4
PILAS Y COLAS

Este capítulo presenta las pilas y las colas, dos estructuras de datos que recuperan los datos almacenados según el orden de inserción. Las pilas y las colas son muy similares y solo presentan pequeñas diferencias de implementación. Sin embargo, el simple hecho de que las pilas devuelvan primero los datos insertados más recientemente, mientras que las colas devuelvan los más antiguos, cambia por completo el comportamiento de los algoritmos y la eficiencia con la que podemos acceder a los datos.

Las pilas constituyen el núcleo de la búsqueda en profundidad, que busca cada vez más a fondo a lo largo de una ruta individual hasta llegar a un punto muerto. Las colas permiten la búsqueda en amplitud, que explora superficialmente las rutas adyacentes antes de profundizar. Como veremos más adelante, este cambio puede afectar drásticamente el comportamiento en el mundo real, como la forma en que navegamos por páginas web o investigamos sobre café.

Pilas

Una *pila* es una estructura de datos *LIFO (último en entrar, primero en salir)* que funciona de forma similar a una pila de papeles: añadimos nuevos elementos a la parte superior de la pila y eliminamos elementos comenzando por la parte superior. Formalmente, una pila admite dos operaciones:

Empujar Agrega un nuevo elemento a la parte superior de la pila.
Pop Quita el elemento de la parte superior de la pila y devuélvelo.

Dado que los elementos se extraen de la parte superior de la pila, el siguiente elemento eliminado siempre será el añadido más recientemente. Si insertamos los elementos 1, 2, 3, 4 y 5 en una pila, los recuperamos en el orden 5, 4, 3, 2, 1.

Puedes imaginar las pilas como si fueran contenedores de lechuga en un bufé de ensaladas mal gestionado, donde los contenedores se vacían cada pocos años. Los camareros vierten lechuga nueva continuamente encima del contenedor, sin prestar atención a la masa cada vez más blanda que se acumula debajo. Los comensales ven la lechuga nueva encima y la sirven en sus platos, ajenos a los horrores que se encuentran unas capas más abajo.

Podemos implementar pilas con matrices o listas enlazadas.

Pilas como matrices

Al implementar una pila como una matriz, usamos la matriz para almacenar los valores en la pila y usamos una variable adicional para rastrear el índice correspondiente a la parte superior de la pila, el último elemento de nuestra matriz:

```
Stack {
    Integer: array_size
    Integer: top
    Array of values: array
}
```

Inicialmente establecemos el `top` índice en −1, lo que indica que no hay nada en la pila.

Al insertar un nuevo elemento en la pila, incrementamos el `top` índice al siguiente espacio y añadimos el nuevo valor a dicho índice. De esta forma, ordenamos el array de abajo a arriba, como se muestra en *la Figura 4-1* . Si el último elemento del array es lechuga fresca y crujiente, el primer elemento del array representa los elementos que se encuentran en la parte inferior de la pila.

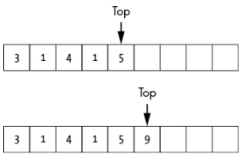

Figura 4-1 : Empujar un elemento en la parte superior de una pila representada como una matriz

Al agregar elementos a un array de tamaño fijo, debemos tener cuidado de no agregar más elementos de los que permite el espacio. Si nos quedamos sin espacio, podemos expandir el array con una técnica como la duplicación de arrays (véase el Capítulo 3), como se muestra en el siguiente código. Esto permite que nuestra pila crezca a medida que añadimos datos, aunque tenga en cuenta que esto supone un coste adicional para algunas inserciones.

```
Push(Stack: s, Type: value):
    IF s.top == s.array_size − 1:
        Expand the size of the array
    s.top = s.top + 1
    s.array[s.top] = value
```

Este código para insertar un elemento en una pila implementada como un array comienza comprobando que haya espacio para insertar el nuevo elemento. De no ser así, el código expande el array. A continuación, incrementa el índice del elemento superior e inserta el valor en ese nuevo índice.

Al extraer un elemento de la pila, volvemos a usar el `top` índice para encontrar el elemento correcto. Eliminamos este elemento del array y decrementamos el `top` índice, como se muestra en *la Figura 4-2* . En otras palabras, extraemos la lechuga más nueva del contenedor y la acercamos un nivel a las capas inferiores, más

antiguas.

