4
STACKS AND QUEUES

This chapter introduces stacks and queues, two data structures that retrieve stored data based on the order in which it was inserted. Stacks and queues are very similar and require only minor implementation differences. However, the simple fact that stacks return the most recently inserted data first, while queues return the oldest, completely changes the behavior of algorithms and the efficiency with which we can access data.

Stacks form the core of depth-first search, which searches deeper and deeper along an individual path until it hits a dead end. Queues enable breadth-first search, which shallowly explores adjacent paths before digging deeper. As we will see later, this one change can dramatically impact real-world behavior, such as how we surf web pages or conduct coffee research.

Stacks

A *stack* is a *last-in, first-out (LIFO)* data structure that operates much like a pile of papers: we add new elements to the top of the stack and remove elements starting with the top of the stack. Formally, a stack supports two operations:

Push Add a new element to the top of the stack.
Pop Remove the element from the top of the stack and return it.

Since items are extracted from the top of the stack, the next item removed will always be the one most recently added. If we insert the elements 1, 2, 3, 4, and 5 into a stack, we retrieve them in the order 5, 4, 3, 2, 1.

You can visualize stacks as lettuce bins in a suboptimally run salad bar where the bins are cleaned out every few years. The waiters continually dump new lettuce on the top of the bin, paying no attention to the increasingly squishy mass of lettuce accumulating underneath. Diners see the new lettuce on top and scoop it onto their plates, oblivious to the horrors a few layers below.

We can implement stacks with either arrays or linked lists.

Stacks as Arrays

When implementing a stack as an array, we use the array to hold the values in the stack and use an extra variable to track the index corresponding to the top of the stack—the last element in our array:

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

Initially we set the `top` index to −1, indicating that there is nothing in the stack.

When we push a new element onto the stack, we increment the `top` index to the next space and add the new value to that index. We thus order the array from bottom to top, as shown in *Figure 4-1*. If the last item of the array is fresh, crispy lettuce, the first element of the array represents the items languishing at the bottom of the stack.

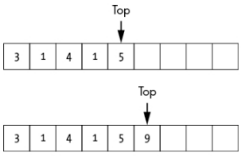


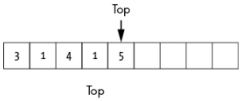
Figure 4-1: Pushing an element on top of a stack represented as an array

When adding elements to an array of fixed size, we must be careful to avoid adding more elements than space allows. If we run out of space, we might expand the array with a technique such as array doubling (see Chapter 3), as shown in the following code. This allows our stack to grow as we add data, though be aware that it introduces additional cost for some insertions.

```
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
```

This code for pushing an element onto a stack implemented as an array starts by checking that we have room to insert the new element. If not, the code expands the array. The code then increments the index of the top element and inserts the value at that new index.

When we pop an item off the stack, we again use the `top` index to find the correct element. We remove this element from the array and decrement the `top` index, as shown in *Figure 4-2*. In other words, we scoop the newest lettuce from the bin and move one level closer to the lower, older layers.



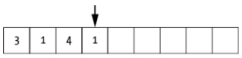


Figure 4-2: Popping an element from a stack represented as an array

The code for popping an element from a stack implemented as an array is simpler than the code for insertion:

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

The code starts by checking that the stack is not empty. If it isn't, the code copies the last element of the array into `value`, then decrements the pointer to the last element. The code returns the value of the top element or `null` if the stack was empty. Since we are only adding or removing items from the end of the array, we don't need to shift around any other elements.

As long as our stack has sufficient room, we can perform both additions and removals at a constant cost. Whether we have 10 elements or 10,000, adding or removing an element requires the same number of operations. However, we do pay additional cost when expanding the size of an array during an insertion, so it helps to preallocate a stack that is large enough for the use case.

Stacks as Linked Lists

Alternately, we could implement a stack as either a linked list or doubly linked list, as shown in *Figure 4-3*. Here the list is drawn left to right, the reverse order of lists in previous chapters, to show the same order as the array representation. Our standard pointer to the head of the list also serves as the pointer to the top of the stack.

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

Instead of filling in new array bins and updating indices, the linked list implementation requires us to create and remove nodes in the linked list, update the respective node pointers, and update the pointer to the top of the stack.

