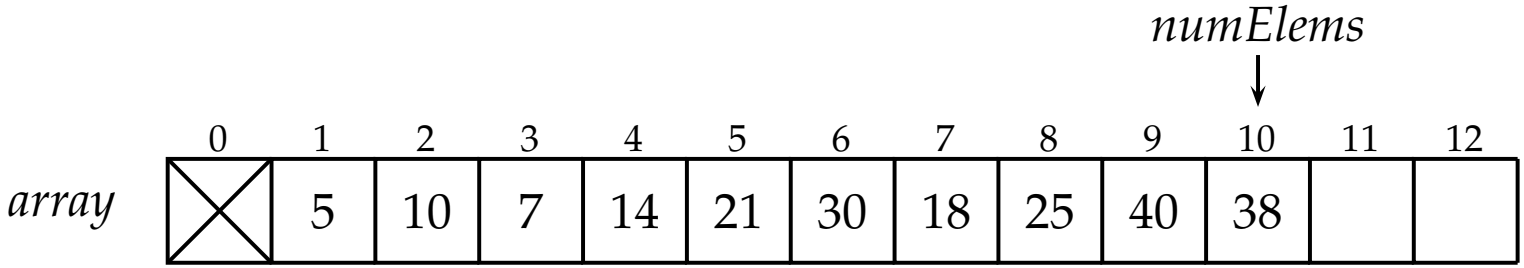
  es decir, $h - 1 = \lfloor \log n \rfloor$ y de aquí $h = \lfloor \log n \rfloor + 1$.

