1

So the Last shall be First, and the First Last

Linear data structures: stack, queue, and priority queue

In the second part (out of the three parts) of the course, we will concentrate on fundamental data structures, how to organize data for more efficient problem solving. The first type of data structure is index-based data structures, such as lists and hashtables. Each element is accessed by an *index*, which points to the position the element within the data structure. This is covered in Chapter 6 of the primary textbook[Conery(2011)].

In the three chapters of this handout, we will be expanding our coverage to three other types of data structures. In Chapter 1 (this chapter), we will explore *linear* data structures, whereby each element is accessed sequentially in a particular order. In the next two chapters, we will look at a couple of other data structures, where the main objective is to store the relationship between elements. In Chapter 2, we will look at binary trees. In Chapter 3, we will look at graphs.

There are several linear data structures that we will explore in this chapter. Unlike index-based data structures where any element can be accessed by its index, in the linear data structures that we will discuss in this chapter, only one element can be accessed at any point of time. These data structures differ in the order in which the elements are accessed or retrieved. In *stacks*, the last item inserted will be the first to be retrieved. In *queues*, the first item inserted will also be the first item retrieved. We will also briefly cover *priority queues*, which maintain a sorted order of elements at all times, and retrieve the item with the highest priority order first.

1.1 Stack

We are familiar with the term ”stack” as used in the English language. It literally means a pile of objects. We talk about a stack of coins, a stack of books, a stack of boxes, etc. Common among all these real-world scenarios is that the single object in a stack that is easiest to access is the one at the top.

This nature of stacks is known as LIFO, which stands for Last-In, First-Out. Using stack as a data structure means that we take care to ensure that the item we need to access first will be put onto the stack last.

Access to any element in a stack has to go through one of the following operations.

- `push` operation inserts a new element onto the top of the stack.
- `peek` operation returns the element currently at the top of the stack, but does not remove it.
- `pop` operation returns the element currently at the top of the stack and removes it from the stack. As a result, the element previously below the top element now becomes the top element itself.

In addition to the above operations that are valid only for stack, all the linear data structures also have `new` operation, which creates a new instance of the data structure, as well as `count` operation, which returns the number of elements currently in the data structure.

Tutorial Project

Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'is103'
=>  true

>>  include LinearDSLab
=>  Object
```

T1. Make a new stack object and save it in a variable named `s`:

```
>>  s = Stack.new
=>  []
```

This new stack is currently empty.

T2. Let’s add a couple of strings to the new stack:

```
>>  s.push("grape")
=>  ["grape"]

>>  s.push("orange")
=>  ["orange", "grape"]
```

The top of the stack is the first element in the array. It is always the most recently added element. The bottom of the stack or the last element is the least recently added.

T3. Let us visualize the stack:

```
>>  view_stack(s)
=>  true
```

Can you see now there are two elements in the stack? Do the top and the bottom elements match what you expect?

T4. Add more strings to the stack:

```
>>  ["mango", "guava", "kiwi"].each { |x| s.push(x) }
=>  ["mango", "guava", "kiwi"]
```

T5. Check the strings in the stack currently:

```
>>  s
=>  ["kiwi", "guava", "mango", "orange", "grape"]

>>  s.count
=>  5
```

T6. We can find out what the topmost string in the stack is without removing it by using `peek`:

```
>>  s.peek
=>  "kiwi"
```

The return value is “kiwi”. The visualization now “highlights” the top element by changing its color, as shown in Figure 1.1. However, the element is not removed yet.

T7. Check that no string has been removed from the stack as a result of calling `peek`, and make sure there are still five strings in the stack:

```
>>  s
=>  ["kiwi", "guava", "mango", "orange", "grape"]

>>  s.count
=>  5
```

T8. Remove a string from the stack by using `pop` and store it in a variable named `t`:

```
>>  t = s.pop
=>  "kiwi"
```

Do you see how the visualization changes to indicate that “kiwi” is no longer part of the stack?

T9. Check that the topmost string has been removed from the stack `s` and is now stored in `t`:

```
>>  s
=>  ["guava", "mango", "orange", "grape"]

>>  t
=>  "kiwi"
```

T10. Find out what the topmost string is now without removing it:

