

7

COLAS Y MONTONES DE PRIORIDAD

Las colas de prioridad son estructuras de datos que recuperan elementos ordenados según la puntuación dada para cada uno. Mientras tanto las pilas como las colas del Capítulo 4 dependían únicamente del orden de inserción de los datos, las colas de prioridad utilizan información adicional para determinar el orden de recuperación: la prioridad del elemento. Como veremos, esta nueva información nos permite adaptarnos mejor a los datos y, entre otras aplicaciones útiles, procesar primero las solicitudes urgentes.



Por ejemplo, imagina que una nueva cafetería, Dynamic Selection Coffee, ha abierto en tu barrio. Lleno de entusiasmo, entras y ves 10 tipos de granos de café que nunca has probado. Antes de lanzarte a probarlos, dedicas la siguiente hora a analizar cuidadosamente las ventajas de cada nueva marca, basándote en su menú, lamentablemente deficiente. Descripciones, lo que te deja con una lista ordenada de cafés para probar. Seleccionas la marca más prometedora de la lista, la compras y te vas a casa a disfrutar de la experiencia.

Al día siguiente, regresas a Dynamic Selection Coffee para probar el segundo plato de tu lista, solo para descubrir que han añadido dos cafés más al menú. Cuando le preguntas al barista por qué han hecho este cambio, te señala el cartel de la tienda, que explica que Dynamic Selection Coffee ofrece una selección de cafés en constante expansión. Su objetivo es llegar a servir más de mil variedades. Sientes una emoción y un miedo a la vez. Cada día tendrás que priorizar los nuevos cafés e incluirlos en tu lista para saber cuál probar a continuación.

La tarea de recuperar elementos de una lista priorizada es una tarea recurrente en los programas informáticos: dada una lista de elementos y sus prioridades asociadas, ¿cómo podemos recuperar eficientemente el siguiente elemento en orden de prioridad? A menudo, necesitamos realizar esta recuperación en un contexto dinámico donde llegan nuevos elementos constantemente. Podríamos tener que elegir qué paquete de red procesar según la prioridad, ofrecer la mejor sugerencia en el corrector ortográfico basándose en errores ortográficos comunes o elegir la siguiente opción en una búsqueda prioritaria. En el mundo real, podríamos usar nuestras propias colas de prioridades mentales para decidir qué tarea urgente realizar a continuación, qué película ver o a qué paciente atender primero en una sala de urgencias abarrotada. Una vez que se empieza a buscar, las recuperaciones priorizadas están por todas partes.

En este capítulo, presentamos la cola de prioridad, una clase de estructura de datos para recuperar elementos priorizados de un conjunto, y luego analizamos la estructura de datos más común para implementar esta útil herramienta: el montón. Los montones hacen que las operaciones centrales de una cola de prioridad sean extremadamente eficientes.

Colas de prioridad

Las colas de prioridad almacenan un conjunto de elementos y permiten al usuario recuperar fácilmente el elemento con mayor prioridad. Son dinámicas, lo que permite combinar inserciones y recuperaciones. Necesitamos poder añadir y eliminar elementos de nuestra lista de tareas priorizadas. Si estuviéramos limitados a usar una estructura de datos fija, el autor podría pasar el día consultando su lista estática y realizando repetidamente la tarea de mayor prioridad: "Preparar el café de la mañana". Sin la posibilidad de eliminar esa tarea una vez completada, permanecería en la parte superior de la lista del autor. Si bien esto podría resultar en un día agradable, es poco probable que sea productivo.

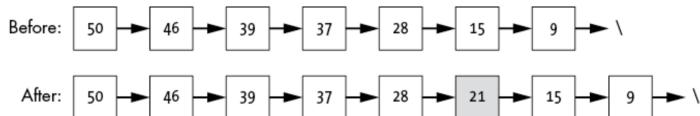
En su forma más básica, las colas de prioridad admiten algunas operaciones principales:

- Agregue un elemento y su puntuación de prioridad asociada.
- Busque el elemento con mayor prioridad (o nulo si la cola está vacía).
- Eliminar el elemento con mayor prioridad (o nulo si la cola está vacía).

También podemos agregar otras funciones útiles que nos permitan comprobar si una cola de prioridad está vacía o devolver el número de elementos almacenados actualmente.

Establecemos las prioridades de los elementos según el problema en cuestión. En algunos casos, los valores de prioridad pueden ser obvios o estar determinados por el algoritmo. Al procesar solicitudes de red, por ejemplo, cada paquete puede tener una prioridad explícita, o podemos optar por procesar primero la solicitud más antigua. Sin embargo, decidir qué priorizar no siempre es sencillo. Al priorizar la marca de café que probaremos a continuación, podríamos priorizarla según el precio, la disponibilidad o el contenido de cafeína; esto depende de cómo planeemos usar nuestra cola de prioridades.

Es posible, aunque no ideal, implementar colas de prioridad con estructuras de datos primitivas, como listas enlazadas ordenadas o matrices ordenadas, añadiendo nuevos elementos a la lista según su prioridad. [La Figura 7-1](#) muestra un ejemplo de cómo añadir el valor 21 a una lista enlazada ordenada.

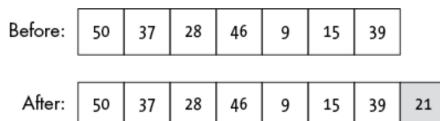


[Figura 7-1](#) : Agregar un elemento (21) a una lista enlazada ordenada que representa una cola de prioridad

Una lista enlazada ordenada mantiene el elemento de mayor prioridad al principio para facilitar la búsqueda. De hecho, en este caso, la búsqueda toma un tiempo constante, independientemente de la longitud de la cola de prioridad; solo se examina el primer elemento. Desafortunadamente, añadir nuevos elementos puede ser costoso. Podríamos tener que recorrer toda la lista cada vez que añadimos un nuevo elemento, lo que nos cuesta un tiempo proporcional a la longitud de la cola de prioridad.

