Colas con prioridad y montículos

Alberto Verdejo

Dpto. de Sistemas Informáticos y Computación

Universidad Complutense de Madrid

Octubre 2013

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 J. J. Prichard. *Data Abstraction and Problem Solving with C++*. Third edition. Addison-Wesley, 2002.
  Capítulo 11
- 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.

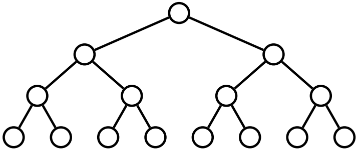
Colas con prioridad

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

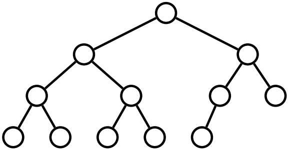
- crear una cola con prioridad vacía,
- añadir un elemento,
- consultar el primer elemento (el elemento más prioritario),
- eliminar el primer elemento, y
- determinar si la cola con prioridad es vacía.

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.



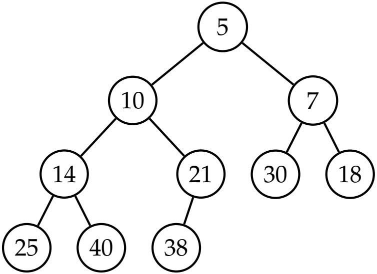
- Un árbol binario de altura h es semicompleto si o bien es completo o tiene vacantes una serie de posiciones consecutivas del nivel h empezando por la derecha, de tal manera que al rellenar dichas posiciones con nuevas hojas se obtiene un árbol completo.



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.

  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 $\lfloor \log n \rfloor + 1$.

  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

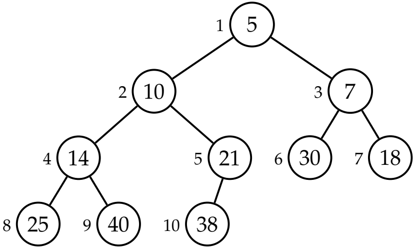
  $$\log(2^{h-1}) \leq \log n \leq \log(2^h - 1) < \log(2^h);$$

  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



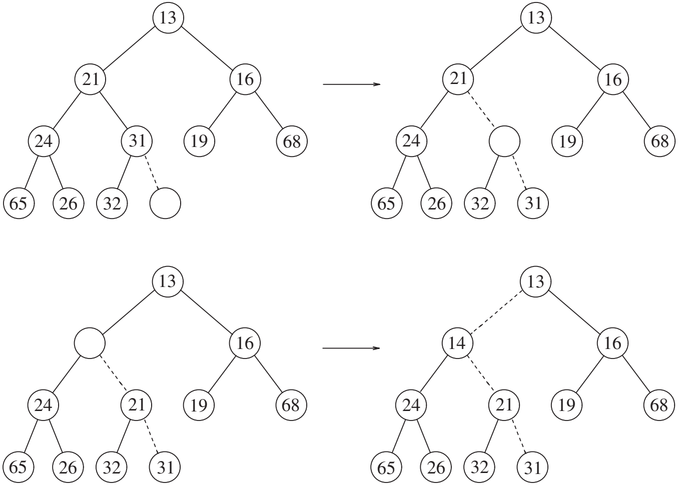
1	2	3	4	5	6	7	8	9	10
5	10	7	14	21	30	18	25	40	38

Implementación de las colas con prioridad mediante montículos

```
template <class T, bool(*antes)(const T &, const T &)>
class ColaPrio {

private:

   /** Puntero al array que contiene los datos. */
   T* v;

   /** Tamaño del vector v. */
   unsigned int tam;

   /** Número de elementos reales guardados. */
   unsigned int numElems;

public:
   /** Constructor; operación ColaPVacia */
   ColaPrio(int t = TAM_INICIAL) :
      v(new T[t+1]), tam(t), numElems(0) {};  // 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

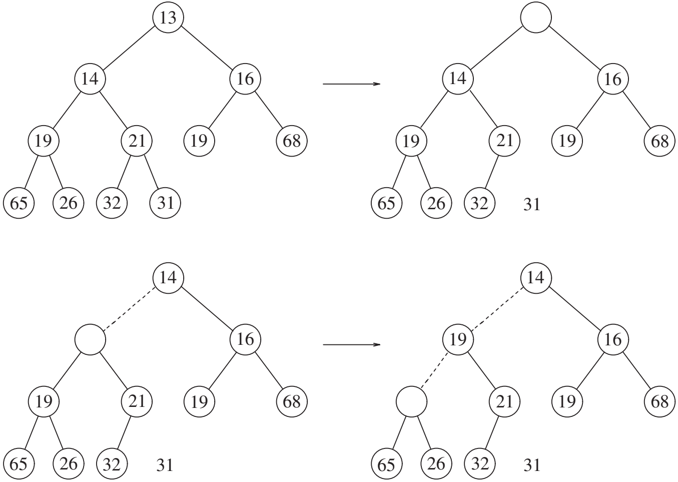
```
void inserta(const T& x) {
   if (numElems == tam) throw EColaPrLlena();
   else {
      numElems++;
      v[numElems] = x;
      flotar(numElems);
   }
   return;
}