```
>>  s.peek
=>  "guava"
```

This is the string added second last, which is now the topmost string after removing `"kiwi"`.

T11. Try a few more `push`, `peek`, and `pop` operations on your own.

A Simple Application of Stack

In computing, stacks are used in various applications. One application is to support the *Backspace* (on a PC) or *delete* (on a Mac) key on a computer keyboard. When you press this key, the last letter typed (Last-In) is the one deleted first (First-Out). Another application is to manage the function calls of a software program. A function f_1 may call another function f_2. In this case, f_1 may be “suspended” until f_2 completes. To manage this, the operating system maintains a stack of function calls. Each new function call is pushed onto the stack. Every time a function call completes, it is popped from the stack.

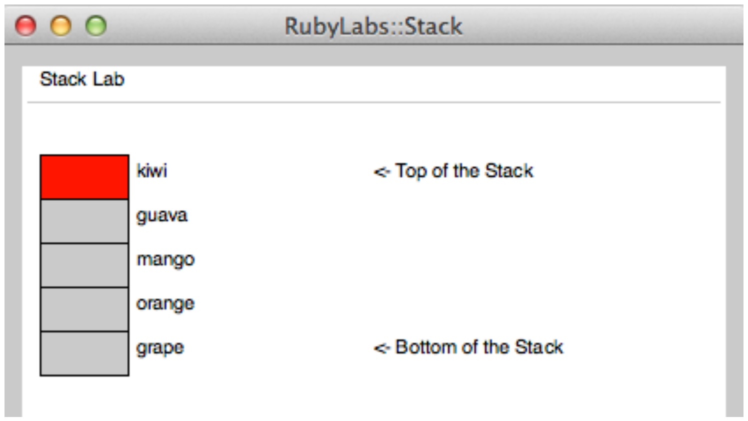


Figure 1.1: *Visualization of stack after* `peek` *is called*

To further appreciate how a stack is used in an application, we will explore one simple application: *checking for balanced braces*. Braces are balanced if there is a matching closing brace for each opening brace. For example, the braces in the string "a{b{c}d}" are balanced, whereas the braces in the string "ab{{c}d" are not balanced. This application is based on a similar example in [Prichard and Carrano(2011)].

Tutorial Project

T12. Create an array of strings with balanced braces:

```
>>  t = ["a", "{", "b", "{", "c", "}", "d", "}"]
=>  ["a", "{", "b", "{", "c", "}", "d", "}"]
```

T13. A simpler way to do it is by "splitting" a string into its individual characters using `split()` method of a string, passing it an argument `//` (which is the space between characters).

```
>>  t = "a{b{c}d}".split(//)
=>  ["a", "{", "b", "{", "c", "}", "d", "}"]
```

T14. We now create a new stack to experiment with this application:

```
>>  s = Stack.new
=>  []
```

T15. Traverse the array, and whenever we encounter an opening brace, we push it into the stack:

```
>>  t.each { |x| s.push(x) if x == "{" }
```

T16. Check that the stack now contains two opening braces:

```
>>  s
=>  ["{", "{"]
```

T17. Traverse the array again, and whenever we encounter a closing brace, we pop the topmost string from the stack:

```
>>  t.each { |x| s.pop if x == "}" }
```

T18. Check that the stack is now empty:

```
>>  s
=>  []
```

Because the number of opening and closing braces are equal, the same number of `push` and `pop` operations are conducted. Since the stack `s` is empty at the end, we conclude that the array `t` contains balanced braces.

T19. We now experiment with a new array with imbalanced braces:

```
>>  t = "ab{{c}d".split(//)
=>  ["a", "b", "{", "{", "c", "}", "d"]
```

T20. We run the previous push and pop operations in sequence:

```
>>  t.each { |x| s.push(x) if x == "{" }

>>  t.each { |x| s.pop if x == "}" }
```

T21. Check the current status of the stack:

```
>>  s
=>  ["{"]
```

In this case, there are two push operations, but only one pop operation, resulting in the stack containing an opening brace at the end. Since the resulting stack is not empty, we conclude that the array does not contain balanced braces.

T22. Try the same set of commands for a different sequence of strings:

```
>>  t = "ab}c{d".split(//)

>>  t.each { |x| s.push(x) if x == "{" }

>>  t.each { |x| s.pop if x == "}" }
```

T23. Check the current status of the stack:

```
>>  s
=>  []
```

The stack is empty, implying that the braces are balanced. However, this is not a correct conclusion, because although the numbers of opening and closing braces are the same, they are out of sequence (the closing brace comes before the opening brace).

The way of checking for balanced braces in the previous tutorial does not always produce the correct answer. This is because we push *all* the opening braces first, and then pop for *all* the closing braces. We have disregarded the order in which the braces occur.

Let us revisit what it means for an array to contain balanced braces. The braces are balanced if:

1. Each time we encounter a closing brace "}", there is already a previously encountered opening brace "{".
2. By the time we reach the end of the array, we have matched every "{" with a "}", and there is no remaining unmatched opening or closing brace.

The right way of doing it is to push *each* opening brace when we encounter one during the traversal of the array, and to pop the topmost opening brace as soon as we encounter *each* closing brace . Therefore, in the out-of-sequence case, there will be an occasion when we try to pop the stack when it is empty. This should be flagged as an imbalanced case.

The Ruby function shown in Figure 1.2 implements the above logic. It maintains a stack s and a boolean indicator balancedSoFar. It then iterates through the input array t, pushes each opening brace as it is encountered, and pops once for each closing brace. The key step to flag the out-of-sequence cases is to check that the stack is not empty when popping, otherwise the balancedSoFar is set as false. At the end, the array contains balanced braces if both the stack s is empty and balancedSoFar is true.

We will now try this code, and verify that it indeed can identify the various cases of balanced and imbalanced braces.

