School of Information Systems

# IS103 Computational Thinking

# Handout on Fundamental Data Structures

**INFORMATION FOR CANDIDATES**

1. **Scope:** The handout supplements the reference materials for students enrolled in IS103. It covers topics on **fundamental data structures**, which is within the scope of the course, but is not covered in the primary textbook (*Explorations in Computing by John S. Conery*).

2. **Authorship**: This handout is authored by Dr Hady Wirawan LAUW, who is thankful to course co-instructors Mr MOK Heng Ngee and Dr LAU Hoong Chuin for their helpful suggestions and feedback. Teaching Assistants, Mr LAI Xin Chu and Ms Yosin ANGGUSTI wrote the Ruby codes used in the tutorial exercises, as well as the visualization modules.

3. **Bibliography**: In preparing this material, we occasionally refer to the contents in the primary and supplementary texts listed in the Bibliography. Where we do so, we indicate it in the handout. The formatting of this handout is deliberately similar to that of the primary textbook (with its author's permission) for consistency.

4. **Copyright Notice**: This handout is provided for free to students and instructors, provided proper acknowledgement is given to the source of the materials (the author, the school, and the university). All rights reserved.

© 2012 Hady W. Lauw

# Contents

**1 So the Last shall be First, and the First Last** **2**
*Linear data structures: stack, queue, and priority queue*
1.1 Stack . . . 3
1.2 Queue . . . 14
1.3 Priority Queue . . . 19
1.4 Summary . . . 22

**2 Good Things Come in Pairs** **24**
*Modeling hierarchical relationships with binary trees*
2.1 Tree . . . 24
2.2 Binary Tree . . . 25
2.3 Traversals of a Binary Tree . . . 29
2.4 Binary Search Tree . . . 32
2.5 Summary . . . 37

**3 It's A Small World After All** **40**
*Modeling networks with graphs*
3.1 Real-world Networks . . . 40
3.2 Graph . . . 42
3.3 Traversals of a Graph . . . 48
3.4 Topological Sorting . . . 53
3.5 Shortest Path . . . 58
3.6 Summary . . . 59

# 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

The tutorial exercises are provided in a file ‘lineardslab.rb’ that you can download from eLearn. Ensure that the file ‘lineardslab.rb’ is in the same directory as when you start the IRB session. Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'lineardslab.rb'
=>  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

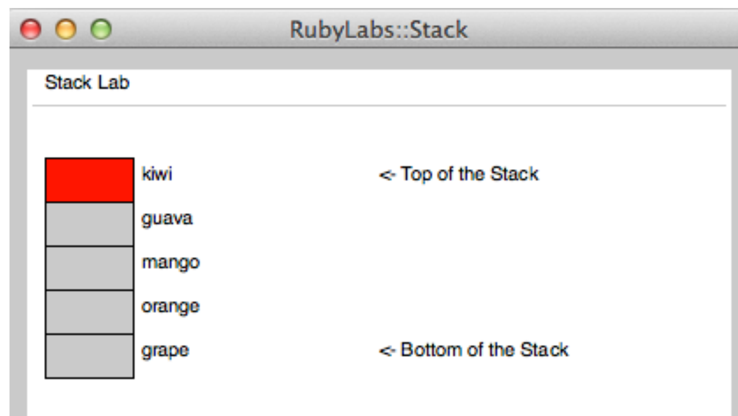



**Figure 1.1:** *Visualization of stack after* `peek` *is called*

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.