El autor utiliza una versión práctica de este método de lista ordenada para organizar su refrigerador, ordenando los artículos de adelante hacia atrás según su fecha de caducidad. El artículo más cercano siempre tiene la máxima prioridad, el que caducará antes. Encontrar el artículo correcto es fácil: basta con coger lo que esté delante. Este sistema es especialmente beneficioso al guardar leche o crema para el café de la mañana: nadie quiere pasarse los primeros minutos de la mañana leyendo fechas de caducidad con la vista nublada. Sin embargo, colocar artículos nuevos detrás de los viejos puede llevar tiempo y requerir una cantidad modesta de cambios.

De forma similar, podríamos mantener la cola de prioridad en una lista enlazada o matriz *sin ordenar*. Añadir elementos nuevos es trivial: basta con etiquetar el elemento al final de la lista, como se ilustra en [la Figura 7-2](#).



[Figura 7-2](#) : Agregar un elemento (21) a una matriz sin ordenar que representa una cola de prioridad

Desafortunadamente, ahora pagamos un alto costo al buscar el siguiente elemento. Debemos escanear toda la lista para determinar cuál tiene la mayor prioridad. Si lo eliminamos, también lo desplazamos todo para llenar el vacío. Este enfoque equivale a escanear todos los artículos del refrigerador del autor para encontrar el que esté más cerca de caducar. Esto podría funcionar para un refrigerador con solo unos pocos cartones de leche, pero imagine la sobrecarga de revisar cada ar-

título de comida o bebida en un supermercado grande.

Implementar colas de prioridad como listas ordenadas puede funcionar mejor que usar listas desordenadas, o viceversa, dependiendo de cómo planeemos usar la cola de prioridad. Si las adiciones son más comunes que las búsquedas, preferimos la lista desordenada. Si las búsquedas son más comunes, deberíamos pagar el costo de mantener los elementos ordenados. En el caso de los refrigeradores, las búsquedas son mucho más comunes: usamos un solo cartón de leche con más frecuencia que uno nuevo, por lo que conviene mantener la leche ordenada. El desafío surge cuando tanto las adiciones como las búsquedas son comunes; priorizar una operación sobre la otra generará inefficiencia general, por lo que necesitamos un método que equilibre sus costos. Una estructura de datos inteligente, el montón, nos ayuda a resolver este problema.

Max Heaps

Un *montículo máximo* es una variante del árbol binario que mantiene una relación ordenada especial entre un nodo y sus hijos. Específicamente, un montículo máximo almacena los elementos según la *propiedad de montículo máximo*, que establece que el valor en cualquier nodo del árbol es mayor o igual que los valores de sus nodos hijos. Para simplificar, a lo largo del capítulo, usaremos los términos más generales "*montículos*" y "*propiedad del montículo*" para referirnos a los *montículos máximos* y a la *propiedad de montículo máximo*.

La figura 7-3 muestra una representación de un árbol binario organizado de acuerdo con la propiedad de montón máximo.

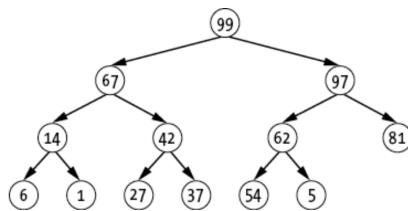


Figura 7-3 : Un montón representado como un árbol binario

No hay preferencia ni orden especial entre los hijos izquierdo y derecho, salvo que tienen menor prioridad que su padre o madre. A modo de comparación, imaginemos un programa de mentoría para amantes del café de élite, la Sociedad para la Mejora del Conocimiento Relacionado con el Café. Cada miembro (nodo) se compromete a mentorizar hasta a otros dos amantes del café (nodos hijos). La única condición es que ninguno de los aprendices sepa más sobre café que su mentor; de lo contrario, la mentoría sería un desperdicio.

El informático JWJ Williams inventó los montículos como componente de un nuevo algoritmo de ordenación, heapsort, que analizaremos más adelante en este capítulo. Sin embargo, reconoció que los montículos son estructuras de datos útiles para otras tareas. La estructura simple del montículo máximo le permite gestionar eficientemente las operaciones necesarias para las colas de prioridad: (1) permitir al usuario buscar eficientemente el elemento más grande, (2) eliminar el elemento más grande y (3) añadir un elemento arbitrario.

Los montículos suelen visualizarse como árboles, pero suelen implementarse con matrices para mayor eficiencia. En este capítulo, presentamos estas dos representaciones en paralelo para que el lector pueda establecer conexiones mentales entre ellas. Sin embargo, no es necesario usar matrices para la implementación.

En la implementación basada en matrices, cada elemento de la matriz corresponde a un nodo del árbol, cuyo nodo raíz se encuentra en el índice 1 (omitimos el índice 0 para mantener la coherencia con la convención común para montículos). Los índices de los nodos secundarios se definen en relación con los índices de sus padres, por lo que un nodo en el índice i tiene hijos en los índices $2i$ y $2i + 1$. Por ejemplo, el nodo en el índice 2 tiene un hijo en el índice $2 \times 2 = 4$ e índice $2 \times 2 + 1 = 5$, como se muestra en *la Figura 7-4*.

66	65	60	10	17	22
----	----	----	----	----	----

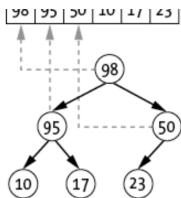


Figura 7-4 : La posición del montón corresponde a las ubicaciones del índice.

De forma similar, calculamos el índice del padre de un nodo como `Floor(i/2)`. El esquema de indexación permite al algoritmo calcular fácilmente el índice de un hijo a partir del del padre, y el índice de un padre a partir de su hijo.

El nodo raíz siempre corresponde al valor máximo en un montículo máximo. Dado que almacenamos el nodo raíz en una posición fija del array (índice = 1), siempre podemos encontrar este valor máximo en tiempo constante. Es simplemente una búsqueda en el array. Por lo tanto, la disposición de los datos aborda una de las operaciones necesarias de la cola de prioridad.