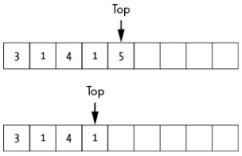

Figura 4-2 : Extraer un elemento de una pila representada como una matriz

El código para extraer un elemento de una pila implementada como una matriz es más simple que el código para la inserción:

```
Pop(Stack: s):
    Type: value = null
    IF s.top > -1:
        value = s.array[s.top]
        s.top = s.top – 1
    return value
```

El código comienza comprobando que la pila no esté vacía. De no ser así, copia el último elemento del array en [ ] `value` y luego decrementa el puntero al último elemento. El código devuelve el valor del elemento superior o `null` si la pila estaba vacía. Como solo añadimos o eliminamos elementos del final del array, no es necesario desplazar ningún otro elemento.

Mientras nuestra pila tenga suficiente espacio, podemos realizar adiciones y eliminaciones con un coste constante. Independientemente de si tenemos 10 elementos o 10 000, añadir o eliminar un elemento requiere el mismo número de operaciones. Sin embargo, aumentamos el coste al expandir el tamaño de un array durante una inserción, por lo que conviene preasignar una pila lo suficientemente grande para el caso de uso.

Pilas como listas enlazadas

Como alternativa, podríamos implementar una pila como una lista enlazada o una lista doblemente enlazada, como se muestra en *la Figura 4-3* . Aquí, la lista se dibuja de izquierda a derecha, en orden inverso al de las listas de capítulos anteriores, para mostrar el mismo orden que la representación del array. Nuestro puntero estándar a la cabecera de la lista también sirve como puntero a la parte superior de la pila.

```
Stack {
    LinkedListNode: head
}
```

En lugar de llenar nuevos contenedores de matriz y actualizar índices, la implementación de la lista vinculada requiere que creemos y eliminemos nodos en la lista vinculada, actualicemos los punteros de los nodos respectivos y actualicemos el puntero a la parte superior de la pila.

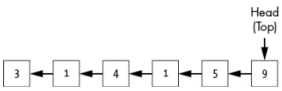

Figura 4-3 : Una pila implementada como una lista enlazada

Empujamos elementos a la pila agregándolos al frente de nuestra lista vinculada:

```
Push(Stack: s, Type: value):
    LinkedListNode: node = LinkedListNode(value)
    node.next = s.head
    s.head = node
```

El código para enviar comienza creando un nuevo nodo de lista enlazada. Luego, inserta este nodo al principio de la lista actualizando su `next` puntero y el de la pila `head`.

Similarly, when we pop an item from the stack, we return the value in the head node and move the head node pointer to the next item in our list:

```
Pop(Stack: s):
    Type: value = null
    IF s.head != null:
        value = s.head.value
        s.head = s.head.next
    return value
```

The code starts with a default return value of `null`. If the stack is not empty (`s.head != null`), the code updates the return value to be the head node's value and then updates the head pointer to the next node on the stack. Finally, it returns `value`.

Along with the memory cost of storing additional pointers, the pointer assignments add a small, constant cost to both the push and pop operations. We're no longer setting a single array value and incrementing an index. However, as with all dynamic data structures, the tradeoff is increased flexibility: a linked list can grow and shrink with the data. We no longer have to worry about filling up our array or paying the additional costs to increase the array's size.

Queues

A *queue* is a *first-in, first-out (FIFO)* data structure that operates like the line at your favorite coffee bar: we add new elements at the back of the queue and remove old elements from the front. Formally a queue supports two operations:

Enqueue Add a new element to the back of the queue.
Dequeue Remove the element from the front of the queue and return it.

If we enqueue five elements in the order 1, 2, 3, 4, 5, we would retrieve them in the same order: 1, 2, 3, 4, 5.

Queues preserve the order in which elements are added, allowing such useful behavior as processing items in the order they arrive. For example, the FIFO property allows our favorite café to serve its customers in an organized fashion. Due to its amazing menu, this shop always has a line of excited customers waiting for their morning brew. New customers enter the store and enqueue at the back of the line. The next customer to be served is the person at the front of the line. They place their order, dequeue from the front of the line, and eagerly await the perfect start to their morning.

Like stacks, queues can take the form of both arrays and linked lists.