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

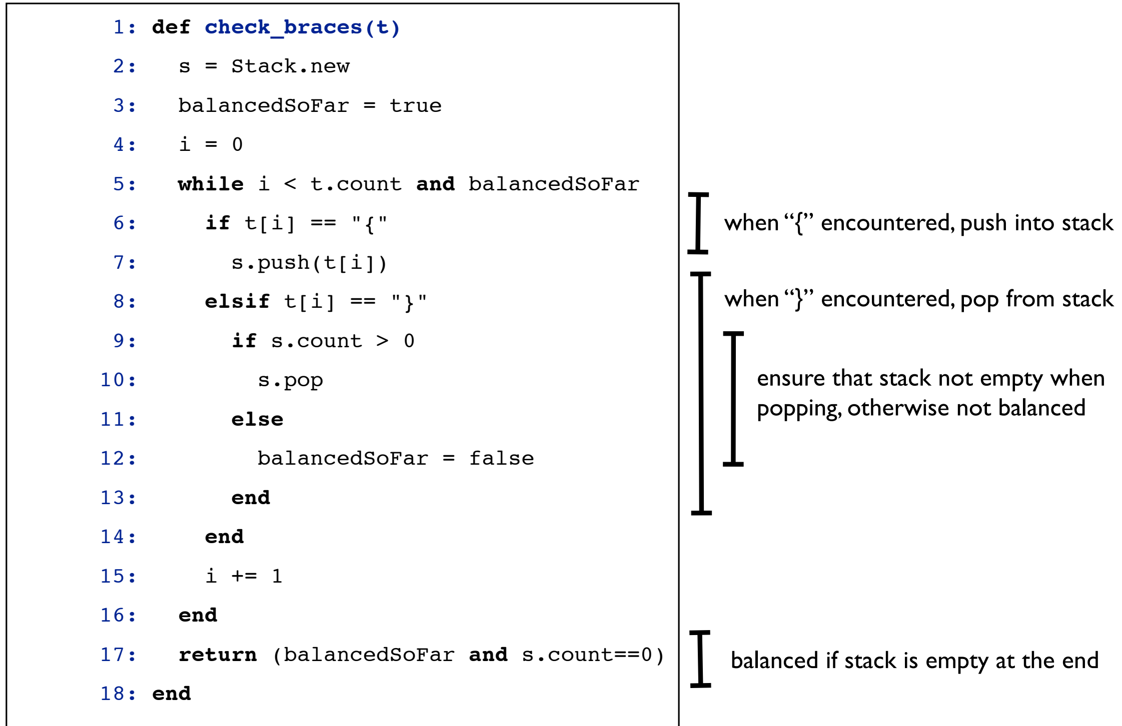



**Figure 1.2:** *An algorithm to check balanced braces*

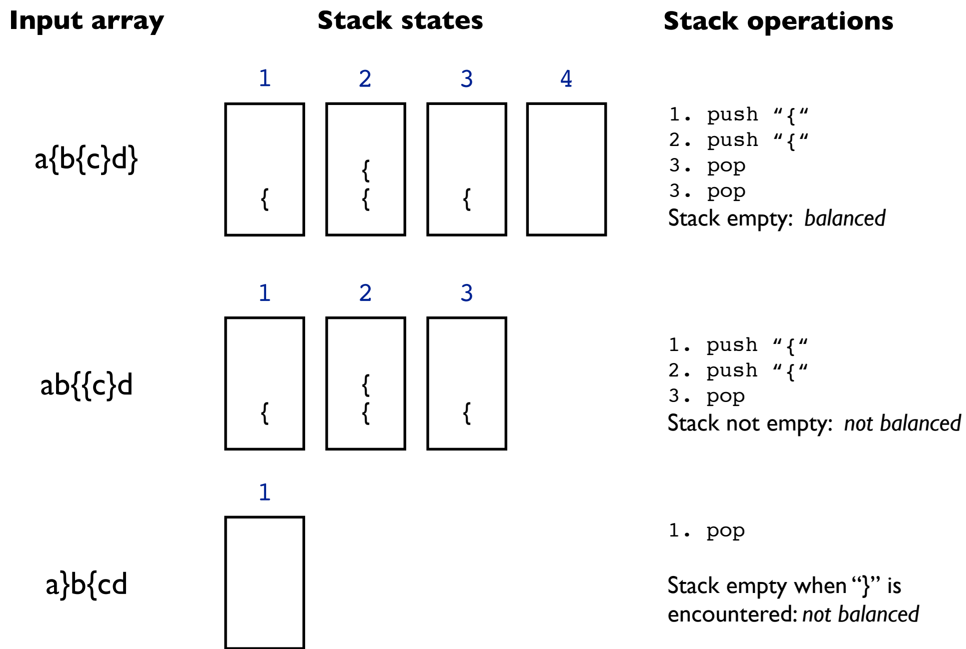



**Figure 1.3:** *Traces of the algorithm to check balanced braces*

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.