Dado que añadiremos y eliminaremos elementos arbitrarios en nuestra cola de prioridad, para evitar redimensionar constantemente el array, queremos preasignar un array lo suficientemente grande como para albergar la cantidad de elementos que esperamos añadir. Recuerde, como se explicó en el Capítulo 3, que redimensionar dinámicamente un array puede ser costoso, ya que nos obliga a crear un nuevo array y copiar los valores, desperdiando así la valiosa eficiencia del montón. En su lugar, podemos asignar inicialmente un array grande, rastrear el índice del último elemento lleno del array y llamar a ese índice el final virtual del array. Esto nos permite añadir nuevos elementos simplemente actualizando el índice del último elemento.

```
Heap {
    Array: array
    Integer: array_size
    Integer: last_index
}
```

Por supuesto, el costo de esta sobreasignación es una porción de memoria potencialmente no utilizada si nuestro montón no crece tanto como se espera.

Usar un array para representar una estructura de datos basada en árboles es un paso interesante. Podemos usar arrays empaquetados y una asignación matemática para representar un montón en lugar de depender de punteros, lo que nos permite almacenar el montón con menos memoria. Al mantener una asignación deseada del índice de un nodo hasta el de sus hijos, podemos recrear una estructura de datos basada en árboles sin los punteros. Como veremos más adelante, esta representación basada en arrays es viable para montones porque la estructura de datos siempre mantiene un árbol casi completo y equilibrado. Esto da como resultado un array empaquetado sin huecos. Si bien podríamos usar la misma representación de array para otros árboles, como los árboles de búsqueda binaria, estas estructuras de datos suelen presentar huecos y requerirían arrays muy grandes (y posiblemente casi vacíos) para almacenar árboles con ramas profundas.

Agregar elementos a un montón

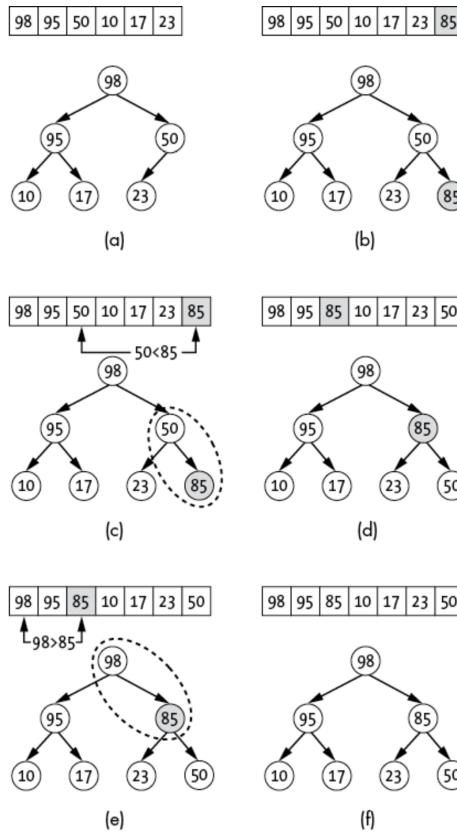
Al añadir un nuevo elemento a un montón, debemos asegurarnos de que la estructura conserve la propiedad del montón. Así como no se asignaría a un general condecorado que se reportara a un teniente recién ascendido, tampoco se colocaría un nodo del montón con alta prioridad bajo uno con baja prioridad. Debemos añadir el elemento a la estructura de árbol del montón de forma que todos los elementos por debajo de la adición tengan una prioridad menor o igual a la del nuevo nodo. De igual forma, todos los nodos por encima de la nueva adición deben tener prioridades mayores o iguales a la del nuevo nodo.

Parte de la brillantez de la implementación de un montón como matriz reside en que conserva esta propiedad al almacenar los nodos como una matriz compacta. En capítulos anteriores, añadir nodos al centro de una matriz era costoso, lo que

obligaba a desplazar las entradas posteriores hacia abajo. Afortunadamente, no necesitamos pagar este coste lineal cada vez que añadimos un nuevo elemento a un montón. En su lugar, añadimos elementos primero descomponiendo la propiedad del montón y luego intercambiando elementos a lo largo de una sola rama del árbol para restaurarla.

En otras palabras, para añadir un nuevo elemento al montón, lo añadimos al primer espacio vacío del nivel inferior del árbol. Si este nuevo valor es mayor que el de su nodo padre, lo expandimos hasta que sea menor o igual que su padre, restaurando así la propiedad del montón. La propia estructura del montón nos permite hacerlo de forma eficiente. En la implementación de un montón como array, esto equivale a añadir el nuevo elemento al final del array e intercambiarlo hacia adelante.

Consideré el ejemplo de [la Figura 7-5](#), que muestra la estructura del montón como un array y un árbol en cada paso. [La Figura 7-5](#) (a) muestra el montón antes de añadir el nuevo elemento. En [la Figura 7-5](#) (b), añadimos el nuevo elemento. El elemento 85 se coloca al final del array, insertándolo en la parte inferior del árbol. Tras la primera comparación ([Figura 7-5](#) (c)), intercambiamos el nuevo elemento con su nodo padre, ya que 85 es mayor que 50. El intercambio se muestra en la [Figura 7-5](#) (d). La segunda comparación ([Figura 7-5](#) (e)) revela que el nuevo nodo se encuentra ahora en la posición correcta en la jerarquía: 98 es mayor que 85, por lo que no es necesario que el nuevo nodo se intercambie con su padre una segunda vez. [La Figura 7-5](#) (f) muestra el montículo una vez completada la adición.