Queues as Arrays

To implement queues with arrays, we track two indices: the first and last element in the queue. When we enqueue a new element, we add it behind the current last element and increment the back index, as shown in *Figure 4-4*.

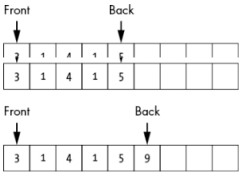

Figura 4-4 : Colocación de un elemento en una cola representada como una matriz

Cuando quitamos un elemento de la cola, eliminamos el elemento frontal e incrementamos el índice frontal en consecuencia, como se muestra en *la Figura 4-5* .

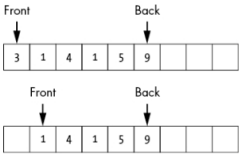

Figura 4-5 : Eliminación de un elemento de una cola representada como una matriz

When dequeuing from a fixed array, we quickly run into a drawback: a block of empty space will accumulate at the front of the array. To solve this problem, we can either wrap the queue around the end of the array or shift items down to fill up the space. As we saw in Chapter 1, shifting elements is expensive, since we have to move all the remaining elements with each dequeue operation. Wrapping is a better solution, though it does require us to carefully handle indices being incremented past the end of the array during both enqueuing and dequeuing, as shown in *Figure 4-6*.

Front Back

quiere un manejo cuidadoso de los índices que se incrementan más allá del final del array durante la encola y la desencola, como se muestra en *la Figura 4-6* .

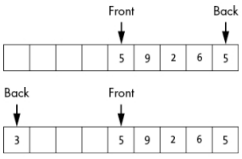

Figura 4-6 : El uso de una matriz para representar colas puede provocar que los elementos se envuelvan.

Si bien el envoltorio agrega cierta complejidad a nuestra implementación, evita el alto costo de mover elementos.

Colas como listas enlazadas

Es mejor implementar la cola como una lista enlazada o una lista doblemente enlazada. Además de mantener un puntero especial al principio de la lista (el principio de la cola), mantenemos uno al último elemento de la lista, la cola o la parte final:

```
Queue {
    LinkedListNode: front
    LinkedListNode: back
}
```

Esta lista, que se muestra en *la Figura 4-7* , es similar a la lista enlazada que usamos para la pila en *la Figura 4-3* . Cada elemento de la cola se enlaza con el elemento inmediatamente posterior, lo que nos permite recorrer los `next` punteros desde el principio hasta el final de la cola.

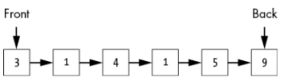

Figura 4-7 : Una cola representada como una lista enlazada con punteros adicionales para los elementos frontal y posterior

Una vez más, las inserciones y eliminaciones requieren que actualicemos tanto los nodos en la lista como nuestros nodos punteros especiales:

```
Enqueue(Queue: q, Type: value):
    LinkedListNode: node = LinkedListNode(value)
 ❶ IF q.back == null:
        q.front = node
        q.back = node
```

```
    ELSE:
      ❷ q.back.next = node
      ❸ q.back = node
```

Al añadir un nuevo elemento a la cola, usamos el `back` puntero de la cola para encontrar la ubicación correcta de la inserción. El código comienza creando un nuevo nodo para el valor insertado y luego comprueba si la cola está vacía ❶ . De ser así, el código añade el nuevo nodo configurando los punteros `front` y `back` de la cola para que apunten a él. Ambos punteros deben actualizarse, ya que, de lo contrario, no apuntarían a nodos válidos.

Si la cola no está vacía, el código añade el nuevo nodo al final de la lista modificando el `next` puntero del último nodo actual para que apunte a nuestro nuevo nodo ❷ . Finalmente, el código actualiza el `back` puntero de la cola para que apunte a nuestro nuevo nodo, indicando que es el último nodo de la cola ❸ . El `front` puntero no cambia a menos que la cola estuviera vacía previamente.

Las eliminaciones actualizan principalmente el puntero al frente de la cola:

```
Dequeue(Queue: q):
  ❶ IF q.front == null:
        return null

  ❷ Type: value = q.front.value
  ❸ q.front = q.front.next
    IF q.front == null:
        q.back = null
    return value
```

The code first checks whether there is anything in the queue by testing whether the queue's `front` pointer actually points to anything or is `null` ❶. If the queue is empty (`q.front == null`), the code immediately returns `null`. If there is at least one element in the queue, the code saves that value to return later ❷. Then the code updates `q.front` to point to the next element in the queue ❸. We must take care to update the `back` pointer as well when dequeuing the last element. If the front item is no longer pointing to a valid element, then the queue is empty, and we set the back pointer to `null` as well.