## 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:

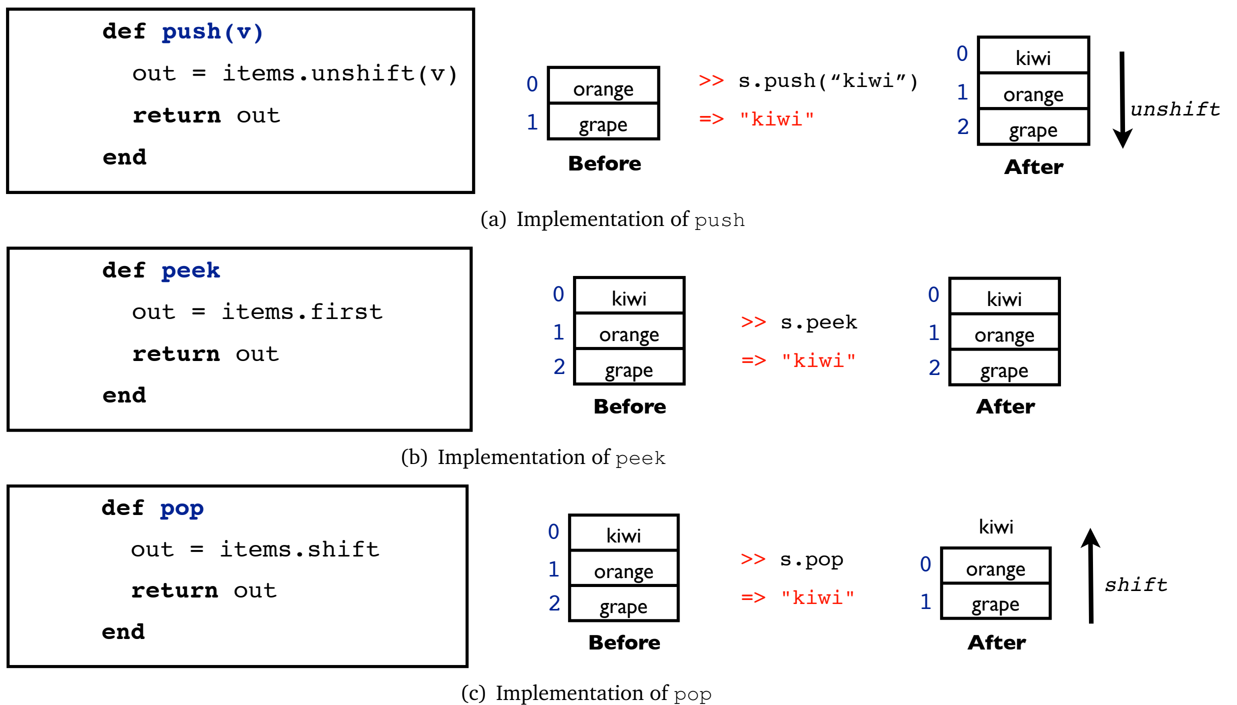



**Figure 1.4:** *Array-based implementation of a stack and its operations*

- 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.
- 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?

```
1: def fibonacci(n)
2:   if n == 0
3:     return 0
4:   elsif n == 1
5:     return 1
6:   else
7:     return fibonacci(n-1) + fibonacci(n-2)
8:   end
9: end
```

(a) Recursive

```
 1:  def fibonacci_stack(n)
 2:    s = Stack.new
 3:    s.push(n)
 4:    result = 0
 5:    while s.count > 0
 6:      current = s.pop
 7:      if current == 0
 8:        result += 0
 9:      elsif current == 1
10:        result += 1
11:      else
12:        s.push(current - 2)
13:        s.push(current - 1)
14:      end
15:    end
16:    return result
17:  end
```