Implementación de montículos



Implementación de montículos



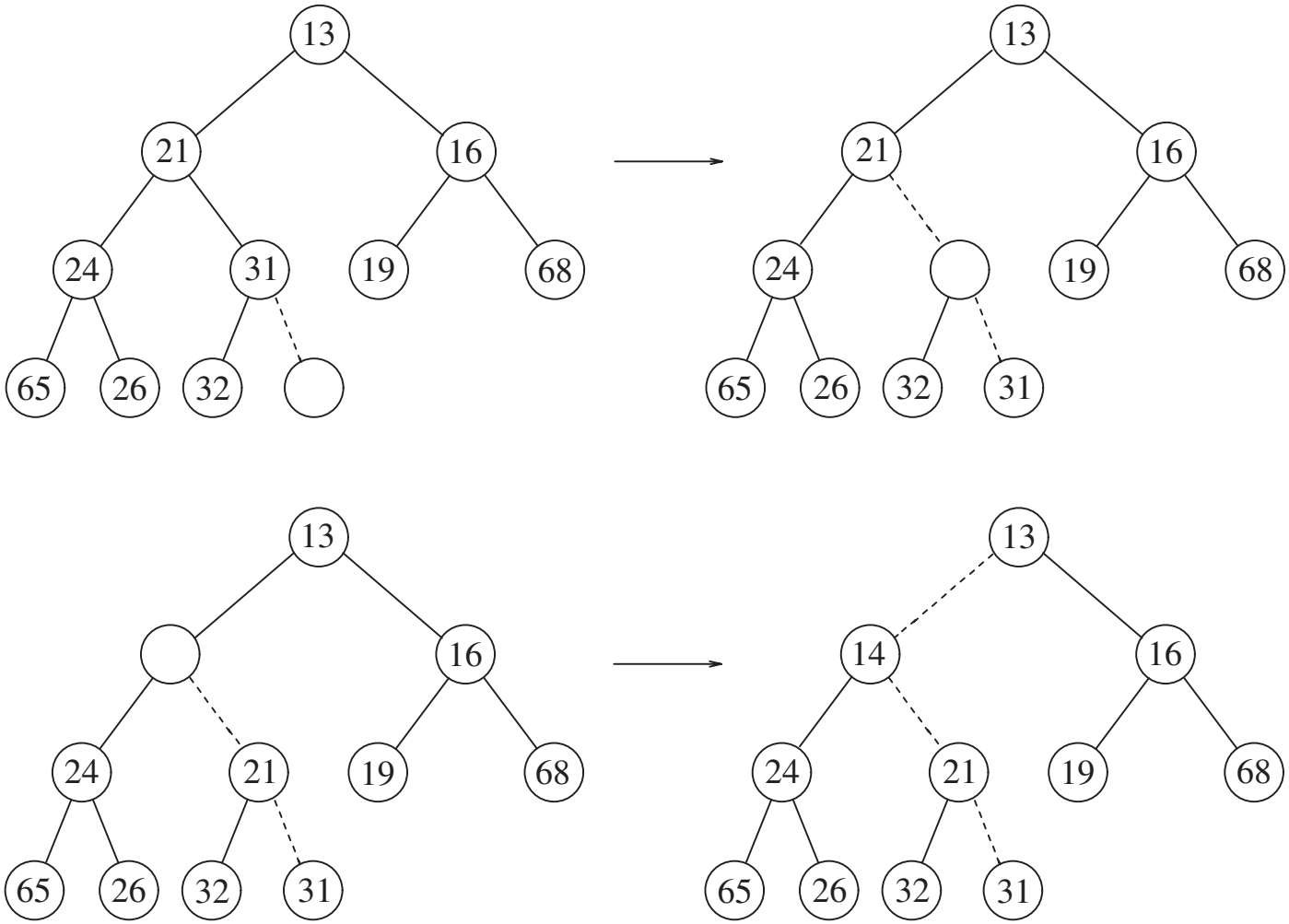


Implementación de las colas con prioridad mediante montículos

```
template <typename T = int, typename Comparator = std::less<T>>
class PriorityQueue {

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

  /** Número de elementos en el montículo */
  size_t numElems;

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

public:
   /** Constructor */
   PriorityQueue(size_t t = TAM_INICIAL, Comparator c = Comparator()) :
     array(t+1), numElems(0), antes(c) {};  // índices de v de 1 a t
```

Implementación de las colas con prioridad mediante montículos

- Inserción del 14:



Implementación de las colas con prioridad mediante montículos

```
public:
  void push(T const& x) {
    if (numElems == array.size() - 1)    // array lleno
        array.resize(array.size() * 2);  // se aumenta la capacidad
    ++numElems;
    array[numElems] = x;
    flotar(numElems);
  }

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

Implementación de las colas con prioridad mediante montículos

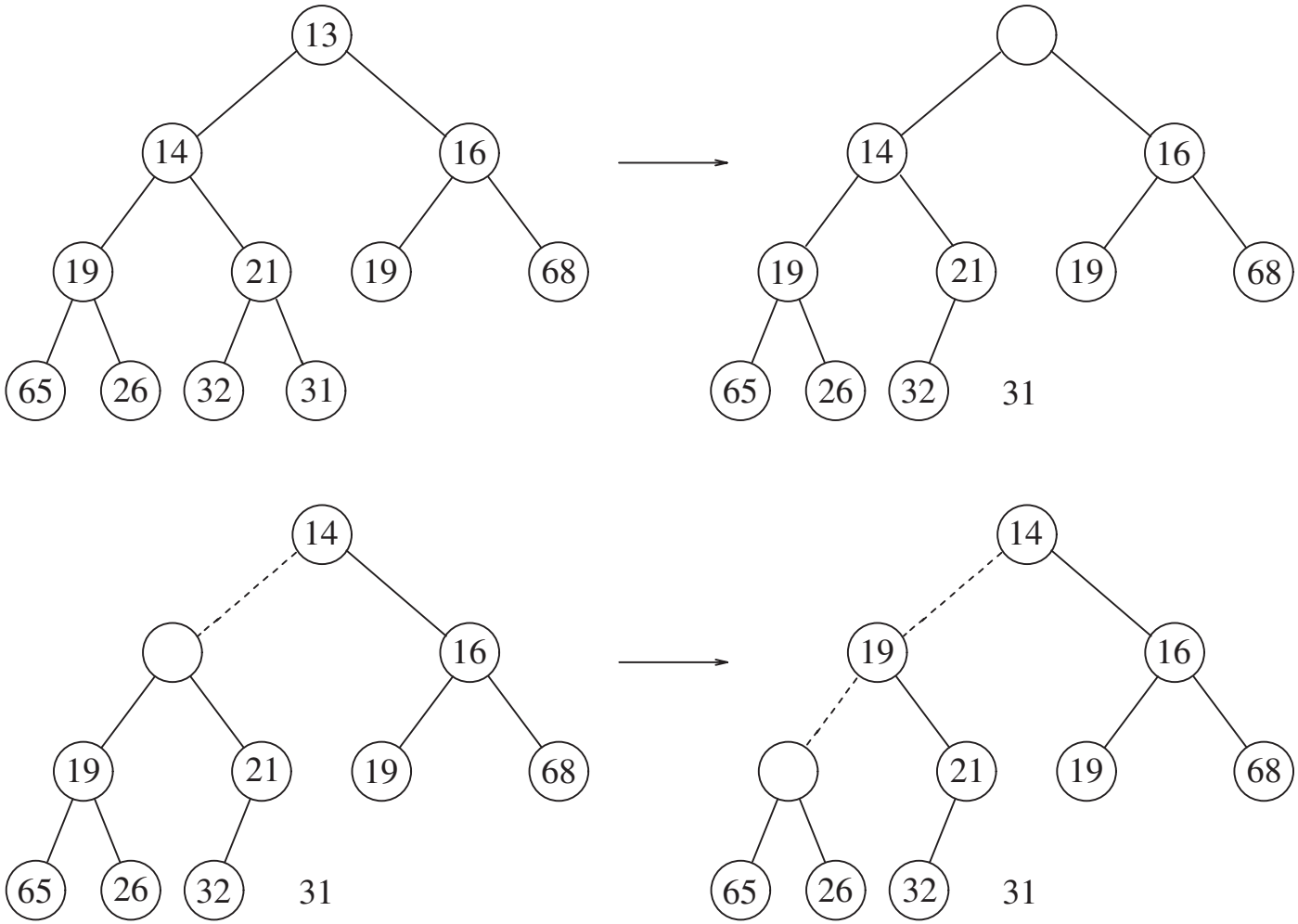
```
public:
  void push(T && x) {
    if (numElems == array.size() - 1)    // array lleno
        array.resize(array.size() * 2);  // se aumenta la capacidad
    ++numElems;
    array[numElems] = std::move(x);
    flotar(numElems);
  }

