CAPÍTULO 3

Montículos

(Parte del Capítulo 3 del libro, *Algoritmos y estructuras de datos, con programas verificados en Dafny***, Ricardo Peña Marí, Garceta 2019)**

Los montículos[1] son implementaciones eficientes de un tipo de datos que aparece con frecuencia en la programación, llamado *cola de prioridad*. Dicho tipo tiene el siguiente comportamiento observable: al igual que en una cola FIFO, existe una operación de inserción, que incorpora nuevos elementos a la cola y operaciones que consultan y borran el primer elemento de la misma. Pero, a diferencia de las colas FIFO, aquí los elementos son recuperados en orden de prioridad, es decir, el primer elemento en salir es el de mayor prioridad. Normalmente se toma como prioridad el valor del elemento y se admite que cuanto menor es el valor tanto más alta es la prioridad. Este comportamiento intuitivo se da por ejemplo en la cola de urgencias de un hospital: los enfermos esperan a ser atendidos en el orden de importancia de su enfermedad, independientemente del orden en que se incorporaron al servicio.

Ejemplos de aplicación a la programación los proporcionan algunos algoritmos sobre grafos en los que las aristas se insertan de forma dinámica en un montículo y cada arista tiene asociado un valor numérico que representa el coste de la misma. El algoritmo va obteniendo las aristas del montículo de menor a mayor coste. En definitiva, los elementos se recuperan de menor a mayor y la cola se llama *de mínimos*. En consonancia con ello, llamaremos *min* a la operación que accede al primer elemento y *deleteMin* a la que lo suprime de la cola. Si se cambia el convenio, y los elementos se recuperan en orden decreciente, la cola se llamaría de máximos y las respectivas

[1] En inglés, *heap*.

operaciones, *max* y *deleteMax*. A diferencia de los árboles de búsqueda vistos en el capítulo 2, aquí se admiten valores repetidos. El modelo matemático de una cola de prioridad consiste en un multiconjunto de elementos, con la propiedad adicional de que existe una relación de orden entre los elementos que permite preguntar por el mínimo de ellos.

La implementación ingenua de una cola de prioridad de n elementos, mediante una lista o un vector, da lugar a costes en $O(n)$, o bien en la operación de inserción, o bien en la que busca el mínimo, dependiendo, respectivamente, de si la estructura está ordenada o no. Como veremos, los montículos están basados en árboles de diferentes tipos y dan lugar a costes mucho mejores. En particular, consultar el mínimo siempre tendrá un coste en $O(1)$, e insertar o borrar un elemento tendrán un coste máximo en $O(\log n)$.

3.1. Montículos de Williams

Fueron ideados por J. W. J. Williams en 1964. La implementación de una cola de prioridad mediante un montículo de Williams se basa en un árbol binario implementado a su vez sobre un vector. Dicho árbol binario satisface la llamada *propiedad de montículo* que puede enunciarse así:

Definición 3.1 *Un árbol binario de elementos satisface la* propiedad de montículo *si*

- *o bien es el árbol vacío,*
- *o bien su elemento raíz es menor o igual que el resto de los elementos del árbol y sus subárboles izquierdo y derecho son montículos.*

□

Para implementarlo sobre un vector, se exige que el árbol binario sea *casi completo* y se utiliza la representación vectorial para árboles completos y casi completos. Un árbol casi completo tiene el máximo de nodos posibles en cada nivel, excepto quizás en el último nivel en el que pueden faltar 0 o más nodos consecutivos contados desde el final del mismo. En dicha representación, los nodos del árbol se numeran por niveles desde la raíz y de izquierda a derecha dentro de cada nivel, sin dejar huecos en la numeración. A continuación, se almacena el elemento de cada nodo en la posición del vector que corresponda con el número asignado al nodo. En la figura 3.1 se muestra un ejemplo de dicha disposición.