(b) Non-recursive using stack

**Figure 1.5:** *Algorithms to compute Fibonacci numbers*

## 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 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?

```
 1: def rbsearch_stack(a, k, lower=-1, upper=a.length)
 2:   s = Stack.new
 3:   s.push(lower)
 4:   s.push(upper)
 5:   while s.count > 0
 6:     upper = s.pop
 7:     lower = s.pop
 8:     mid = (lower + upper)/2
 9:     return nil if upper == lower + 1
10:     return mid if k == a[mid]
11:     if k < a[mid]
12:       s.push(lower)
13:       s.push(mid)
14:     else
15:       s.push(mid)
16:       s.push(upper)
17:     end
18:   end
19: 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

The tutorial exercises are provided in a file 'lineardslab.rb' that you can download from eLearn. Ensure that the file 'lineardslab.rb' is in the same directory as when you start the IRB session. Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'lineardslab.rb'
=>  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

The tutorial exercises are provided in a file 'lineardslab.rb' that you can download from eLearn. Ensure that the file 'lineardslab.rb' is in the same directory as when you start the IRB session. Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'lineardslab.rb'
=>  true

>>  include LinearDSLab
=>  Object
```

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

```
>>  q = 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"
```

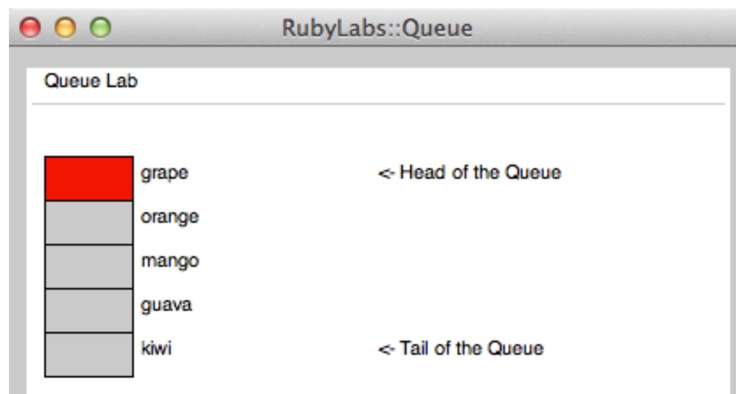



**Figure 1.7:** *Visualization of queue after* `peek` *is called*

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

```
 1: def is_palindrome(v)
 2:   t = v.split(//)
 3:   q = Queue.new
 4:   s = Stack.new
 5:   t.each { |x| q.enqueue(x) }
 6:   t.each { |x| s.push(x) }
 7:   while s.count > 0
 8:     left = q.dequeue
 9:     right = s.pop
10:     return false if left != right
11:   end