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

Implementación de las colas con prioridad mediante montículos

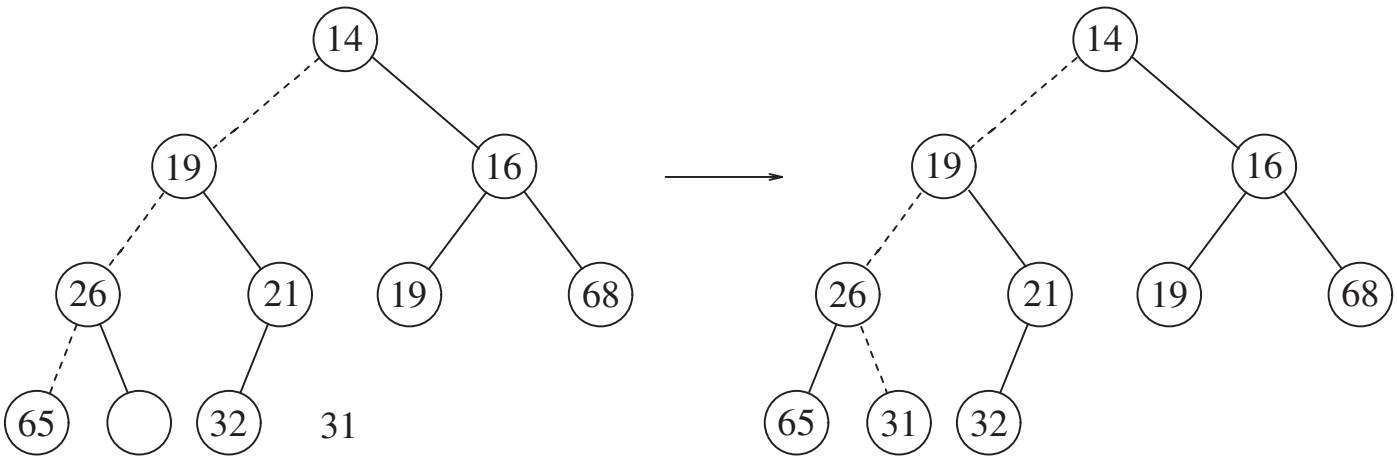
```
size_t size() const {
   return numElems;
}

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

T const& top() const {
    if (empty()) throw std::domain_error("Error cola vacía.");
    else return array[1];
}
```