Un montículo de Williams con capacidad para n elementos se representa entonces mediante un vector de tamaño n y un natural *next*, con $0 \leq next \leq n$, que indica la primera posición libre del vector. Cuando $next = 0$, el montículo está vacío y siempre que $next \leq n-1$, el montículo admite una nueva inserción. Con esta representación,

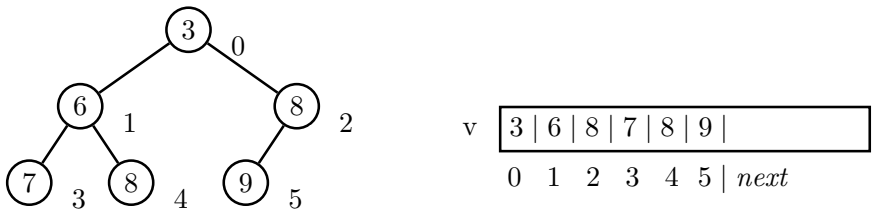

Figura 3.1: Ejemplo de montículo de Williams.

los elementos hijos de $v[i]$, si existen, son $v[2i+1]$ y $v[2i+2]$ y el padre de $v[i]$, si $i > 0$, es el elemento $v[(i-1)/2]$. En la figura 3.2 se muestra la definición de la clase que implementa esta idea y un predicado `isHeap` que especifica el invariante de la representación. El constructor anónimo crea un montículo vacío, con capacidad para un tamaño `size` que recibe como argumento.

En base a esta implementación, la creación de un montículo vacío y la operación *min*, que simplemente devolvería $v[0]$, tienen coste constante. La operación *insert* sigue la siguiente secuencia:

- Si $next < size$, añade el nuevo elemento en la posición *next* del vector, posición que corresponde a la hoja más a la derecha posible del último nivel del árbol.
- Incrementa en uno el valor de *next*. De este modo, el árbol conserva la propiedad de ser casi completo.
- Para preservar también la propiedad de montículo, se hace *flotar* el elemento, intercambiándolo repetidamente con su padre hasta conseguir que, o bien el elemento sea mayor o igual que su padre, o bien el elemento se convierta en la raíz.

Si n es el número de elementos del montículo, este proceso puede llevarse a cabo en un tiempo en $\Theta(\log n)$, ya que el acceso al padre se realiza en tiempo constante y, en el peor caso, ha de recorrerse el camino completo desde la hoja a la raíz. La implementación en Dafny de ambas operaciones `insert` y `float` se muestra en la figura 3.3.

El invariante del bucle **while** de `float` expresa que el predicado `isHeap` se satisface para todos los elementos del vector, excepto quizás para la relación entre $v[j]$ y su padre, propiedad que es un debilitamiento de la postcondición. Al terminar el bucle, o bien la relación entre $v[j]$ y su padre es la correcta, o bien $v[j]$ no tiene padre.

```
1  class Williams_heap {
2      var v: array<int>;
3      var next: nat;
4
5      predicate isHeap ()
6      reads this, v
7      {
8          0 ≤ next ≤ v.Length ∧
9          ∀ i | 0 < i < next • v[(i-1)/2] ≤ v[i]
10     }
11     constructor (size: nat)
12     ensures isHeap()
13     {
14         v := new int [size];
15         next := 0;
16     }
17     ...
18 }
```

Figura 3.2: Clase e invariante que implementan un montículo de Williams.

La operación *deleteMin* ha de suprimir la raíz del montículo. Como resultado, se obtienen dos montículos. Para unirlos de nuevo, se realiza la siguiente secuencia de operaciones:

- Siempre que se cumpla $next > 0$, se asigna a $v[0]$ el contenido de $v[next - 1]$. Ello equivale a convertir en raíz la hoja más a la derecha del último nivel.
- Se decrementa en uno el valor de *next*. Con ello se ha conseguido un árbol casi completo con los elementos remanentes del montículo.
- Para preservar la propiedad de montículo, se *hunde* el elemento raíz, intercambiándolo repetidamente con el menor de sus hijos, siempre que estos existan, hasta conseguir que, o bien el elemento es menor o igual que sus dos hijos, o bien el elemento es una hoja.