```
def check_braces(t)
  s = Stack.new
  balancedSoFar = true
  i = 0
  while i < t.count and balancedSoFar
    if t[i] == "{"
      s.push(t[i])
    elsif t[i] == "}"
      if s.count > 0
        s.pop
      else
        balancedSoFar = false
      end
    end
    i += 1
  end
  return (balancedSoFar and s.count==0)
end
```

when “{” encountered, push into stack

when “}” encountered, pop from stack

ensure that stack not empty when popping, otherwise not balanced

balanced if stack is empty at the end

Figure 1.2: *An algorithm to check balanced braces*

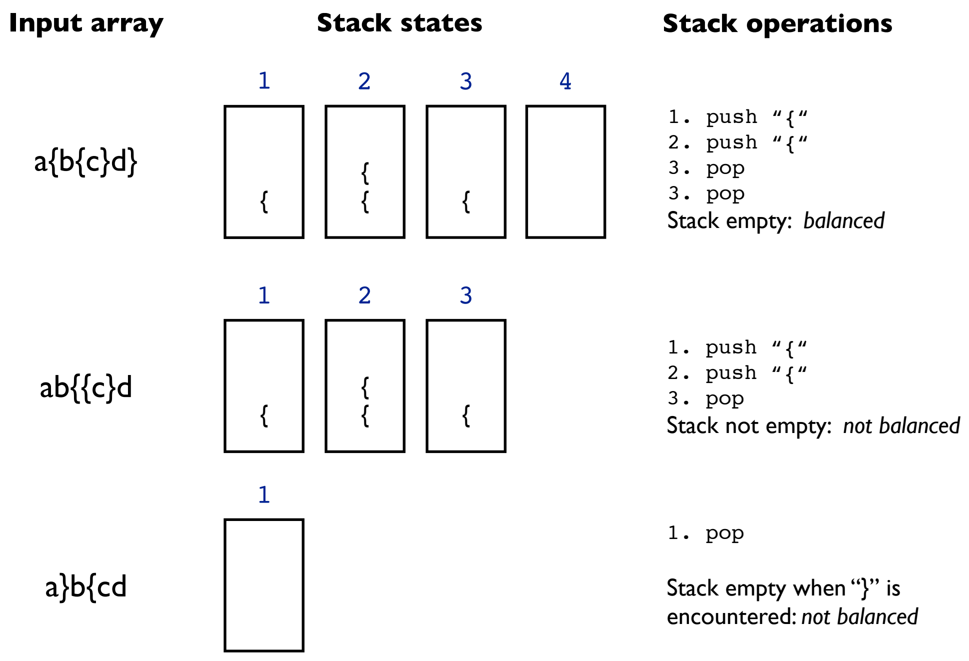


Figure 1.3: *Traces of the algorithm to check balanced braces*

Tutorial Project

T24. Create an array of strings with balanced braces:

```
>>  t = "a{b{c}d}".split(//)
=>  ["a", "{", "b", "{", "c", "}", "d", "}"]
```

T25. Use check_braces to verify that the braces are balanced:

```
>>  check_braces(t)
=>  true
```

The trace of execution for this array is shown in the top row of Figure 1.3. It illustrates the states of the stack after each push and pop operation. In Step 1, the brace "{" after ”a” is pushed onto the stack. In Step 2, the brace "{" after letter "b" is pushed, resulting in two opening braces in the stack. In Step 3, a pop operation is run after the first "}". Finally, in Step 4, another pop operation after the second "}". Since in the end, the stack is empty, the braces in t are balanced.

T26. Create a new array with imbalanced braces:

```
>>  t = "ab{{c}d".split(//)
=>  ["a", "b", "{", "{", "c", "}", "d"]
```

T27. Use check_braces to verify that the braces are not balanced:

```
>>  check_braces(t)
=>  false
```

The trace of execution for this array is shown in the middle row of Figure 1.3.

T28. Create an array with out-of-sequence braces:

```
>>  t = "a}b{cd".split(//)
=>  ["a", "}", "b", "{", "c", "d"]
```

T29. Use check_braces.rb to verify that the braces are not balanced although there is equal number of opening and closing braces:

```
>>  check_braces(t)
=>  false
```

The trace of execution for this array is shown in the bottom row of Figure 1.3.

Array-based Implementation of a Stack in Ruby

So far we have used a stack and its various operations without knowing how it is implemented, which is how it should be. As we discussed earlier in the course, an abstract data type (such as a stack) is defined by the operations, not by a specific implementation.

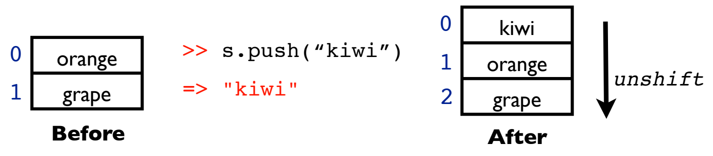
Here we will explore an implementation written in Ruby based on Array object. In this array-based implementation, we use an array variable called items to store the elements in the stack.