Implementación de las colas con prioridad mediante montículos

- Eliminación del primero:



Implementación de las colas con prioridad mediante montículos



Implementación de las colas con prioridad mediante montículos

```
void pop() {
    if (empty()) throw std::domain_error("Error cola vacía");
    else {  array[1] = std::move(array[numElems]);
            --numElems;
            hundir(1);
    }
}

void hundir(size_t n) {
   T elem = std::move(array[n]);
   size_t hueco = n;
   size_t hijo = 2*hueco; // hijo izquierdo, si existe
   while (hijo <= numElems)  {
       // cambiar al hijo derecho si existe y va antes que el izquierdo
       if (hijo < numElems && antes(array[hijo + 1], array[hijo]))
           ++hijo;
       // flotar el hijo si va antes que el elemento hundiéndose
       if (antes(array[hijo], elem)) {
           array[hueco] = std::move(array[hijo]);
           hueco = hijo; hijo = 2*hueco;
       } else break;
   }
   array[hueco] = std::move(elem);
}
```

Resumen de costes de implementaciones de colas con 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 (auto j = 2; j <= numElems; ++j) {
      flotar(j);
   }
}
```

Convertir un vector en un montículo

```
void monticulizar1() {
   for (auto j = 2; j <= numElems; ++j) {
      flotar(j);
   }
}
```

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 (auto j = 2; j <= numElems; ++j) {
      flotar(j);
   }
}
```

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 (auto j = numElems/2; j >= 1; --j)
      hundir(j);
}
```

Convertir un vector en un montículo

```
void monticulizar2() {
   for (auto j = numElems/2; j >= 1; --j)
      hundir(j);
}
```

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 (auto j = numElems/2; j >= 1; --j)
      hundir(j);
}
```

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	$h-1$

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

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

Heapsort

Método de ordenación basado en la utilización de un montículo.

```
void heapsort_abstracto(std::vector<int> & v) {
    PriorityQueue<int> colap(v.size());
    for (auto e : v)
        colap.push(e);
    for (auto 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

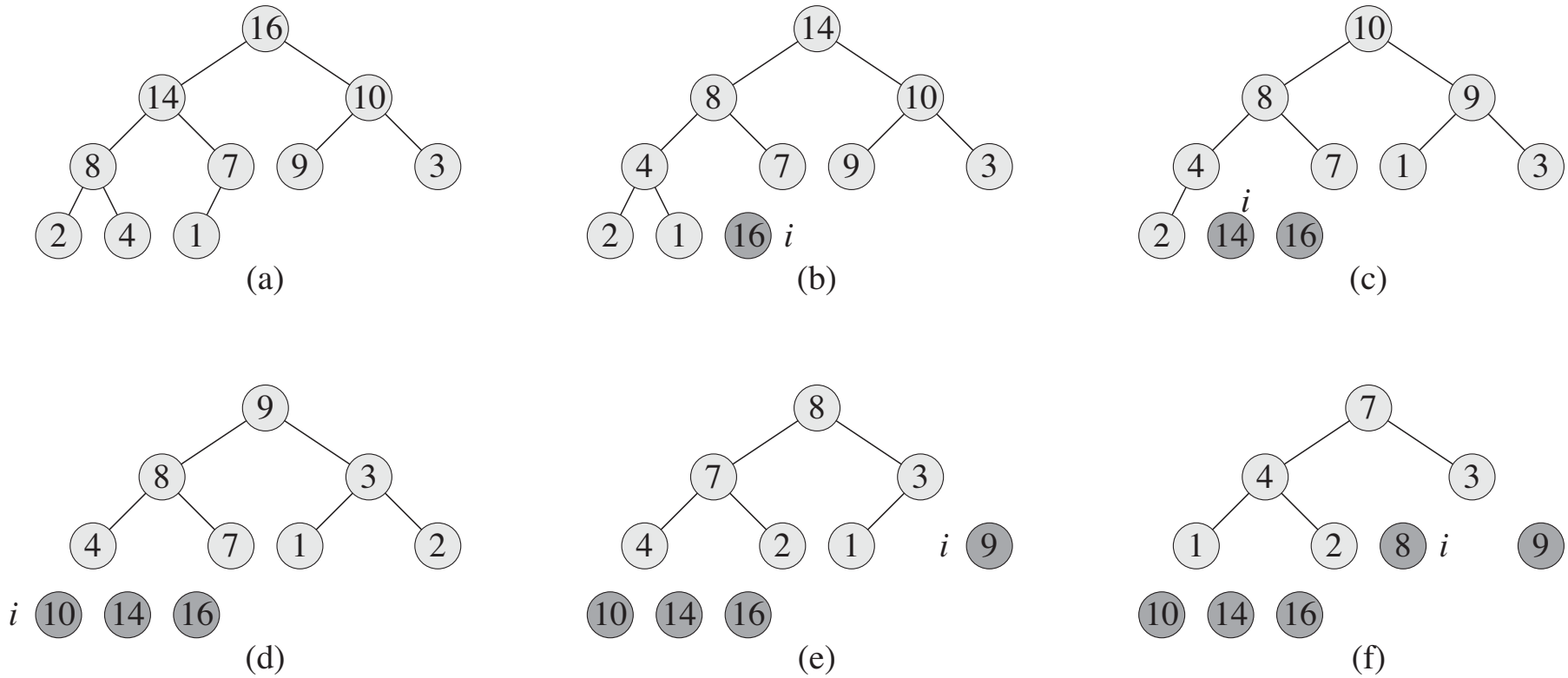
- Podemos ahorrarnos este espacio adicional si utilizamos el mismo vector para representar el montículo auxiliar.
- Primero el vector se convierte en un montículo.
- Después se recorren las posiciones del vector de derecha a izquierda extrayendo cada vez el primero del montículo para colocarlo al principio 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 (auto i = v.size() - 1; i > 0; --i) {
        std::swap(v[i], v[0]);
        hundir_max(v, i, 0, cmp);
    }
}
```