[Figura 7-5](#) : Agregar un elemento (85) a un montón

El código para implementar esta adición utiliza un solo `WHILE` bucle para avanzar por los niveles del montón hasta que llega a la raíz o encuentra un parent mayor o igual al valor del nuevo nodo:

```
HeapInsert(Heap: heap, Type: value):
    ① IF heap.last_index == heap.array_size - 1:
        Increase Heap size.

    ② heap.last_index = heap.last_index + 1
        heap.array[heap.last_index] = value

        # Swap the new node up the heap.
```

```

③ Integer: current = heap.last_index
           Integer: parent = Floor(current / 2)
④ WHILE parent >= 1 AND (heap.array[parent] <
                           heap.array[current]): 
    ⑤ Type: temp = heap.array[parent]
            heap.array[parent] = heap.array[current]
            heap.array[current] = temp
            current = parent
            parent = Floor(current / 2)

```

Dado que usamos un array para almacenar el montón, el código comienza comprobando que aún haya espacio en el array para añadir un nuevo elemento ❶. De no ser así, aumenta el tamaño del montón, quizás aplicando la técnica de duplicación de arrays descrita en el Capítulo 3. A continuación, el código añade el nuevo elemento al final del array y actualiza la posición del último elemento ❷. El WHILE bucle comienza en el elemento recién añadido ❸, asciende por cada capa del montón comparando el valor actual con el de su elemento primario ❹, y cambiando si es necesario ❺. El bucle termina cuando llegamos al principio del montón (`parent == 0`) o encontramos un elemento primario mayor o igual que el secundario.

Podríamos comparar este proceso con un centro de distribución de paquetes con un diseño extraño, pero eficiente, como se muestra en [la Figura 7-6](#). Los empleados organizan los paquetes en filas ordenadas en el suelo utilizando la propiedad heap: en la primera fila hay un único paquete con la máxima prioridad, el siguiente en ser enviado. Detrás de ese paquete se sitúan dos paquetes de menor prioridad. Detrás de cada uno de estos hay dos paquetes más (para un total de cuatro en esa fila) de modo que cada par tiene prioridades menores o iguales que el paquete correspondiente que tienen delante. Con cada nueva fila, el número de paquetes se duplica, de modo que la disposición se extiende cada vez más a medida que se avanza hacia la parte trasera del almacén. Cada paquete tiene como máximo dos paquetes de menor o igual prioridad detrás de él y como máximo un paquete de mayor o igual prioridad al frente. Los rectángulos pintados en el suelo del almacén indican de forma útil las posibles ubicaciones de los paquetes en cada una de las filas.

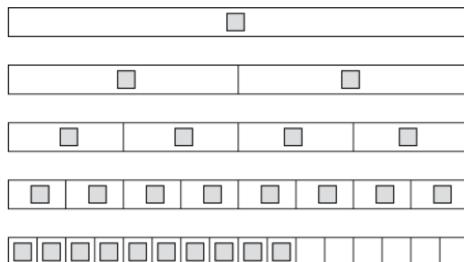


Figura 7-6: Un piso de almacén organizado como un montón

Se traen nuevos paquetes desde la parte trasera del almacén. Cada repartidor murmura algo sobre el extraño sistema de clasificación, aliviado de no tener que llevar su paquete hasta el frente, lo deja en el primer lugar disponible y se va lo más rápido posible. Los empleados del almacén entran en acción, comparando la prioridad del nuevo paquete con el del inmediatamente anterior (ignorando el resto de los paquetes en esa fila). Si encuentran una inversión, intercambian los paquetes. De lo contrario, dejan el paquete donde está. Esto continúa hasta que el nuevo paquete ocupa el lugar apropiado en la jerarquía, con el paquete anterior con mayor o igual prioridad. Como los paquetes son pesados y están dispersos, los empleados minimizan su trabajo con un máximo de una comparación y un cambio por fila. Nunca reorganizan los paquetes dentro de una fila. Después de todo, nadie quiere mover cajas innecesariamente.

Intuitivamente, podemos ver que añadir montículos no es excesivamente costoso. En el peor de los casos, tendríamos que intercambiar el nuevo nodo hasta la raíz del árbol, pero esto solo implica intercambiar una pequeña fracción de los valores del array. Por diseño, los montículos son árboles binarios *balanceados*: completamos un nivel del árbol antes de insertar un nodo en el siguiente. Dado que el número de nodos en un árbol binario completo se duplica con cada nivel, la opera-

ción de adición requiere, en el peor de los casos, logaritmo de $2 \times N$ intercambios. Esto es significativamente mejor que, en el peor de los casos, N intercambios necesarios para mantener una lista ordenada.

Eliminación de los elementos de mayor prioridad de los montones

Buscar y eliminar el elemento de mayor prioridad de una cola de prioridad es una operación fundamental que nos permite procesar elementos según su prioridad. Quizás estemos almacenando una lista de solicitudes de red pendientes y queramos procesar la de mayor prioridad. O podríamos estar gestionando una sala de urgencias y buscando atender al paciente más urgente. En ambos casos, queremos eliminar este elemento de nuestra cola de prioridad para poder extraer el siguiente elemento de mayor prioridad.

Para eliminar el nodo de mayor prioridad, primero debemos romper y luego restaurar la propiedad del montón. Consideré el ejemplo de [la Figura 7-7](#). Comenzamos intercambiando el nodo de mayor prioridad con el último nodo en el nivel inferior del árbol ([Figura 7-7 \(b\)](#)), convirtiendo efectivamente el último elemento en el nuevo nodo raíz. Sin embargo, es casi seguro que esta promoción épica del nuevo nodo raíz no resistirá un análisis riguroso. En la implementación del array, esto equivale a intercambiar el primer y el último elemento del array. Este intercambio cierra el hueco al principio del array que se crearía al eliminar el primer elemento y, por lo tanto, mantiene un array compacto.

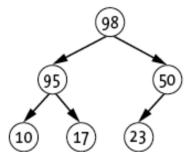
A continuación, se elimina el valor máximo original de 98, que actualmente es el último elemento del árbol, como se muestra en [la Figura 7-7 \(c\)](#). Hemos eliminado el nodo correcto, pero probablemente hemos dañado la propiedad del montón en el proceso. Es posible que hayamos movido un paquete de baja urgencia al principio del almacén.