12:   return true
13: 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 = 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.

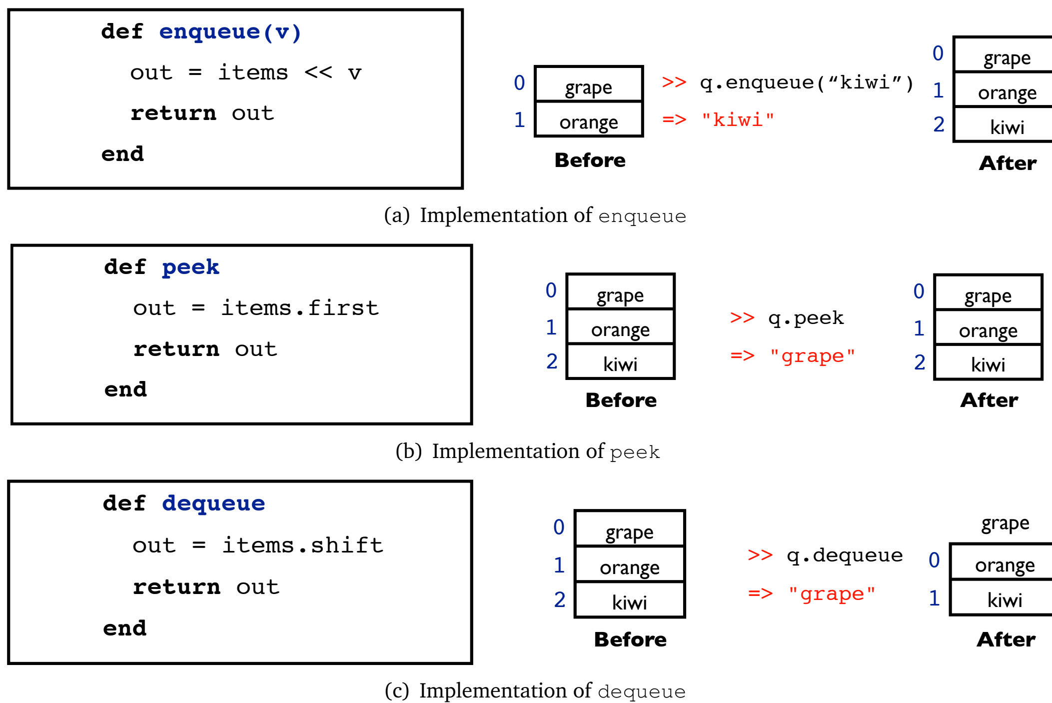



**Figure 1.9:** *Array-based implementation of a queue and its operations*

- 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 en-

queue 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.

### Exercises

1. Which data structure (stack, queue, or priority queue) will you use for each of the following applications, and why?
   a) managing customer calls in a helpdesk office
   b) implementing the undo feature on a software application
   c) managing waiting list for matching organs for transplant
2. Give 3 examples of real life applications of stacks and queues.
3. What is the output of calling `s.pop` after the following operations?
```
>> s = Stack.new
>> s.push(1)
>> s.push(3)
>> s.pop
>> s.push(4)
>> s.pop
>> s.push(2)
```
4. What is the output calling `q.shift` after the following operations?
```
>> q = Queue.new
>> q << 1
>> q << 3
>> q.shift
>> q << 4
>> q.shift
>> q << 2
```

5. What would be the answer to the above question if q had been a priority queue, with smaller numbers given higher priority?

6. Suppose we run the following commands on irb:

```
>>  t = ["{", "1", "+", "11", "}", "*", "{", "21", "}"]
>>  check_braces(t)
```

   a) What is the content of the stack when we get to "11"?

   b) What is the content of the stack when we get to "21"?

   c) What is the return value of check_braces(t)?

7. The following is a recursive implementation of the factorial function. Convert it into a non-recursive implementation using stack.

```
def factorial(n)
  return 1 if n == 1
  return n * factorial(n-1)
end
```

8. The following is a recursive implementation of the Dijkstra's algorithm to find the greatest common divisor of two numbers a and b. Convert it into a non-recursive implementation using stack.