Both the enqueue and dequeue operations require a constant number of operations, regardless of the size of the queue. Each operation requires us to adjust a few pointers. We don't care what else is in the data structure; we could be appending elements at the end of a list snaking through the entirety of the computer's memory.

The Importance of Order

The order in which we insert or remove elements can have a staggering impact nificativo en el comportamiento de los algoritmos (y, en el caso de la barra de ensaladas, en la salud de nuestros comensales). Las colas funcionan mejor cuando necesitamos que nuestro almacenamiento preserve el orden de las inserciones. Al procesar solicitudes de red entrantes, por ejemplo, queremos procesar primero las solicitudes anteriores. En cambio, usamos pilas cuando queremos procesar primero el elemento más reciente. Por ejemplo, los lenguajes de programación pueden usar pilas para procesar llamadas a funciones. Cuando se llama a una nueva función, el estado actual se coloca en una pila y la ejecución salta a la nueva función. Cuando una función se completa, el último estado se extrae de la pila y el programa regresa al punto donde se llamó a esa función.

Podemos modificar por completo el comportamiento de un algoritmo de búsqueda según la estructura de datos que elijamos. Imaginemos que estamos explorando nuestra enciclopedia en línea favorita para investigar métodos de molienda de café. Al recorrer la página sobre molinillos, vemos enlaces a otros fascinantes...Opciones. Si siguiéramos uno de esos enlaces, accederíamos a otra página de información con ramas de nuevos temas potenciales para explorar. El uso de una pila o una cola para rastrear los temas que queremos abordar más adelante influirá en la naturaleza de nuestra exploración del café, como veremos en las siguientes dos secciones.

Búsqueda en profundidad

La búsqueda en profundidad es un algoritmo que continúa explorando, cada vez más a fondo, a lo largo de una única ruta hasta llegar a un punto muerto. El algoritmo retrocede entonces hasta la última ramificación y comprueba otras opciones. Mantiene una lista de estados futuros para explorar mediante una pila, eligiendo siempre la opción insertada más recientemente para probar a continuación.

In the coffee research example, suppose we start a depth-first search at the page queda en profundidad en la página sobre molinillos y encontramos rápidamente tres temas adicionales. Los agregamos a la pila, como se muestra en *la Figura 4-8* (1). La mayoría de las búsquedas añadirán las opciones en el orden en que se encuentren. Para mantener la coherencia en este ejemplo, añadimos los elementos en orden alfabético inverso, de modo que comenzamos nuestra investigación en la A y continuamos hasta la Z (o la letra que corresponda al último tema).

Para simplificar, *la Figura 4-8* muestra cada tema (página web) como una letra individual. Las líneas que los separan representan los enlaces web. Esta estructura se conoce como *grafo* , que analizaremos en detalle en el Capítulo 15. Los nodos sombreados indican los temas explorados, y el tema encerrado en un círculo indica el tema que estamos examinando en esa iteración de la búsqueda. Los datos de la pila, que almacenan las futuras opciones para explorar, se representan como la `Next` matriz a la derecha del grafo.

Una vez que hemos leído todo lo posible sobre los molinillos de muelas (A), pasamos al siguiente tema: los molinillos de cuchillas (B). En este punto, nuestra búsqueda está a la espera de explorar los temas B, F y H. Tomamos el tema B del principio de la lista, abrimos esa página, comenzamos a leer y encontramos aún más temas de interés (C). El café es, sin duda, un tema profundo y complejo; podríamos dedicarnos toda la vida a investigar hasta el último detalle. Colocamos el nuevo tema C en la parte superior de la lista para futuras investigaciones, como se muestra en *la Figura 4-8* (2). De este modo, los temas más recientes están preparados para ser las siguientes áreas de exploración.