NOTA

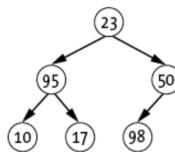
También es posible que intercambiar el primer y el último elemento del montón no rompa la propiedad del montón, especialmente en el caso de prioridades duplicadas o montones pequeños.

98 95 50 10 17 23

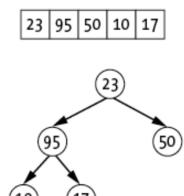
23 95 50 10 17 98



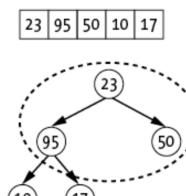
(a)



(b)



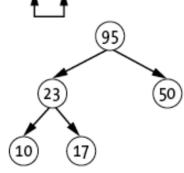
(c)



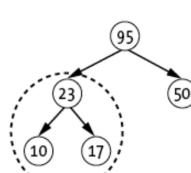
(d)

95 23 50 10 17

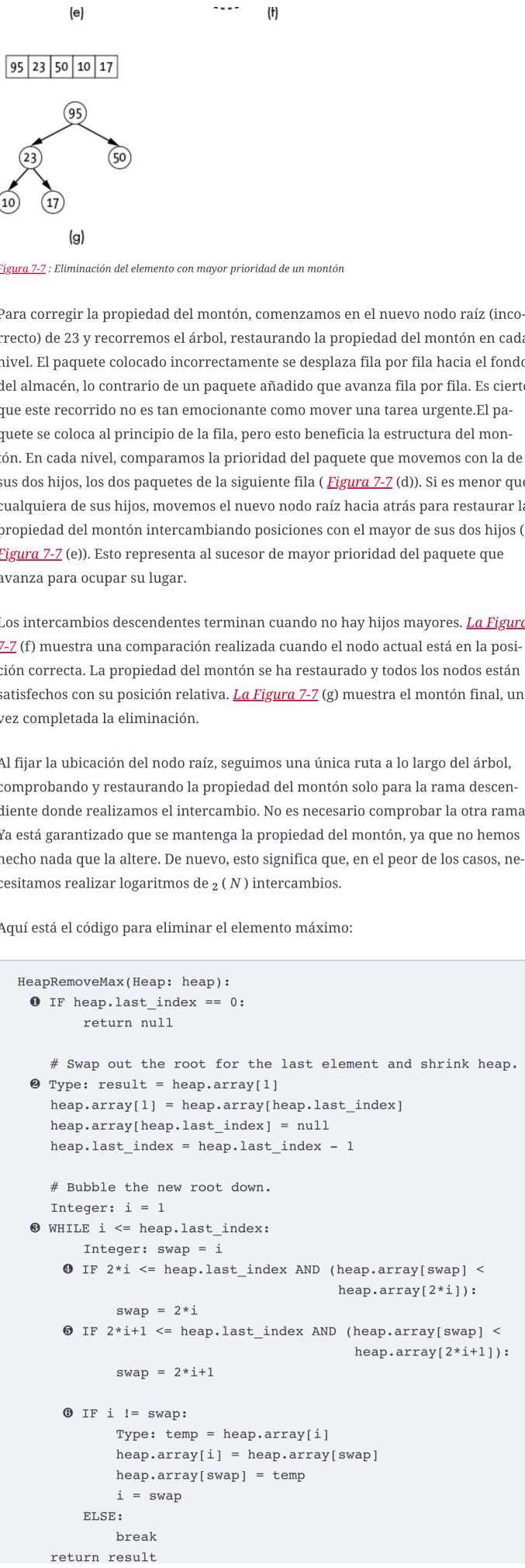
95 23 50 10 17



(e)



(f)



Este código comienza comprobando que el montón no esté vacío ❶. Si lo está, no hay nada que devolver. El código luego intercambia el primer elemento (`index == 1`) con el último elemento (`index == heap.last_index`), rompiendo la propiedad del montón para preparar el elemento máximo para su eliminación ❷. El código luego usa un `WHILE` bucle para recorrer el montón en una serie de comparaciones, reparando la propiedad del montón ❸. Durante cada iteración, compara el valor actual con ambos hijos e intercambia con el mayor si es necesario ❹. Debemos agregar verificaciones adicionales ❺ ❻ para asegurar que el código solo esté comparando el valor actual con un hijo existente. No queremos que intente comparar con una entrada que esté más allá del último índice válido del array. El bucle termina cuando ha llegado al final del montón o ha pasado una iteración sin un intercambio (a través de la `break` declaración).

Almacenamiento de información auxiliar

Muchas veces, necesitamos que nuestro montón almacene información adicional para cada entrada. En nuestra lista de tareas, por ejemplo, necesitamos almacenar información sobre las tareas pendientes, no solo su prioridad. De nada sirve saber que debemos realizar la tarea con prioridad 99 si no sabemos cuál es. Podríamos revisar la lista original manualmente.

Aumentar el montón para almacenar estructuras de datos u objetos compuestos, como un `TaskRecord`, es simple:

```
TaskRecord {  
    Float: Priority  
    String: TaskName  
    String: Instructions  
    String: PersonWhoWillYellIfThisIsNotDone  
    Boolean: Completed  
}
```

Modificamos el código anterior para gestionar las comparaciones según el campo de prioridad de este registro compuesto. Podríamos hacerlo modificando directamente el código (como en la `HeapInsert` función):

```
WHILE parent >= 1 AND (heap.array[parent].priority <  
                        heap.array[current].priority):
```

Sin embargo, esto requiere que especialicemos la implementación del montón a la estructura de datos compuesta específica. Un enfoque más limpio consiste en añadir una función auxiliar específica para la estructura de datos compuesta, como:

```
IsLessThan(Type: a, Type: b):  
    return a.priority < b.priority
```

Usaríamos esta función en lugar del signo matemático menor que en el código para `HeapInsert`:

```
WHILE parent >= 1 AND IsLessThan(heap.array[parent],  
                                   heap.array[current]):
```

De manera similar, modificaríamos las comparaciones en la `HeapRemoveMax` función para utilizar la función auxiliar.

```
IF 2*i <= heap.last_index AND IsLessThan(heap.array[swap],  
                                         heap.array[2*i])  
    swap = 2*i  
IF 2*i+1 <= heap.last_index AND IsLessThan(heap.array[swap],  
                                         heap.array[2*i+1])  
    swap = 2*i+1
```

Estos pequeños cambios nos permiten crear montones a partir de estructuras de

uatos compuestas. Si podemos definir una función para ordenar los elementos, podemos crear una cola de prioridad eficiente para ellos.