```
def dijkstra(a, b)
  return a if a == b
  if a > b
    return dijkstra(a -  b, b)
  else
    return dijkstra(a, b - a)
  end
end
```

9. The recursive algorithm for qsort quicksort is presented in Figure 5.11 (page 124) in [Conery(2011)]. Convert this into a non-recursive algorithm using stack. You may assume that the same partition method is available.

10. Instead of using a queue and a stack, design alternative implementations of is_palindrome in Figure 1.8, which use:

    a) For the first modified implementation, use two queues.

    b) For the second modified implementation, use one stack only.

11. Implement a stack such that push and pop will only require $O(1)$ complexity.

12. Implement the enqueue, peek, and dequeue operations of a priority queue, using the following underlying data structure:

    a) for the first implementation, use an unsorted Array object

    b) for the second implementation, use an Array object which is kept sorted

# 2

# Good Things Come in Pairs

### *Modeling hierarchical relationships with binary trees*

All the previous data structures that we have explored so far have been *linearly* organized. They are good at keeping a set of elements that we need to access in a certain order. Each element is accessed by its position, either in terms of an index within the data structure, or in terms of its relative position in the top/bottom of a stack, or the head/tail of a queue.

In some applications what we need is a data structure that organizes elements based on their relationship to one another. For example, if we want to represent an organizational chart, we need to keep the information of who reports to whom. To represent a family tree, we need to know who are the parents of whom. To represent an inter-city road network, we need to record which city can be reached from which other city.

A data structure that can model such relationships are called *non-linear* data structures. In this course, we will cover two such structures. In this chapter, we will model *hierarchical* relationships with *trees*, and more specifically *binary trees*. In the next chapter (Chapter 3), we will model non-hierarchical relationships using graphs.

## 2.1 Tree

Trees are used to represent hierachical relationships. Figure 2.1 shows an organizational chart of a company, which can be modeled as a tree. Each element in a tree is called a **node** (e.g., CEO, VP Operations).

Nodes in a tree are related to one another. A node $n_1$ (e.g., CEO) is the **parent** of another node $n_2$ (e.g., VP Operations), if there is an edge between the two nodes, and $n_1$ is above $n_2$ in the tree. In this case, $n_2$ is a **child** of $n_1$. Every node can have zero or more children, but at most one parent. Nodes with the same parent are **siblings** of one another.

If we extend this relationship through multiple hops, we say a node $n_1$ is an **ancestor** of another node $n_3$ (or equivalently $n_3$ is a **descendant** of $n_1$), if there exists an unbroken path from $n_1$, to a child of $n_1$, to a child of this child, and so on, and finally to $n_3$. For example, in Figure 2.1, Manager (HR) is a descendant of CEO.

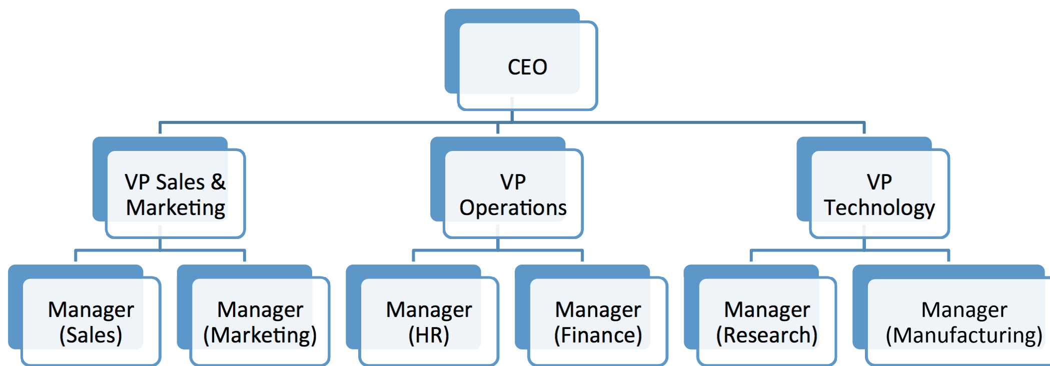



**Figure 2.1:** *Example of Tree Structure: Organization Chart*

A **tree** consists of a node and all its descendant. The ultimate ancestor node is called the **root**. Nodes without children, at the lowest levels of the tree, are called **leaf** nodes. Note the irony, and contrast to real-life trees, of having a tree with roots at the top, and leaves at the bottom. Figure 2.1 is a tree with CEO as the root node. The managers occupy the leaf nodes of this tree.

The root node is at **level** 1 of the tree. Its children are at level 2, and so on. The **height** of a tree is the maximum level of any leaf node. In Figure 2.1, the height of the tree is 3.

A tree can have many **subtrees**, where again each subtree consists of a node and all its descendants in the original tree. For instance, in Figure 2.1, each VP is the root of its own subtree.