```
items = Array.new
```

We then define the operations in terms of array operations on this array items, as follows:

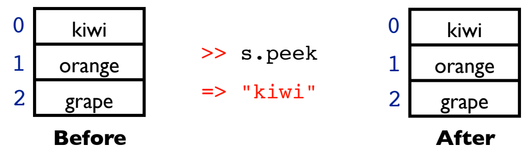
- Figure 1.4(a) shows the implementation of the push operation. It uses the operation unshift of an array, which inserts the new element v as the first element at position 0, and shifts all the existing elements in the array to larger indices.

```
def push(v)
  out = items.unshift(v)
  return out
end
```



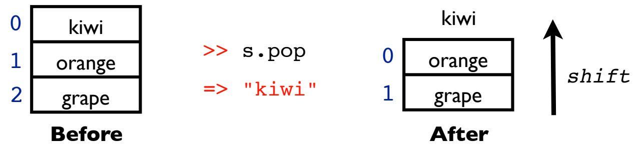
(a) Implementation of `push`

```
def peek
  out = items.first
  return out
end
```



(b) Implementation of `peek`

```
def pop
  out = items.shift
  return out
end
```



(c) Implementation of `pop`

Figure 1.4: *Array-based implementation of a stack and its operations*

- Figure 1.4(b) shows the implementation of the `peek` operation. It returns the first element of the array `items` (at position `0`), and it does not change the composition of the array.
- Figure 1.4(c) shows the implementation of the `pop` operation. It uses the operation `shift` of an array, which removes the first item at `0`, and shifts all the existing elements in the array to smaller indices.

Complexity. The `peek` has complexity of $O(1)$, because it involves simply accessing an array element by index. In the above implementation, we leverage on Ruby's `shift` and `unshift` operations for simplicity. Every `push` or `pop` operation requires shifting all elements by one position, the complexity of these operations will be $O(n)$, where n is the current count of elements.

Challenge

Can you think of alternative implementations of `push` and `pop` operations with $O(1)$ complexity?

Stack and Recursion

Earlier in Week 4 of the course, we learnt about recursion as a problem solving technique. When a recursive algorithm is compiled, it is typically converted into a non-recursive form

```
def fibonacci(n)
  if n == 0
    return 0
  elsif n == 1
    return 1
  else
    return fibonacci(n-1) + fibonacci(n-2)
  end
end
```

(a) Recursive

```
def fibonacci_stack(n)
  s = Stack.new
  s.push(n)
  result = 0
  while s.count > 0
    current = s.pop
    if current == 0
      result += 0
    elsif current == 1
      result += 1
    else
      s.push(current - 2)
      s.push(current - 1)
    end
  end
  return result
end
```

(b) Non-recursive using stack

Figure 1.5: *Algorithms to compute Fibonacci numbers*

by the compiler. To better understand how recursion is related to stacks, let us take a look at how a recursive algorithm can be re-written into an iterative algorithm using stack.

We will first see a simple example based on Fibonacci series, and later we will re-visit binary search `rbsearch` to search for an item in an array.

Fibonacci Series

Fibonacci series are the numbers in the following sequence: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$. In mathematical terms, the n-th number in the series is the sum of the $(n-1)$-th and $(n-2)$-th numbers. The first and second numbers in the series are 0 and 1.

Figure 1.5(a) shows a recursive algorithm `fibonacci(n)` to compute the `n`-th number in the Fibonacci series for `n >= 0`. For the base case `n == 0`, it returns 0. For the other base case `n == 1`, it returns 1. Otherwise, it makes two recursive calls `fibonacci(n-1)` and `fibonacci(n-2)` and adds up the outputs.

To convert a recursion into a non-recursive version, we follow several simple steps.

1. Create a new stack

2. Push initial parameters onto the stack
3. Insert a `while` loop to iterate until the stack is empty or the return condition is met.
    a) Pop the current parameter to be processed from stack
    b) If it is a base case, do the base case computation and do not make any more recursive calls.
    c) If it is a reduction step, we replace the recursive calls with appropriate `push` operations to insert the next set of parameter onto the stack. Take care to push last the parameters to be processed first.

Figure 1.5(b) shows a non-recursive algorithm `fibonacci_stack(n)`, which uses a stack. For the base cases `n == 0` and `n == 1`, it conducts the addition operations. Instead of making recursive calls, we push the next two parameters, `n-1` and `n-2`, onto the stack.

Take note that in Figure 1.5(a), we call `fibonacci(n-1)` before `fibonacci(n-2)`. In Figure 1.5(b), we push `n-2` before `n-1`. This is because to mimic the recursive call to `n-1` first, we need to push it last onto the stack.

Binary Search

Let us now revisit the recursive algorithm to search an array in Figure 5.11 (page 124) in [Conery(2011)]. The algorithm searches a sorted array `a`, in the range of positions from `lower` to `upper`, for the position where the value `k` occurs. It starts from the full range, and checks the middle position `mid`. If the key `k` is not at `mid`, it recursively calls the same operation, but with a smaller range, either from `lower` to `mid`, or from `mid` to `upper`.

Let us now see how a similar algorithm can be implemented without recursion, but using a stack instead. Figure 1.6 shows such an algorithm. It starts by pushing the initial boundary positions `lower` and `upper` onto the stack. It then operates on the top-most items of the stack until either the key `k` is found, or the stack is empty (meaning `k` cannot be found). Note that items are popped (`upper` first, then `lower`) in opposite order in which they are pushed (`lower` first, then `upper`).

Understanding how recursion can be “implemented” using stack will help us understand the sequence in which recursive calls are made. Later, in the tutorial exercises, we will experiment with how a recursive algorithm will continuously make further recursive calls until the base cases are reached (which can be simulated by pushing new parameter values onto a stack), and then sequentially return the outputs to the previous recursive calls (which can be simulated by popping “completed” parameter values from the stack).

Challenge

What are the advantages and disadvantages of using recursion versus using stack?