Esta secuencia de operaciones puede realizarse también en un tiempo en $\Theta(\log n)$. En la figura 3.4 se proporciona el código Dafny de las operaciones *deleteMin* y *sink*.

El invariante del bucle **while** de `sink` es también un debilitamiento de la postcondición: la propiedad de montículo se satisface en el vector, excepto quizás para la relación que guardan el elemento j y sus dos hijos. Cuando el bucle termina, o bien $v[j]$ es una hoja, o bien guarda la relación de montículo con sus dos hijos.

```
1       method insert (val:int)
2       requires isHeap()
3       requires next < v.Length
4       modifies this, v
5       ensures isHeap()
6       {
7           v[next] := val; next := next + 1;
8           float ();
9       }
10      method float()
11      requires 0 < next ≤ v.Length
12      requires ∀ i | 0 < i < next-1 • v[(i-1)/2] ≤ v[i]
13      modifies v
14      ensures isHeap()
15      {
16          var j := next-1;
17          while j > 0 ∧ v[(j-1)/2] > v[j]
18              invariant 0 ≤ j ≤ next-1 < v.Length
19              invariant ∀ i | 0 < i < next • i ≠ j ⟹
20                        v[(i-1)/2] ≤ v[i]
21          {
22              v[(j-1)/2], v[j] := v[j], v[(j-1)/2];
23              j := (j-1)/2;
24          }
25      }
```

Figura 3.3: Implementación de *insert* y *float* en un montículo de Williams.

3.1.1. Ordenación por el método del montículo

Las propiedades de los montículos representados como vectores los hacen interesantes para basar en ellos el algoritmo de ordenación de vectores conocido como *ordenación por el método del montículo*, algoritmo de Williams (1964) o *heapsort*. La idea del mismo es muy sencilla: dado un vector de n elementos, en una primera fase se construye con ellos un montículo de mínimos. A continuación, se extrae sucesivamente el mínimo del montículo hasta dejar este vacío, y los elementos extraídos del montículo se van copiando en posiciones consecutivas del vector, quedando este finalmente ordenado. Esta versión, cuya implementación puede verse en la figura 3.5, necesita un espacio adicional en $\Theta(n)$ para la variable `queue` de tipo montículo, siendo n la longitud del vector a. Dado que los costes de *insert* y de *deleteMin* están en $\Theta(\log m)$, siendo m el número de elementos del montículo en cada momento, el

```
1  method deleteMin ()
2  requires isHeap()
3  requires 0 < next
4  modifies this, v
5  ensures isHeap()
6  {
7     v[0] := v[next-1]; next := next - 1;
8     sink (0, next);
9  }
10     // It sinks v[s] in a heap ending in v[l-1]
11 method sink(s:nat, l:nat)
12 requires 0 ≤ s ≤ l = next ≤ v.Length
13 requires ∀ i | 0 < i < next • (i-1)/2 ≠ s ⟹
14                                v[(i-1)/2] ≤ v[i]
15 modifies v
16 ensures isHeap()
17 {
18    var j := s;
19    while 2*j+1 < l
20    invariant ∀ k | 0 < k < l • (k-1)/2 ≠ j ⟹
21                                  v[(k-1)/2] ≤ v[k]
22    {
23      var m: nat;
24      if 2*j+2 < l ∧ v[2*j+2] ≤ v[2*j+1] {
25          m := 2*j+2;     // Right son is the minimum
26      } else {
27          m := 2*j+1;     // Left son is the minimum
28      }
29      if v[j] > v[m] {
30          v[j], v[m] := v[m], v[j];
31          j  := m;
32      } else {
33          break;
34      }
35    }
36 }
```