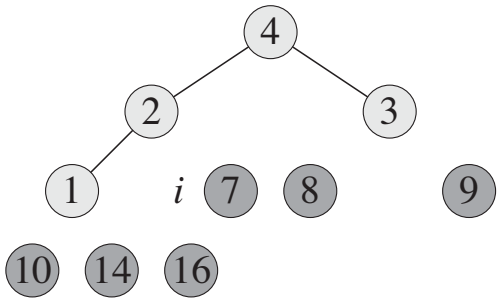
Heapsort

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

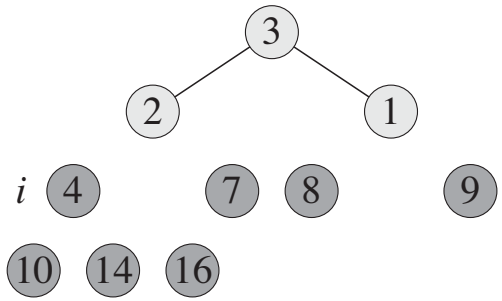
Heapsort


16
14
10
8
7
9
3
2
4
1
(a)
14
8
10
4
7
9
3
2
1
16
i
(b)
10
8
9
4
7
1
3
i
2
14
16
(c)
9
8
3
4
7
1
2
i
10
14
16
(d)
8
7
3
4
2
1
i
9
10
14
16
(e)
7
4
3
1
2
8
i
9
10
14
16
(f)

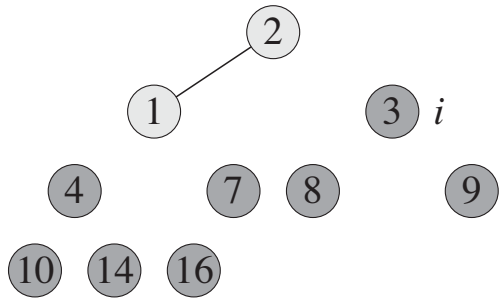
Heapsort



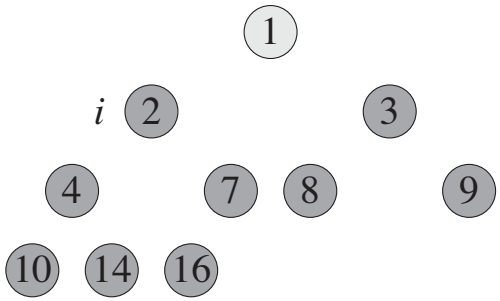
(g)