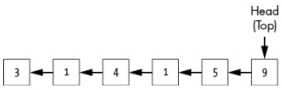


Figure 4-3: A stack implemented as a linked list

We push items on to the stack by adding them to the front of our linked list:

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

The code for pushing starts by creating a new linked list node. Then it inserts this node into the front of the list by updating the new node's `next` pointer and the stack's `head` pointer.

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 per-

fect 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*.

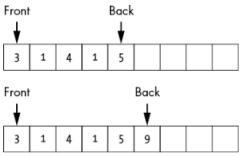


Figure 4-4: Enqueuing an element in a queue represented as an array

When we dequeue an element, we remove the front element and increment the front index accordingly, as shown in *Figure 4-5*.

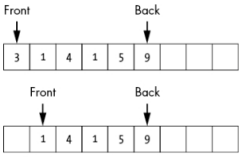


Figure 4-5: Dequeuing an element in a queue represented as an array

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*.

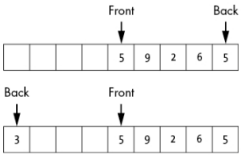


Figure 4-6: Using an array to represent queues can result in the elements wrapping around.

While wrapping adds some complexity to our implementation, it avoids the high cost of shifting elements.

Queues as Linked Lists

It's a better idea to implement the queue as a linked list or a doubly linked list. In addition to maintaining a special pointer to the head of the list (the front of the queue), we maintain one to the last element in the list, the tail or back:

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

This list, shown in *Figure 4-7*, is similar to the linked list we used for the stack in *Figure 4-3*. Each element in the queue links to the element immediately behind it, allowing us to traverse the `next` pointers from the front of the queue to the back.

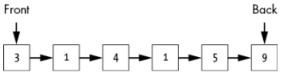


Figure 4-7: A queue represented as a linked list with additional pointers for the front and back elements

Once again, insertions and deletions require us to update both the nodes in the list and our special pointer nodes:

```
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
```

When we add new element to the queue, we use the queue's `back` pointer to find the correct location for the insertion. The code starts by creating a new node for the inserted value, then checks whether the queue is empty ❶. If it is, the code adds the new node by setting the queue's `front` and `back` pointers to point to the new node. Both pointers need to be updated because otherwise they wouldn't be pointing to valid nodes.

If the queue is not empty, the code appends the new node to the end of the list by modifying the current last node's `next` pointer to point to our new node ❷. Finally, the code updates the queue's `back` pointer to point to our new node, indicating it is the new last node in the queue ❸. The `front` pointer doesn't change unless the queue was previously empty.