```
def rbsearch_stack(a, k, lower=-1, upper=a.length)
  s = Stack.new
  s.push(lower)
  s.push(upper)
  while s.count > 0
    upper = s.pop
    lower = s.pop
    mid = (lower + upper)/2
    return nil if upper == lower + 1
    return mid if k == a[mid]
    if k < a[mid]
      s.push(lower)
      s.push(mid)
    else
      s.push(mid)
      s.push(upper)
    end
  end
end
```

start by pushing the initial values into the stack

operate on the top-most values of the stack

if item is found or no more items, return

if item is not found yet, push new search boundaries into the stack

Figure 1.6: *A non-recursive algorithm for* `rbsearch` *using stack*

Tutorial Project

Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'is103'
=>  true

>>  include LinearDSLab
=>  Object
```

T30. First, test the recursive version of the `fibonacci` method.

```
>>  fibonacci(0)
=>  0
>>  fibonacci(1)
=>  1
>>  fibonacci(2)
=>  1
>>  fibonacci(3)
=>  2
>>  fibonacci(4)
=>  3
```

T31. Make sure the stack-based version also produces the same outputs.

```
>>  fibonacci_stack(0)
=>  0
>>  fibonacci_stack(1)
=>  1
>>  fibonacci_stack(2)
=>  1
>>  fibonacci_stack(3)
=>  2
>>  fibonacci_stack(4)
=>  3
```

T32. To see the sequence of recursive calls, we will insert a probe to `fibonacci(n)` method and print out each `n` being recursively called.

```
>>  Source.probe("fibonacci", 2, "puts n")
=>  true
>>  trace {fibonacci(4)}
 4
 3
 2
 1
 0
 1
 2
 1
 0
```

Trace by hand the sequence of recursive calls for `fibonacci(4)`. Is it the same as the above?

T33. Let's compare this sequence to the stack-based version. We will insert another probe to `fibonacci_stack(n)` method and print out the content of stack `s` in each iteration.

```
>>  Source.probe("fibonacci_stack", 6, "puts s")
=>  true
>>  trace {fibonacci_stack(4)}
 [4]
 [3, 2]
 [2, 1, 2]
 [1, 0, 1, 2]
 [0, 1, 2]
 [1, 2]
 [2]
 [1, 0]
 [0]
```

Do you observe that as the iteration goes on, the sequence of elements occupying the top of the stack mirrors the sequence of recursive calls in the previous question?

T34. Trace several more runs of `fibonacci` and `fibonacci_stack` for different values of `n`. Do you understand the sequence of recursive calls, and the sequence of stack operations?

T35. Make a small test array for `rbsearch` (which is sorted).

```
>>  a = TestArray.new(8).sort
=>  [6, 38, 45, 48, 55, 57, 58, 92]
```

T36. Search for a value that exists in the array, and compare the outputs of `rbsearch` and `rbsearch_stack`

```
>>  rbsearch(a, 38)
=>  1

>>  rbsearch_stack(a, 38)
=>  1
```

T37. Search for a value that does not exist in the array, and compare the outputs of `rbsearch` and `rbsearch_stack`

```
>>  rbsearch(a, 16)
=>  nil

>>  rbsearch_stack(a, 16)
=>  nil
```

T38. Attach a probe to `rbsearch` to print brackets around the region being searched at each call:

```
>>  Source.probe("rbsearch", 2, "puts brackets(a, lower, upper)")
=>  true
```

T39. Trace a call to `rbsearch`:

```
>>  trace { rbsearch(a, 92) }
  [6  38  45  48  55  57  58  92]
   6  38  45 [48  55  57  58  92]
   6  38  45  48  55 [57  58  92]
   6  38  45  48  55  57 [58  92]
=>  7
```

T40. Attach a probe to `rbsearch_stack` to print brackets around the region being searched after popping the `lower` and `upper` values from the stack:

```
>>  Source.probe("rbsearch_stack", 8, "puts brackets(a, lower, upper)")
=>  true
```

T41. Trace a call to `rbsearch_stack`:

```
>>  trace { rbsearch_stack(a, 92) }
  [6  38  45  48  55  57  58  92]
   6  38  45 [48  55  57  58  92]
   6  38  45  48  55 [57  58  92]
   6  38  45  48  55  57 [58  92]
=>  7
```

Is it similar to the trace of the recursive version of binary search `rbsearch`?