Actualización de prioridades

Algunos casos de uso podrían requerir otro modo de comportamiento dinámico: permitir que el algoritmo actualice las prioridades de los elementos dentro de la cola de prioridad. Considere la base de datos de una librería que prioriza los libros a reponer según el número de usuarios que han solicitado cada título. El sistema construye el montón sobre una lista inicial de libros y la utiliza para determinar qué título pedir a continuación. Sin embargo, después de que un popular artículo de blog señala la importancia vital de las estructuras de datos en el pensamiento computacional, la librería experimenta un aumento drástico, aunque totalmente comprensible, en el número de usuarios que solicitan libros sobre estructuras de datos. Su cola de prioridad debe estar preparada para gestionar esta afluencia repentina.

Para satisfacer esta necesidad, utilizamos los mismos enfoques que aplicamos para la adición y la eliminación. Al cambiar el valor de un elemento, verificamos si estamos aumentando o disminuyendo la prioridad. Si aumentamos su valor, debemos subirlo al montón máximo para restaurar su propiedad. De igual forma, si disminuimos su valor, lo dejamos descender al montón máximo hasta su posición correcta.

```
UpdateValue(Heap: heap, Integer: index, Float: value):
    Type: old_value = heap.array[index]
    heap.array[index] = value

    IF old_value < value:
        Bubble the element up the heap using the
        procedure from inserting new elements
        (swapping with parent).
    ELSE:
        Drop the element down the heap using the
        procedure from removing the max element
        (swapping with the larger child).
```

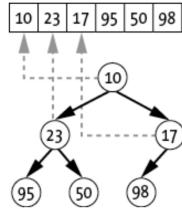
Incluso podemos desglosar el código para permitir que los elementos suban o bajen, de modo que se pueda usar exactamente el mismo código tanto para actualizar como para agregar y eliminar el máximo.

¿Cómo encontramos el elemento que queremos actualizar en primer lugar? Como se mencionó, los montículos no están optimizados para encontrar elementos específicos. Si no tenemos información sobre el elemento que nos interesa, aparte de su valor, podríamos... Es necesario buscar en una parte sustancial del array para encontrarlo. A menudo, podemos resolver este problema usando una estructura de datos secundaria, como una tabla hash (discutida en el Capítulo 10), para mapear la clave del elemento a su elemento en el montón. En el ejemplo de esta sección, asumimos que el programa ya tiene el índice actual del elemento.

Montones mínimos

Hasta ahora, nos hemos centrado en el montículo máximo, que utiliza la propiedad de que el valor en cualquier nodo del árbol es mayor (o igual) que los valores de sus hijos. El *montículo mínimo* es una versión del montículo que facilita la búsqueda del elemento con el valor más bajo. Con un montículo mínimo, la raíz del árbol es el valor más pequeño, lo que nos permite encontrar fácilmente el elemento con la puntuación más baja. Por ejemplo, en lugar de ordenar los paquetes de red por prioridad, podríamos ordenarlos por hora de llegada, procesando los paquetes con una hora de llegada anterior antes que los paquetes recibidos más recientemente. Más importante aún, si nos quedamos sin espacio en nuestra estantería de café, tendremos que eliminar nuestra marca menos favorita. Tras un desgarrador debate interno sobre si sería mejor aumentar el tamaño de nuestro almacenamiento de café desechar los platos o tazones de un estante en lugar de deshacernos de nuestros preciados posos de café, decidimos descartar el café con la puntuación más baja. Consultamos nuestra lista de puntuaciones de disfrute para cada café y seleccionamos el que tenga el valor más bajo.

En teoría, podríamos seguir usando un montículo máximo simplemente negando los valores. Sin embargo, una estrategia más sencilla consiste en modificar ligeramente nuestra propiedad del montículo y resolver el problema por completo. La *propiedad del montículo mínimo* permite que el valor en cualquier nodo del árbol sea menor (o igual) que los valores de sus hijos. La [Figura 7-8](#) muestra un ejemplo de montículo mínimo. Al insertar nuevos elementos, los que tienen la puntuación más baja ascienden en la jerarquía. De igual forma, siempre extraemos y devolvemos el elemento con la puntuación más baja.



[Figura 7-8](#): La posición del montón mínimo corresponde a las ubicaciones de los índices.

Por supuesto, los algoritmos para añadir y eliminar elementos en el montículo mínimo deben modificarse según corresponda. Para la inserción, modificamos la función de comparación en el `WHILE` bucle para comprobar si el valor principal es mayor que el valor actual:

```
MinHeapInsert(MinHeap: heap, Type: value):
    IF heap.last_index == heap.array_size - 1:
        Increase Heap size.

    heap.last_index = heap.last_index + 1
    heap.array[heap.last_index] = value

    # Swap the new node up the heap.
    Integer: current = heap.last_index
    Integer: parent = Floor(current / 2)
    ❶ WHILE parent >= 1 AND (heap.array[parent] >
        heap.array[current]):
        Type: temp = heap.array[parent]
        heap.array[parent] = heap.array[current]
        heap.array[current] = temp
        current = parent
        parent = Floor(current / 2)
```

La mayor parte del código es idéntico al de la inserción en un montón máximo. El único cambio es reemplazar `<` con `>` al comparar un nodo con su parent **❶**.