void flotar(unsigned int n) {
   unsigned int i = n;
   T elem = v[i];
   while ((i != 1) && antes(elem, v[i/2])) {
      v[i] = v[i/2];
      i = i/2;
   }
   v[i] = elem;
}
```

Implementación de las colas con prioridad mediante montículos

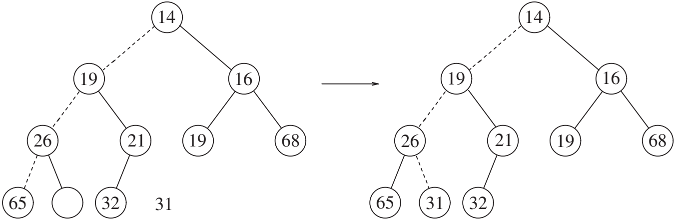
```
bool esVacia() const {
   return (numElems == 0);
}

const T& primero() const {
   if (numElems == 0) throw EColaPrVacia("No existe el primero");
   else return v[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 quitaPrim() {
   if (numElems == 0) throw EColaPrVacia("Imposible eliminar primero");
   else {
      v[1] = v[numElems];
      numElems--;
      hundir(1);
   }
}

void hundir(unsigned int n) {
   unsigned int i = n;
   T elem = v[i];
   unsigned int m = 2*i; // hijo izquierdo de i, si existe
   while (m <= numElems)  {
      // cambiar al hijo derecho de i si existe y va antes que el izquierdo
      if ((m < numElems) && (antes(v[m + 1], v[m])))
         m = m + 1;
      // flotar el hijo m si va antes que el elemento hundiendose
      if (antes(v[m], elem)) {
         v[i] = v[m]; i = m; m = 2*i;
      } else break;
   }
   v[i] = 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(unsigned int 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(unsigned int 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	$h-1$

$$
\begin{aligned}
\sum_{i=1}^{h-1}(h-i)2^{i-1} &= \sum_{j=2}^{h}(j-1)2^{h-j} < \sum_{j=1}^{h} j2^{h-j} = 2^h \sum_{j=1}^{h} \frac{j}{2^j} \\
&= 2^h\left(2 - \frac{h+2}{2^h}\right) \leq 2^{h+1} = 2^{\lfloor \log N \rfloor + 2} \in O(N)
\end{aligned}
$$

Heapsort

Método de ordenación basado en la utilización de un montículo.

```
bool menor(const int& a, const int& b) {
   return a < b;
}

void heapsort_abstracto(int V[], unsigned int N) {
   ColaPrio<int, menor> colap(N);
   for (unsigned int i = 0; i < N; ++i)
      colap.inserta(V[i]);
   for (unsigned int i = 0; i < N; ++i) {
      V[i] = colap.primero();
      colap.quitaPrim();
   }
}
```

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.

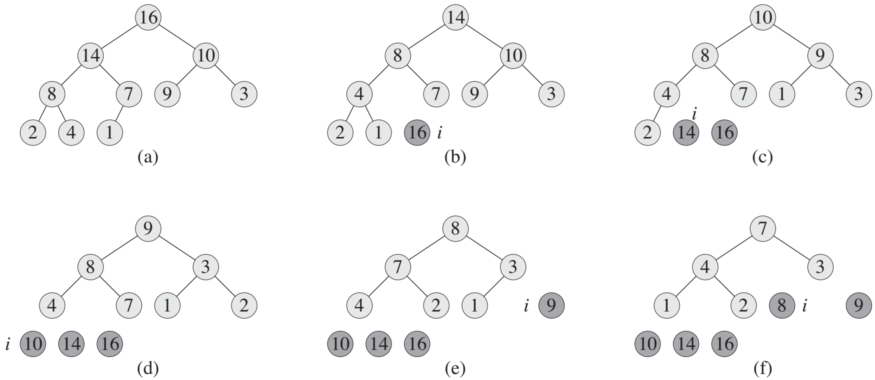
V	montículo	ordenados

```
void heapsort(int V[], unsigned int N) {
   // monticulizar
   for (int i = (N - 1) / 2; i >= 0; --i)
      hundir_max(V, N, i);
   // ordenar
   for (int i = N - 1; i > 0; --i) {
      int aux = V[i];  V[i] = V[0];  V[0] = aux;
      hundir_max(V, i, 0);
   }
}
```