La búsqueda continúa de esta manera, priorizando los temas añadidos recientemente, lo que la hace ideal para quienes siempre desean explorar el tema más reciente. Profundizamos en cada hilo hasta llegar a un punto muerto y volvemos a los elementos anteriores de la pila. En este ejemplo, los nodos visitados nunca se revisan ni se añaden a la pila, pero esta puede contener duplicados. Las subfiguras restantes de *la Figura 4-8* ilustran este proceso.

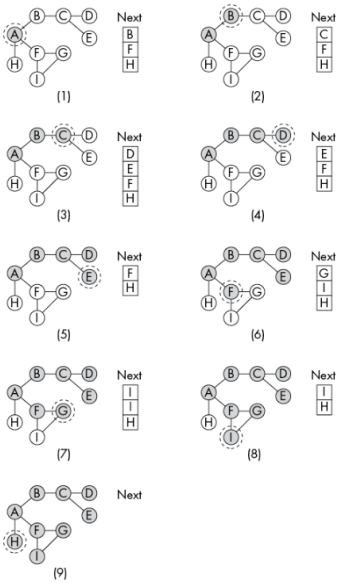

Figura 4-8 : Búsqueda en profundidad: exploración de un gráfico de temas utilizando una pila para rastrear el siguiente tema a explorar

Búsqueda en amplitud

La búsqueda en amplitud utiliza una lógica similar a la de la búsqueda en profundidad para explorar temas, pero almacena los estados futuros en una cola. En cada paso, la búsqueda explora la opción que lleva más tiempo esperando, ramificándose en diferentes direcciones antes de profundizar. *La Figura 4-9* muestra una búsqueda en amplitud del sitio web relacionado con molinillos de café de la sección anterior. De nuevo, los círculos representan temas y las líneas son los vínculos entre ellos. Los temas sombreados ya se han explorado, y el tema en círculo es el que estamos investigando durante esa iteración.

In *Figure 4-9*(1), we read a page about burr grinders (A) and note down three future topics of interest (B, F, and H) in reverse alphabetical order. The first topic in the queue is the same as the last in the stack—this is the key ordering difference between the two data structures.

We continue our research, taking the item at the front of the queue (H), reading the page, and adding any new items of interest to the back of the queue. In *Figure 4-9*(2), we explore the next page, only to hit an immediate dead end.

In *Figure 4-9*(3), the search progresses to the next topic in the queue (F). Here we find new topics (I and G) and add the new links to the queue.

Instead of following those topics further, we take the next item (B) from the front of the queue, as shown in *Figure 4-9*(4), and explore the final link from the initial page. Again, the search only adds unexplored nodes that are not already in the queue.

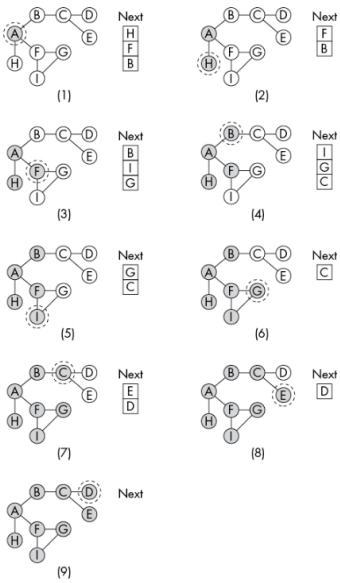

Figura 4-9 : Búsqueda en amplitud: exploración de un gráfico de temas utilizando una cola para rastrear el siguiente tema a explorar

A medida que avanza la búsqueda, sus ventajas se hacen evidentes: en lugar de
A medida que avanza la búsqueda, sus ventajas se hacen evidentes: en lugar de

explorar a fondo cada hilo antes de volver a los anteriores, exploramos a lo largo de una frontera temática, priorizando la amplitud sobre la profundidad. Podríamos examinar cinco tipos diferentes de molinillos antes de profundizar en la historia de cada mecanismo de molienda y sus respectivos inventores. Esta búsqueda es ideal para quienes no quieren dejar temas antiguos pendientes y prefieren descartarlos antes de pasar a temas nuevos.

Dado que tanto la búsqueda en profundidad como la búsqueda en amplitud solo exploran una opción a la vez, ambas funcionan a la misma velocidad: la búsqueda en profundidad explora a fondo unas pocas rutas en el tiempo que la búsqueda en amplitud tarda en explorar superficialmente muchas. Sin embargo, el comportamiento real de la búsqueda es radicalmente diferente.

Por qué esto importa