T42. Trace several more runs of `rbsearch` and `rbsearch_stack` for different input arrays. Do you understand why the sequence of operations for the recursive and the non-recursive version are similar?

1.2 Queue

In contrast to stacks which have LIFO property, queues are described as FIFO or First-In, First-Out. In other words, the next item to be accessed is the earliest item put into the queue.

Access to any element in a queue is allowed through one of the following operations:

- `enqueue` inserts a new element to the last position or the *tail* of the queue
- `peek` returns the element currently at the first position or the *head* of the queue, but does not remove it.

- `dequeue` operation returns the element currently at the head of the queue, and removes it from the queue. As a result, the element following the removed element now occupies the first position.

Tutorial Project

Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'is103'
=>  true

>>  include LinearDSLab
=>  Object
```

T43. Make a new queue object and save it in a variable named `q`:

```
>>  q = LinearDSLab::Queue.new
=>  []
```

This new queue is currently empty.

T44. Let's add a string to the new queue:

```
>>  q.enqueue("grape")
=>  ["grape"]
```

T45. We can also use << operator to enqueue new elements at the tail:

```
>>  q << "orange"
=>  ["grape", "orange"]
```

The first element in the queue is the head, which is also the least recently added. The last element in the queue is the tail, which is also the most recently added.

T46. We can visualize the current state of the queue:

```
>>  view_queue(q)
=>  true
```

Do the elements at the head and the tail respectively match your expectation?

T47. Add more strings to the queue:

```
>>  ["mango", "guava", "kiwi"].each { |x| q << x }
=>  ["mango", "guava", "kiwi"]
```

T48. Check the strings in the queue currently:

```
>>  q
=>  ["grape", "orange", "mango", "guava", "kiwi"]
```

T49. We can find out what the first string in the queue is without removing it by using `peek`:

```
>>  q.peek
=>  "grape"

>>  q
=>  ["grape", "orange", "mango", "guava", "kiwi"]
```

Note that no string has been removed from the queue. The state of the queue after `peek` is called can be visualized, as shown in Figure 1.7.

T50. Remove a string from the queue by using `dequeue` and store it in a variable named `t`:

```
>>  t = q.dequeue
=>  "grape"
```

Do you observe in the visualization how the element at the head is removed after the `dequeue` is called?

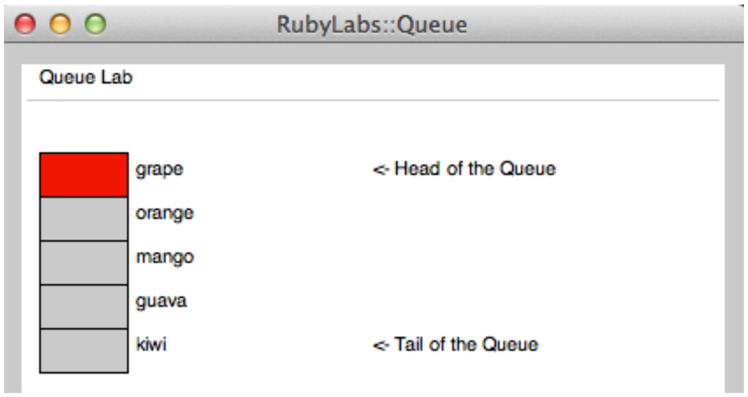


Figure 1.7: *Visualization of queue after* `peek` *is called*

T51. Check that the first string has been removed from the queue `q` and is now stored in `t`:

```
>>  q
=>  ["orange", "mango", "guava", "kiwi"]

>>  t
=>  "grape"
```

T52. A synonym for `dequeue` is the `shift` operator:

```
>>  t = q.shift
=>  "orange"
```

A Simple Application of Queue

Queues are commonly found in real life. It is used to maintain an ordering that favors those that arrive earlier over those that arrive later. In computing, queues have many applications. In a multitasking operating system, different software applications claim access to the CPU, and the operating system maintains a schedule of which application runs. Often times, a queue may be a suitable structure for this.

To more vividly appreciate the usage of queue, we now explore a simple application: *recognizing palindromes*. A palindrome is a string character that reads the same, whether read from left to right, or from right to left. There are many examples of palindromes in the English language, such as `madam`, `radar`, `refer`, etc.

How do we recognize a palindrome? Remember that a queue has the FIFO property. The first item entering a queue is also the first item leaving the queue. When we insert (`enqueue`) letters in a word from left to right into a queue, we are going to "read" from left to right as well when we access (`dequeue`) them from the queue. In contrast, a stack has the LIFO property. When we insert (`push`) letters in a word from left to right into a stack, we are going to "read" from right to left when we access (`pop`) them from the stack. Hence, if both a queue and a stack read the same sequence of letters, the word is a palindrome.