Deletions update primarily the pointer to the front of the queue:

```
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 on the behavior of algorithms (and, in the case of the salad bar, the health of our diners). Queues work best when we need our storage to preserve the ordering of insertions. When processing incoming network requests, for example, we want to process earlier requests first. In contrast, we use stacks when we want to process the most recent item first. For instance, programming languages may use stacks to process function calls. When a new function is called, the current state is pushed onto a stack and execution jumps into the new function. When a function completes, the last state is popped off the stack and the program returns to where that function was called.

We can shift the entire behavior of a search algorithm depending on which data structure we choose. Imagine we are exploring our favorite online encyclopedia to research coffee-grinding methods. As we scan down the page on burr grinders, we see links to other fascinating options. If we were to follow one of those links, we'd reach another page of information along with branches of new potential topics to explore. Whether we use a stack or queue to track the topics we want to pursue later will impact the nature of our coffee exploration, as we'll see in the next two sections.

Depth-First Search

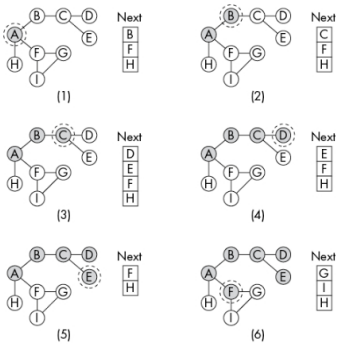
Depth-first search is an algorithm that continues exploring, deeper and deeper, along a single path until it hits a dead end. The algorithm then backs up to the last branching point and checks other options. It maintains a list of future states to explore using a stack, always choosing the most recently inserted option to try next.

In the coffee research example, suppose we start a depth-first search at the page on burr grinders and quickly find three additional topics to pursue. We push these onto the stack, as shown in *Figure 4-8*(1). Most searches will add options in the order in which they are encountered. For consistency in this example, we add elements in reverse alphabetical order, so that we begin our research at A and continue to Z (or whatever letter is the final topic).

For simplicity's sake, *Figure 4-8* shows each of the topics (web pages) as an individual letter. The lines drawn between them represent the web links. This structure is known as a *graph*, which we'll discuss in detail in Chapter 15. Shaded nodes indicate topics we have explored, and the circled topic indicates the topic we're examining on that iteration of the search. The stack data, storing the future options to explore, is represented as the `Next` array to the graph's right.

Once we've finished reading everything we can about burr grinders (A), we move on to the next topic: blade grinders (B). At this point, our search is waiting to explore topics B, F, and H. We take topic B from the top of the stack, open that page, start reading, and find even more topics of interest (C). Coffee is clearly a deep and complex topic; we could spend a lifetime researching the finest details. We push the new topic C onto the top of the stack for future investigation, as shown in *Figure 4-8*(2). Newer topics are thus primed to be the next areas of exploration.

The search continues in this way, prioritizing recently added topics, making this search ideal for someone who always wants to explore the most recent topic they've seen. We explore deeper and deeper into each topic thread until we hit a dead end and return to earlier items on the stack. For this example, visited nodes are never revisited or added to the stack, but the stack may contain duplicates. The remaining subfigures in *Figure 4-8* illustrate this process.



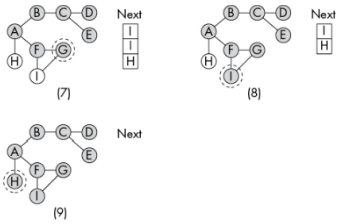


Figure 4-8: Depth-first search: exploring a graph of topics using a stack to track the next topic to explore

Breadth-First Search

Breadth-first search uses logic similar to that of a depth-first search to explore topics, but stores future states with a queue. At each step, the search explores the option that has been waiting the longest, effectively branching out over different directions before going deeper. *Figure 4-9* shows a breadth-first search of the coffee grinder–related website from the last section. Again, the circles represent topics, and lines are the links between them. Shaded topics have been explored, and the circled topic is the topic we are researching during that iteration.

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.

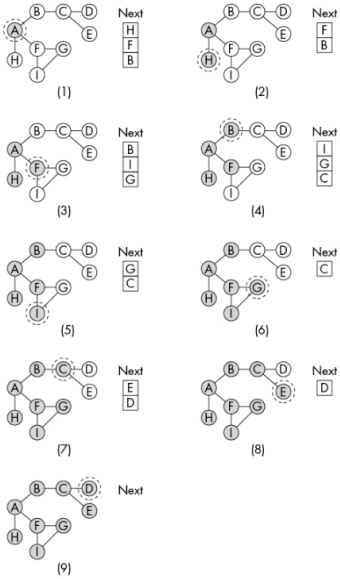


Figure 4-9: Breadth-first search: exploring a graph of topics using a queue to track the next topic to explore

As the search progresses, its advantages become clear: instead of deeply exploring each thread before returning to previous ones, we explore along a frontier of topics, prioritizing breadth over depth. We might survey five different types of grinders before diving into the histories of each grinding mechanism and their respective inventors. This search is ideal for people who don't want to let old topics linger and prefer to cross them off before moving onto new topics.

Since both depth-first search and breadth-first search only explore one option at a time, both work at the same rate: depth-first search deeply explores a few paths in the time it takes breadth-first search to shallowly explore many paths. However, the actual behavior of the search is radically different.

Why This Matters

Different data structures both allow the programmer to use data in different ways and strongly impact the behavior of the algorithms working with that data. Both stacks and queues store objects, both can be implemented with either arrays or linked lists, and both can handle insertions and removals efficiently. From the perspective of simply storing bits, either will suffice. However, the way they handle the data, and specifically the order in which they return their items, gives these similar data structures radically different behavior. Stacks return the newest data they've stored, making them ideal for prioritizing the most recent items. Queues, in contrast, always return the oldest data they've stored, making them ideal for cases where we need to handle items in the order in which they arrive.

As we work to use data structures effectively, efficiency isn't our only consideration. When designing or choosing a data structure for a specific algorithm, we must ask how that data structure's properties will impact the behavior of the algorithm. As the search examples in this chapter show, we can change from breadth-first to depth-first behavior by simply swapping out the data structure. Later

chapters give more detail on this logic, using it to design other data structures that aid both the behavior and performance of our algorithms.