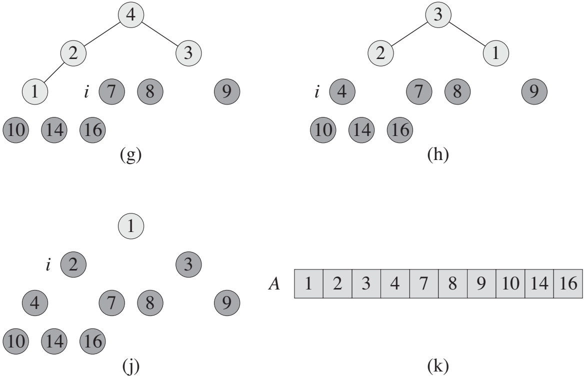
Heapsort

```
void hundir_max(int V[], unsigned int N, unsigned int j) {
   // indices de V de 0 a N-1
   unsigned int i = j;
   int elem = V[i];
   unsigned int m = 2*i+1; // hijo izquierdo de i, si existe
   while (m < N) {
      // cambiar al hijo derecho de i si existe y va antes que el izquierdo
      if ((m + 1 < N) && (V[m + 1] > V[m]))
         m = m + 1;
      // flotar el hijo m si va antes que el elemento hundiéndose
      if (V[m] > elem) {
         V[i] = V[m]; i = m; m = 2*i+1;
      } else break;
   }
   V[i] = 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)

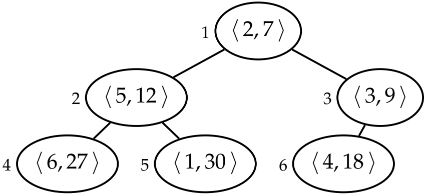
Heapsort



Montículo con prioridades variables

- Queremos utilizar un montículo para almacenar pares de la forma $\langle \mathit{elem}, \mathit{prioridad} \rangle$ donde *elem* es un número natural en el intervalo $1..N$, las prioridades son números reales, y el *elem* de todos los pares es diferente.
- El orden entre los pares viene inducido por el orden entre las prioridades.
- Queremos poder modificar la prioridad asociada a un elemento en el montículo y mantener las propiedades de la estructura en tiempo logarítmico.

Montículo con prioridades variables



último ↓ (6)

V	1	2	3	4	5	6	7
elem	2	5	3	6	1	4	
prioridad	7	12	9	27	30	18	

	1	2	3	4	5	6	7
posiciones	5	1	3	6	2	4	0

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

Montículo con prioridades variables

```
template <class T>
struct Par {
   unsigned int elem;
   T prioridad;
};

template <class T, bool(*antes)(const T &, const T &)>
class ColaPrioPares {
private:
   /** Puntero al array que contiene los datos (pares <elem, prio>). */
   Par<T>* v;
   /** Puntero al array que contiene las posiciones en v de los elementos. */
   unsigned int* posiciones;
   /** Tamaño del vector v. */
   unsigned int tam;
   /** Número de elementos reales guardados. */
   unsigned int numElems;
public:
   /** Constructor */
   ColaPrioPares(int t) :
     v(new Par<T>[t+1]), posiciones(new unsigned int[t+1]), tam(t), numElems(0){
     for(unsigned int i=1; i <= tam; i++)
        posiciones[i] = 0; // el elemento i no esta
   };
```

Montículo con prioridades variables

```
const Par<T>& primero() const {
   if (numElems == 0) throw EColaPrVacia("No se puede consultar el primero");
   else return v[1];
}

void quitaPrim() {
   if (numElems == 0) throw EColaPrVacia("Imposible eliminar primero");
   else {
      posiciones[v[1].elem] = 0; // para indicar que no está
      v[1] = v[numElems];
      posiciones[v[1].elem] = 1;
      numElems--;
      hundir(1);
   }
}
```

Montículo con prioridades variables

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

Montículo con prioridades variables

```
void inserta(unsigned int e, const T& p) {
   if (posiciones[e] != 0) throw EElemRepe();
   else if (numElems == tam) throw EColaPrLlena();
   else {
      numElems++;
      v[numElems].elem = e; v[numElems].prioridad = p;
      posiciones[e] = numElems;
      flotar(numElems);
   }
   return;
}


void flotar(unsigned int n) {
   unsigned int i = n;
   Par<T> parmov = v[i];
   while ((i != 1) && antes(parmov.prioridad, v[i/2].prioridad)) {
      v[i] = v[i/2]; posiciones[v[i].elem] = i;
      i = i/2;
   }
   v[i] = parmov; posiciones[v[i].elem] = i;
}
```

Montículo con prioridades variables

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