```
def is_palindrome(v)
  t = v.split(//)
  q = LinearDSLab::Queue.new
  s = Stack.new
  t.each { |x| q.enqueue(x) }
  t.each { |x| s.push(x) }
  while s.count > 0
    left = q.dequeue
    right = s.pop
    return false if left != right
  end
  return true
end
```

Figure 1.8: *An algorithm to recognize palindromes*

Tutorial Project

T53. Create a new array that contains letters in a palindrome:

```
>>  t = "pop".split(//)
=>  ["p", "o", "p"]
```

T54. Insert the array elements into a queue:

```
>>  q = LinearDSLab::Queue.new
=>  []

>>  t.each { |x| q.enqueue(x) }
=>  ["p", "o", "p"]
```

T55. Insert the array elements into a stack:

```
>>  s = Stack.new
=>  []

>>  t.each { |x| s.push(x) }
=>  ["p", "o", "p"]
```

T56. Read the left-most letter from queue, and the right-most letter from stack, and check that they are the same:

```
>>  left = q.dequeue
=>  "p"

>>  right = s.pop
=>  "p"

>>  left == right
=>  true
```

The first and last letters are the same.

T57. Repeat a couple more times till the queue and stack are empty:

```
>>  left = q.dequeue
=>  "o"

>>  right = s.pop
=>  "o"

>>  left == right
=>  true

>>  left = q.dequeue
=>  "p"

>>  right = s.pop
=>  "p"

>>  left == right
=>  true
```

All the letters are the same whether from left to right (by queue) or from right to left (by stack). The word formed by letters in the array is a palindrome.

This way of checking palindromes is used in the method `is_palindrome` in Figure 1.8. We are now going to use it to recognize whether a word is a palindrome or not.

T58. Let's verify some palindromes:

```
>>  is_palindrome("radar")
=>  true

>>  is_palindrome("madam")
=>  true
```

T59. Let's try some words that are not palindromes:

```
>>  is_palindrome("house")
=>  false

>>  is_palindrome("pot")
=>  false
```

`Array`-based Implementation of Queue in Ruby

We will now briefly explore the implementation of a queue in Ruby based on array. It is similar to the stack implementation with some key differences.

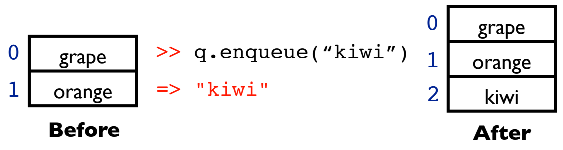
In this array-based implementation, we use an array variable called `items` to store the elements in the stack.

```
items = Array.new
```

We then define the operations in terms of array operations on this array `items`, as follows:

- Figure 1.9(a) shows the implementation of the `enqueue` operation. It uses the operation << of an array, which inserts the new element `v` as the last element. None of the other elements currently in the array is affected.

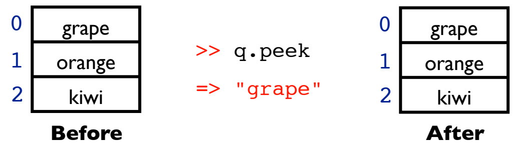
```
def enqueue(v)
  out = items << v
  return out
end
```



```
>> q.enqueue("kiwi")
=> "kiwi"
```

(a) Implementation of `enqueue`

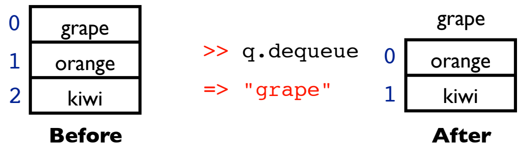
```
def peek
  out = items.first
  return out
end
```