Figura 3.4: Implementación de *deleteMin* y *sink* de un montículo de Williams.

```
1  method heapsort_with_extra_space (a: array<int>)
2  modifies a
3  {
4      var queue := new Williams_heap(a.Length);
5      var i := 0;
6      while i < a.Length
7          invariant queue.isHeap()
8      {
9          queue.insert(a[i]);
10         i := i + 1;
11     }
12     i := 0;
13     while i < a.Length
14         invariant queue.isHeap()
15     {
16         a[i] := queue.min(); queue.deleteMin();
17         i := i + 1;
18     }
19 }
```

Figura 3.5: Versión con espacio extra del algoritmo *heapsort*.

coste del programa de la figura 3.5 es:

$$max\,(\sum_{m=0}^{n-1}\log m, \sum_{m=1}^{n}\log m) \in \Theta(n\log n).$$

Es posible mejorar el coste en tiempo y en espacio del algoritmo *heapsort* si se trabaja directamente con el vector que implementa el montículo. De hecho, el propio vector a ordenar sirve para este propósito, con lo que el consumo en espacio del algoritmo pasa a estar en $\Theta(1)$. Ello marca una diferencia con respecto a otros algoritmos de ordenación interna con una eficiencia en tiempo en $\Theta(n\log n)$:

- El algoritmo *mergesort*, en su versión para ordenar vectores, necesita un espacio adicional en $\Theta(n)$ para el vector donde se almacena la *fusión* de las dos porciones de vector ordenadas previamente. En su versión para listas enlazadas, puede conseguirse un consumo constante del espacio de memoria dinámica, pero no así del espacio de pila, que necesita un coste en $\Theta(\log n)$.
- El algoritmo *quicksort* necesita un espacio, en promedio en $\Theta(\log n)$ y en el caso peor en $\Theta(n)$, para la pila de activación de las llamadas recursivas.

En la nueva versión, un primer bucle se dedica a convertir el vector de entrada en un montículo de máximos. Para ello se emplea la siguiente táctica: si n es el número

de elementos del vector, es fácil demostrar (véase el ejercicio 3.2) que los elementos

$$v[n/2], \ldots, v[n-1],$$

siendo '/' la división entera, corresponden exactamente a las posiciones de las hojas del futuro montículo. De hecho, son ya montículos de un elemento. Entonces, se procede a *hundir* los elementos $v[n/2-1]$ a $v[0]$, en ese orden, construyendo en sentido ascendente montículos de 3, 7, 15, etc. elementos, hasta llegar a formar un solo montículo. Se puede demostrar que el coste de este bucle está en $\Theta(n)$ mediante el siguiente razonamiento: considerando el peor caso, que sería un árbol completo de altura $k = \lfloor \log n \rfloor$, hundir los nodos interiores de profundidad $k-1$, ocasiona $2\frac{n}{4}$ comparaciones, hundir los de profundidad $k-2$, ocasiona $4\frac{n}{8}$ comparaciones, los de profundidad $k-2$, $6\frac{n}{16}$, etc. El número total de comparaciones es, pues:

$$\sum_{i=1}^{\lfloor \log n \rfloor - 1} n \frac{2i}{2^{i+1}} = n \sum_{i=1}^{\lfloor \log n \rfloor - 1} \frac{i}{2^i},$$

donde la cantidad $\sum_{i=1}^{\lfloor \log n \rfloor - 1} \frac{i}{2^i} \leq \sum_{i=1}^{\infty} \frac{i}{2^i}$ está acotada por una constante.

El segundo bucle es muy similar al de la primera versión, con la diferencia de que ahora se extrae sucesivamente el *máximo* del montículo y se coloca al final del vector. En todo momento de la iteración el vector está dividido en dos porciones: la $v[0..i-1]$, que corresponde al montículo remanente y la $v[i,..n-1]$, que contiene los elementos extraídos del montículo y que está ordenada crecientemente. El coste de este segundo bucle está en $\Theta(n \log n)$. El algoritmo completo se presenta en la figura 3.6.