Realizamos un cambio similar para las dos comparaciones durante la operación de eliminación:

```
MinHeapRemoveMin(Heap: heap):
    IF heap.last_index == 0:
        return null

    # Swap out the root for the last element and shrink heap.
    Type: result = heap.array[1]
    heap.array[1] = heap.array[heap.last_index]
    heap.array[heap.last_index] = null
    heap.last_index = heap.last_index - 1

    # Bubble the new root down.
    Integer: i = 1
    WHILE i <= heap.last_index:
        Integer: swap = i
        ❶ IF 2*i <= heap.last_index AND (heap.array[swap] >
            heap.array[2*i]):
            swap = 2*i
        ❷ IF 2*i+1 <= heap.last_index AND (heap.array[swap] >
            heap.array[2*i+1]):
            swap = 2*i+1
```

```

    IF i == swap:
        Type: temp = heap.array[i]
        heap.array[i] = heap.array[swap]
        heap.array[swap] = temp
        i = swap
    ELSE:
        break
    return result

```

Aquí, los únicos cambios son reemplazar `<` con `>` al determinar si se deben intercambiar nodos ❶ ❷.

Ordenamiento por montones

Los montículos son potentes estructuras de datos que se utilizan en diversas tareas informáticas, y no se limitan a la implementación de colas de prioridad ni a la devolución eficiente del siguiente elemento de una lista priorizada. Otra perspectiva interesante para analizar los montículos, y las estructuras de datos en general, es la de los novedosos algoritmos que permiten. JWJ Williams propuso inicialmente los propios montículos en el contexto de un nuevo algoritmo para ordenar arrays: *heapsort*.

Como su nombre indica, la ordenación por montículos (*heapsort*) es un algoritmo para ordenar una lista de elementos mediante la estructura de datos del montículo. La entrada es un array sin ordenar. La salida es un array que contiene los mismos elementos, pero en orden *descendente* (para un montículo máximo). En esencia, la ordenación por montículos consta de dos fases:

1. Construyendo un montón máximo a partir de todos los elementos
2. Extraer todos los elementos del montón en orden descendente y almacenarlos en una matriz

Es así de simple.

Aquí está el código de *heapsort*:

```

Heapsort(Array: unsorted):
    Integer: N = length(unsorted)
    Heap: tmp_heap = Heap of size N
    Array: result = array of size N

    Integer: j = 0
    ❶ WHILE j < N:
        HeapInsert(tmp_heap, unsorted[j])
        j = j + 1

    j = 0
    ❷ WHILE j < N:
        result[j] = HeapRemoveMax(tmp_heap)
        j = j + 1
    return result

```

Este código consta de dos `WHILE` bucles. El primero inserta cada elemento en un montón temporal ❶. El segundo bucle elimina el elemento más grande mediante la `HeapRemoveMax` función y lo añade a la siguiente posición del array ❷. Como alternativa, podemos implementar la ordenación por montículo para generar una respuesta en orden *creciente* mediante un montón mínimo y `HeapRemoveMin`.

Supongamos que queremos ordenar el siguiente array `[46, 35, 9, 28, 61, 8, 38, 40]` en orden decreciente. Empezamos insertando cada uno de estos valores en nuestro montículo. [La Figura 7-9](#) muestra la disposición final del array (y su representación de árbol equivalente) tras la inserción. Recuerde que siempre empezamos insertando nuevos elementos al final del array y luego los intercambiamos hacia adelante hasta que se restablezca la propiedad del montículo. En [la Figura 7-9](#), las flechas representan la ruta del nuevo elemento a través del array hasta su posición final. También se muestra la representación de árbol, con sombreado que indica los nodos modificados.

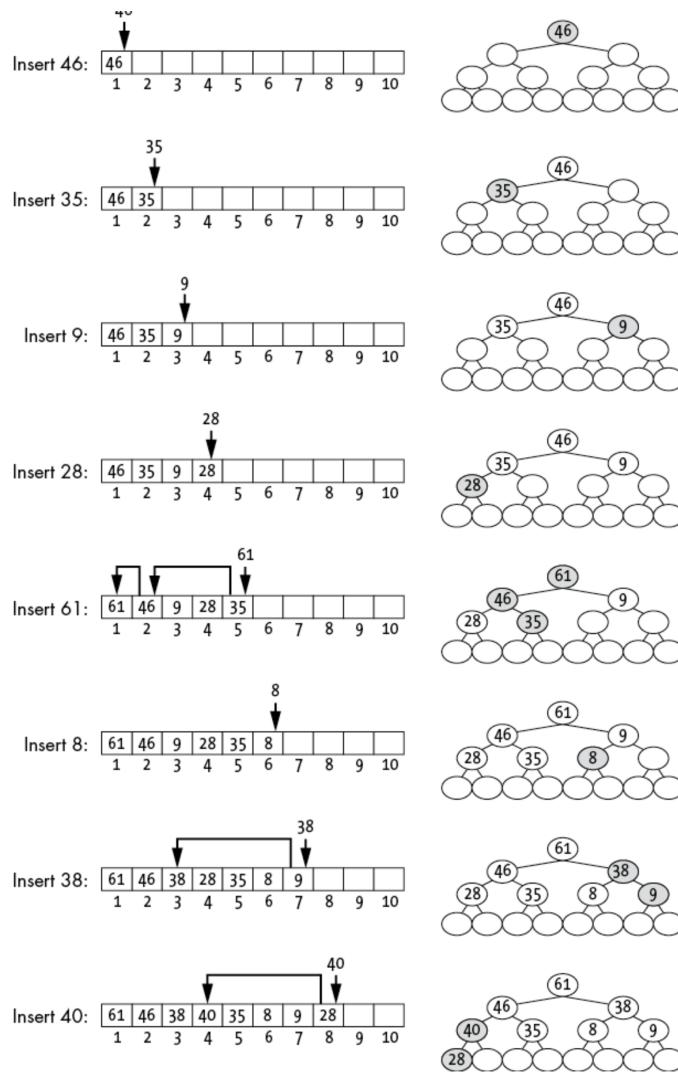
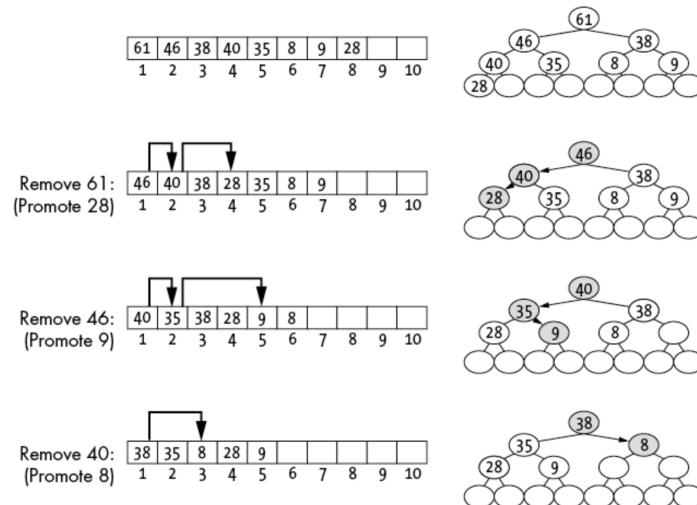