(h)



(i)



(j)

(k)

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 a, string b) {
        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
```

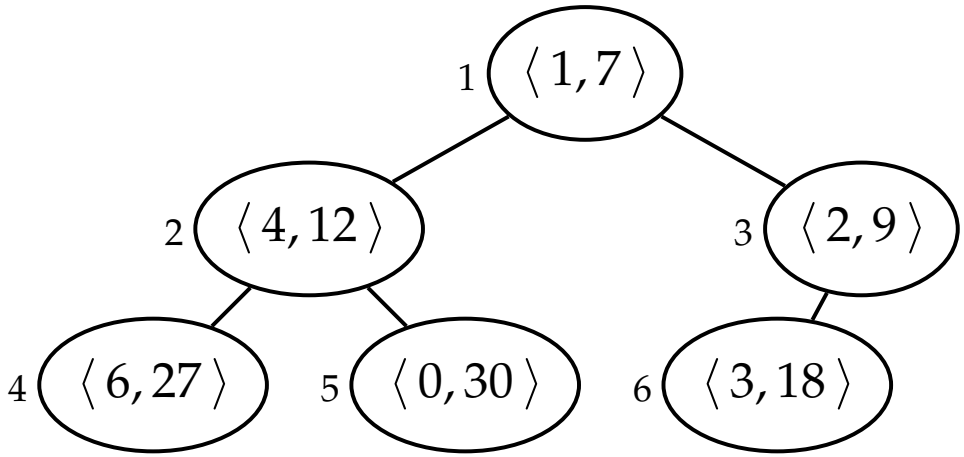
```
class ComparaString {
public:
    bool operator()(string a, string b) {
        return aMinusculas(a) < aMinusculas(b);
    }
};

heapsort(datos, ComparaString());

abeja gato leon Lobo perro Zorro
```

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

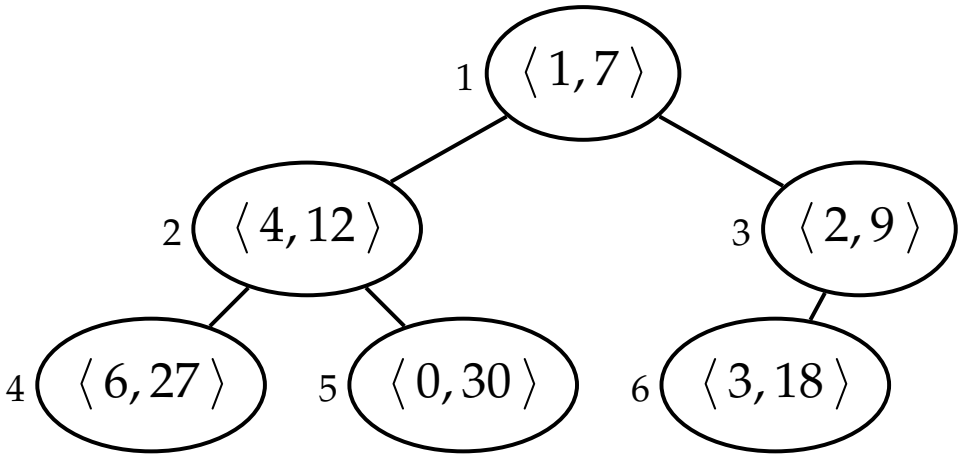
Montículo con prioridades variables asociadas a elementos con identificador

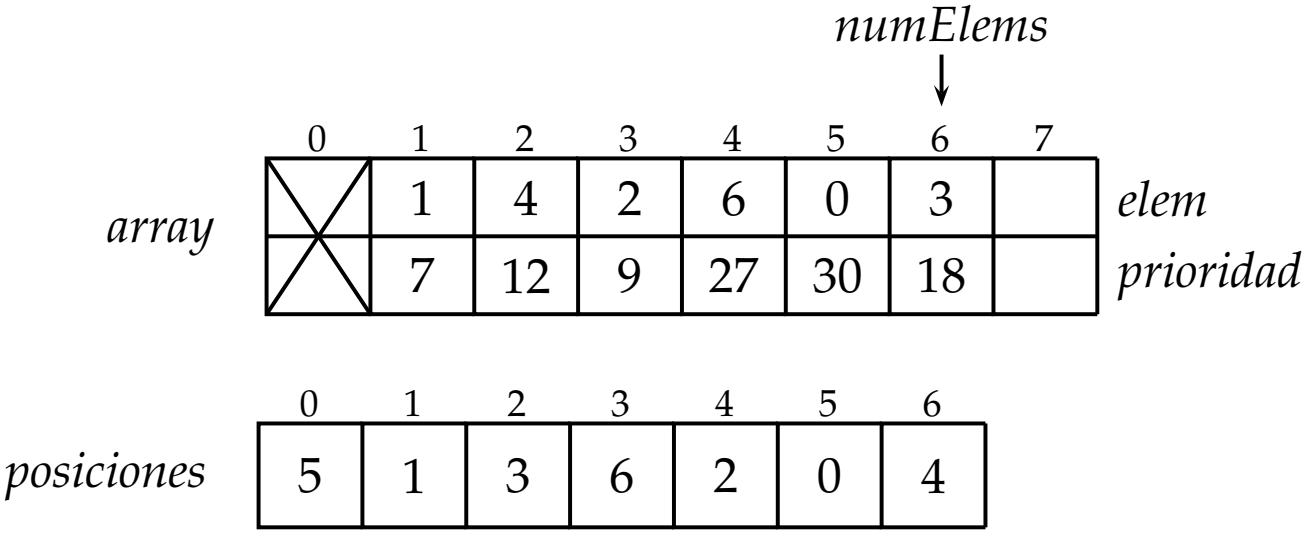
- Queremos una cola con prioridad que almacene elementos en el intervalo $[0..N)$ cada uno con una prioridad asociada.
- Y queremos poder modificar la prioridad asociada a un elemento en tiempo logarítmico.
- Utilizaremos un montículo de pares de la forma $\langle elem, prioridad \rangle$ donde $elem$ 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