El predicado `MAXHEAP` mostrado en la figura es similar al predicado `isHeap` de la figura 3.2, pero en este caso define un montículo de máximos. Precisando, `MAXHEAP(v,i,j)` dice que el segmento de vector $v[i..j-1]$ cumple la propiedad de montículo de máximos. Igualmente, `sorted_seg(v,i,j)` expresa que el segmento $v[i..j-1]$ está ordenado crecientemente. El método `sink` es idéntico al método `sink` de la clase `Williams_heap` de la figura 3.4, salvo por el hecho de que ahora el vector v es un argumento explícito.

El invariante del primer bucle **while** expresa el hecho de que la parte del vector a la derecha de i es un montículo de máximos. Al terminar con $i = -1$, la propiedad se convierte en que todo el vector es un montículo. El invariante del segundo bucle formaliza la propiedad expresada más arriba de que $v[0..i-1]$ es un montículo y que $v[i..n-1]$ está ordenado crecientemente. Otra propiedad invariante adicional, necesaria para completar la verificación, es que todos los elementos del montículo remanente son menores o iguales que cualquiera de los elementos de la parte ordenada. Cuando el bucle termina con $i = 1$, esta última propiedad garantiza que el elemento $v[0]$ es más pequeño que los del segmento $v[1..n-1]$ y, por tanto, todo el vector está ordenado. Nótese también que la propiedad

```
multiset (v[..]) == old (multiset (v[..]))
```

```
1  method heapsort  (v: array<int>)
2  modifies v
3  ensures sorted_seg(v, 0, v.Length)
4  ensures multiset(v[..]) = old(multiset(v[..]))
5  {
6    var i := (v.Length / 2) - 1;
7    while i ≥ 0
8    invariant -1 ≤ i ≤ (v.Length / 2) - 1
9    invariant MAXHEAP (v, i+1, v.Length)
10   invariant  multiset(v[..]) = old(multiset(v[..]))
11   {
12     sink(v, i, v.Length);
13     i := i - 1;
14   }
15   i := v.Length;
16   while i > 1
17   invariant 0 ≤ i ≤ v.Length
18   invariant sorted_seg(v, i, v.Length)
19   invariant MAXHEAP(v, 0, i)
20   invariant ∀ k, m • 0 ≤ k < i ≤ m < v. Length ⟹
21                       v[k] ≤ v[m]
22   invariant multiset(v[..]) = old(multiset(v[..]))
23   {
24      i := i - 1;
25      v[0], v[i] := v[i], v[0];
26      sink(v, 0, i);
27   }
28 }
29 predicate MAXHEAP (v: array<int>, i: nat, j:nat)
30 reads v
31 requires 0 ≤ i ≤ j ≤ v.Length
32 {
33    ∀ k • 2*i+1 ≤ k < j ⟹ v[(k-1)/2] ≥ v[k]
34 }
```

Figura 3.6: Versión definitiva del algoritmo *heapsort*.

es invariante de ambos bucles, lo que garantiza que el vector resultante es una permutación del vector de entrada. Con dichos invariantes y cierta ayuda adicional en forma de algunos asertos intermedios y lemas, Dafny verifica la corrección de este algoritmo.

El coste del algoritmo *heapsort* en el caso peor está en $\Theta(n \log n)$. Su coste promedio está también en dicha clase de complejidad. Además, como se ha dicho, no necesita espacio adicional. Sin embargo, su constante multiplicativa es ligeramente superior a la del algoritmo *quicksort*. Dado que su primera fase tiene un coste lineal, es, no obstante, un algoritmo muy recomendable cuando solo se pretenden obtener los k menores (o mayores) elementos de un vector, siendo k bastante menor que n.