Figura 7-9: La primera etapa de ordenamiento del montón, donde los elementos de la matriz no ordenada se agregan al montón uno a la vez.

Limitamos el tiempo de ejecución de la creación del montón utilizando el tiempo de ejecución en el peor de los casos de una sola inserción. Como vimos anteriormente en este capítulo, en el peor de los casos, insertar un nuevo elemento en un montículo que contiene N elementos puede escalarse proporcionalmente a $\log^2(N)$. Por lo tanto, para construir un montículo con N elementos, *limitamos el tiempo de ejecución en el peor de los casos* proporcional a $\log^2(N)$.

Ahora que hemos construido nuestro montón, procedemos a la segunda etapa y extraemos cada elemento, como se muestra en [la Figura 7-10](#).



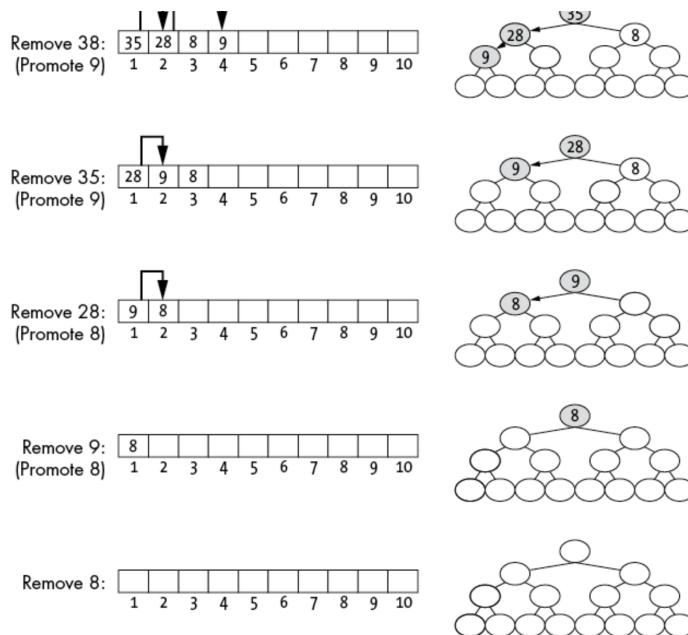


Figura 7-10: La segunda etapa de ordenamiento del montón, donde el elemento máximo se elimina repetidamente del montón con cada iteración

Eliminamos los elementos del montón uno por uno, en orden de prioridad decreciente. Esto produce el orden decreciente. En cada paso, se extrae la raíz, el último elemento del montón se intercambia a la posición de la raíz y, a continuación, la nueva raíz desciende a una posición que restaura la propiedad del montón. La figura muestra el estado del array (y la representación equivalente en árbol) al final de cada iteración. Las flechas en los diagramas y los nodos sombreados ilustran el recorrido del nodo sobrepromocionado a través del montón. A medida que extraemos los elementos, los añadimos directamente al array que almacena el resultado. Una vez vaciado el montón, lo descartamos. Ha cumplido su propósito.

Al igual que con las inserciones, podemos limitar el tiempo de ejecución en el peor de los casos como proporcional a $N \log^2(N)$. Cada extracción requiere como máximo $\log^2(N)$ para restaurar la propiedad del montículo, y necesitamos extraer todos los N elementos de nuestra lista ordenada. Por lo tanto, el tiempo de ejecución total en el peor de los casos del algoritmo de ordenamiento por montículo es proporcional a $N \log^2(N)$.

Por qué esto importa

Los montículos son una variación simple de los árboles binarios que permiten un conjunto diferente de operaciones computacionalmente eficientes. Al cambiar la propiedad del árbol de búsqueda binaria por la propiedad del montículo, podemos modificar el comportamiento de la estructura de datos y admitir un conjunto diferente de operaciones.

Tanto la adición de nuevos elementos como la eliminación del elemento máximo requieren recorrer como máximo una ruta entre la parte superior e inferior del árbol. Dado que podemos duplicar aproximadamente el número de nodos en un montículo añadiendo solo un nivel de nuevos nodos en la parte inferior, incluso los montículos grandes permiten operaciones rápidas. Duplicar el número de nodos de esta manera solo añade una iteración adicional a la inserción y la eliminación. Además, ambas operaciones garantizan que el árbol se mantenga equilibrado, lo que garantiza la eficiencia de las operaciones futuras.

Sin embargo, siempre existe una desventaja: al pasar de la propiedad del árbol de búsqueda binaria a la propiedad del montón, ya no podemos buscar eficientemente un valor específico. Optimizar para un conjunto de operaciones a menudo nos impide optimizar para otras. Debemos pensar detenidamente cómo usaremos los datos y diseñar su estructura en consecuencia.

Capítulo anterior

< [Capítulo 6: Pruebas y adaptaci...](#)

Siguiente capítulo

[Capítulo 8: Cuadriculas](#) >