```
template <typename T = int, typename Comparator = std::less<T>>
class IndexPQ {
public:
    // registro para las parejas < elem, prioridad >
    struct Par {
        size_t 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<size_t> posiciones;   // un 0 indica que el elemento no está

    /** Número de elementos en el montículo. */
    size_t numElems;

    /** 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(size_t t, Comparator c = Comparator()) :
     array(t+1), posiciones(t,0), numElems(0), antes(c) {};

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

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

Montículo con prioridades variables

```
void hundir(size_t n) {
  Par parmov = std::move(array[n]);
  size_t hueco = n;
  size_t hijo = 2*hueco; // hijo izquierdo, si existe
  while (hijo <= numElems)  {
      // cambiar al hijo derecho de i si existe y va antes que el izquierdo
      if (hijo < numElems &&
           antes(array[hijo + 1].prioridad, array[hijo].prioridad))
          ++hijo;
      // flotar el hijo si va antes que el elemento hundiéndose
      if (antes(array[hijo].prioridad, parmov.prioridad)) {
          array[hueco] = std::move(array[hijo]);
          posiciones[array[hueco].elem] = hueco;
          hueco = hijo; hijo = 2*hueco;
      } else break;
  }
  array[hueco] = std::move(parmov);
  posiciones[array[hueco].elem] = hueco;
}
```

Montículo con prioridades variables

```
void push(size_t e, T const& p) {
  if (posiciones.at(e) != 0) throw std::invalid_argument("Elementos repetidos.");
  else {
      ++numElems;
      array[numElems].elem = e; array[numElems].prioridad = p;
      posiciones[e] = numElems;
      flotar(numElems);
  }
}

void flotar(size_t n) {
   Par parmov = std::move(array[n]);
   size_t hueco = n;
   while (hueco != 1 && antes(parmov.prioridad, array[hueco/2].prioridad)) {
       array[hueco] = std::move(array[hueco/2]);
       posiciones[array[hueco].elem] = hueco;
       hueco /= 2;
   }
   array[hueco] = std::move(parmov);
   posiciones[array[hueco].elem] = hueco;
}
```

Montículo con prioridades variables

```
void update(size_t e, T const& p) {
   auto 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);
   }
}
```