```
>> q.peek
=> "grape"
```

(b) Implementation of `peek`

```
def dequeue
  out = items.shift
  return out
end
```



```
>> q.dequeue
=> "grape"
```

(c) Implementation of `dequeue`

Figure 1.9: *Array-based implementation of a queue and its operations*

- Figure 1.9(b) shows the implementation of the `peek` operation. It returns the first element of the array `items` (at position `0`), and it does not change the composition of the array.
- Finally, Figure 1.9(c) shows the implementation of the `dequeue` operation. It uses the operation `shift` of an array, which removes the first item at `0`, and shifts all the existing elements in the array to smaller indices.

Complexity. The `peek` operation has complexity of $O(1)$, because it involves simply accessing an array element by index. Similarly, `enqueue` involves inserting an element at the end of an array, which is also $O(1)$. In an array-based implementation of a queue, the `dequeue` operation is $O(n)$, because it shifts all the elements in the array by one position.

Challenge

Can you think of a more efficient array-based implementation of a queue, such that both `enqueue` and `dequeue` would have complexity of $O(1)$?

1.3 Priority Queue

We will now briefly cover another linear data structure called priority queue. As the name suggests, priority queue is related to queue, in that it has similar operations, namely enqueue and dequeue. There is one crucial difference. A queue maintains the elements in FIFO order, whereas a priority queue maintains the elements in some *priority order*.

In other words, a dequeue operation will remove an element with higher priority before another element with lower priority (even if the low priority element is inserted earlier). In case of two elements with the same priority, a tie-breaking mechanism is employed, such as which element is inserted earlier in the queue.

The definition of this priority order may vary in different applications. One application is maintaining the waiting list in a hospital's emergency department. When a patient comes, she is assessed by a nurse or a doctor, and enters the queue. However, the placement of a patient in the queue is not necessarily based on first come, first served. A patient requiring more urgent medical care will be placed ahead of another patient who arrives earlier with relatively minor ailment. For another example, in a theme park, a customer entering the queue of an attraction may be placed ahead in the queue if she has an Express Pass.

RubyLabs has an implementation of priority queue in the `BitLab` found in Chapter 7 of [Conery(2011)]. In the RubyLabs implementation, enqueue uses the << operator, and dequeue uses the `shift` operator. For simplicity, for our tutorial exercises in this handout, we will assume that a queue can be either numbers or strings. For numbers, smaller numbers have priority over larger numbers. For strings or characters, the priority is in alphabetical order. We will revisit priority queue when we cover Chapter 7 of [Conery(2011)].

Alternative Implementations of a Priority Queue

Because a priority queue always maintains a sorted order of its elements, there could be several different approaches to implement a priority queue.

Unsorted Array The first approach is to keep the elements in an unsorted array. Each enqueue operation will be $O(1)$, because a new element simply enters the last position in the array. Each peek can also be $O(1)$, if we keep a variable to store the minimum value at all times. However, each dequeue operation will be $O(n)$, because it may require traversing the whole array to find the element with minimum value (highest priority), and shift the rest of the elements.

Sorted Array The second approach is to maintain a sorted array at all times. In this case, each enqueue operation will be $O(n)$, because in the worst case, we need to iterate over all the positions in the array to find a suitable position for a new element, so as to maintain the sorted order. Each peek can be $O(1)$, because in a sorted list, we can find out the minimum value very quickly. Each dequeue operation is $O(n)$ if we store the elements in the array in sorted order, i.e., smallest element in first position, because after the smallest element is removed, we have to shift all the remaining elements. A more efficient implementation is to store the elements in reverse sorted order, so removing the smallest element will not affect the remaining ones. In this case, each dequeue operation can be $O(1)$.

Binary Search Tree The third approach is to maintain a data structure called binary search tree (which we will discuss in Chapter 2 of this handout). In this case, any enqueue or dequeue operation will be $O(log\ n)$ in the *average case*, similar complexity to binary search. The worst case complexity of binary search tree is $O(n)$.

	Unsorted Array	*Sorted Array*	*Binary Search Tree (average case)*
`enqueue`	$O(1)$	$O(n)$	$O(log\ n)$
`peek`	$O(1)$	$O(1)$	$O(1)$
`dequeue`	$O(n)$	$O(1)$	$O(log\ n)$

Table 1.1: *Complexity of Different Implementations of a Priority Queue*

Tutorial Project

This tutorial project makes use of the BitLab module of RubyLabs. Start a new IRB session and load the BitLab module:

```
>>  include BitLab
=>  Object
```

T60. Create a new priority queue object:

```
>>  pq = PriorityQueue.new
=>  []
```

T61. Let's add a couple of strings to the priority queue:

```
>>  pq << "watermelon"
=>  ["watermelon"]

>>  pq << "apple"
=>  ["apple", "watermelon"]

>>  pq << "orange"
=>  ["apple", "orange", "watermelon"]
```

Do you notice that the contents of the priority queue are always sorted? As mentioned above, the order of priority here is based on alphabetical order.

T62. Try removing an element from the priority queue:

```
>>  t = pq.shift
=>  "apple"
```

The element removed is the one added second, but is ahead of other elements in the queue due to alphabetical sorting.

T63. Let us see how a priority queue maintains different numbers:

```
>>  pq = PriorityQueue.new
=>  []

>>  pq << 1.02
=>  [1.02]

>>  pq << 1.03
=>  [1.02, 1.03]

>>  pq << 1.01
=>  [1.01, 1.02, 1.03]

>>  pq << 1
=>  [1, 1.01, 1.02, 1.03]

>>  pq << 2
=>  [1, 1.01, 1.02, 1.03, 2]
```

The smallest number is always at the head of the queue.

1.4 Summary

In this chapter, we learn about linear data structures. We discuss primarily two types of data structures: stacks and queues, and briefly touch on a third one: priority queues.

Their differences are in how items are inserted and removed from the data structure.

- A stack has LIFO or Last-In, First-Out property. It uses `push` operation to insert an element onto the top of the stack, and uses `pop` operations to remove the element on the top of the stack. `peek` returns the top-most element without removing it.
- A queue has FIFO or First-In, First-Out property. It uses `enqueue` operation to insert an element into the tail of the queue, and uses `dequeue` operations to remove the head of the queue. `peek` returns the first element without removing it.
- A priority queue has similar operations as a queue, but it maintains the elements in a priority order. In our exercises here, unless otherwise stated, we assume sorted ordering. In practical applications, a different definition of priority may be introduced.

Advanced Material (Optional)

For those interested in implementations in Java, please refer to [Prichard and Carrano(2011)], stacks in Ch 7.3, queues in Ch 8.3, and priority queue in Ch 12.2.