Because each child of the root node is itself the root of its own subtree, we can have the following **recursive definition of a tree**. A tree consists of a root node $n$, and a number of subtrees rooted at the children of the root node $n$.

## 2.2 Binary Tree

In a general tree, a parent can have any number of children. In this section, we focus on a specific type of tree, called *binary tree*, where each parent can have at most two children, which we refer to as the **left child** and the **right child**. Hence, a binary tree is defined recursively as consisting of a root node, a left binary subtree and a right binary subtree.

Figure 2.2 illustrates an example of a binary tree. Mrs. Goodpairs, the CEO, believes that any officer of the company should not have more than two direct reports. She herself has two vice presidents, VP1 and VP2, reporting to her. Each vice president can have up to two managers. In this case, M1 and M2 report to VP1, and M3 reports to VP2.

A binary tree is said to be **full** if all the nodes, except the leaf nodes, have two children each, and all the leaf nodes are at the same level. Figure 2.2 is not a full binary tree.

A binary tree is said to be **complete** if all its nodes at level `h-2` and above have two children each. In addition, when a node at level `h-1` has children, it implies that all the nodes at the same level to its left have two children each. When a node at level `h-1` has only one child, it is a left child. Figure 2.2 is a complete binary tree.

A binary tree supports the following operations:

- `new` creates an empty binary tree with no root

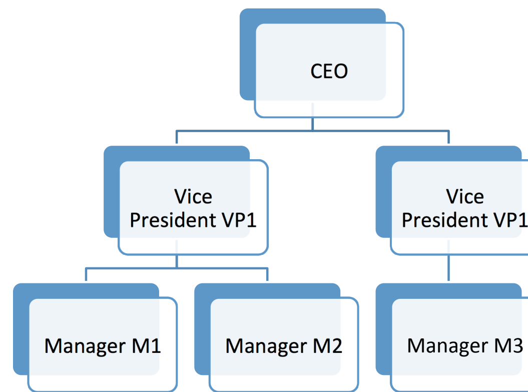



**Figure 2.2:** *Example of Binary Tree*

- `setRoot(node)` sets a node passed as input to be the root of the tree
- `root` returns the node, which is the root of the tree
- `noOfNodes` returns the number of nodes in the tree
- `heightOfTree` returns the height of the tree
- `isEmpty?` checks whether the tree has any node

Importantly, each node `n` in the tree also supports specific operations:

- `new(value)` creates a new node with a specific value passed in as argument
- `left` returns the left child
- `right` returns the right child
- `setLeft(node)` sets the node passed as input to be the left child
- `setRight(node)` sets the node passed as input to be the right child
- `delLeft` removes the current left child
- `delRight` removes the current right child

Let us now try our hands on a tutorial project on IRB, which will get us familiar with these operations of a binary tree.

## Tutorial Project

The tutorial exercises are based on Ruby code provided in a file 'treelab.rb' that you can download from eLearn. Ensure that the file 'treelab.rb' is in the same directory as when you start the IRB session. Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'treelab.rb'
=>  true

>>  include TreeLab
=>  Object
```

T1. Make a new node `r` that we will use as the root of our first binary tree:

```
>>  r = Node.new("CEO")
=>  Value: CEO, left child: nil, right child: nil
```

Note that when we create the node `r`, we give it a value "CEO". The output prints out this value. Currently, the left child and the right child are `nil`, because we have not set them yet.

T2. Let's provide `r` with a couple of children nodes `n1` and `n2`, which we will set as the left and right children respectively.

```
>>  n1 = Node.new("VP1")
=>  Value: VP1, left child: nil, right child: nil

>>  r.setLeft(n1)
=>  Value: CEO, left child: VP1, right child: nil

>>  n2 = Node.new("VP2")
=>  Value: VP2, left child: nil, right child: nil

>>  r.setRight(n2)
=>  Value: CEO, left child: VP1, right child: VP2
```

T3. Let us check that the node `r` now has a left child and a right child.

```
>>  r.left
=>  Value: VP1, left child: nil, right child: nil

>>  r.right
=>  Value: VP2, left child: nil, right child: nil
```

T4. We now create a new binary tree `t`, and set `r` as the root node of this tree.

```
>>  t = BinaryTree.new

>>  t.setRoot(r)
=>  Value: CEO, left child: VP1, right child: VP2
```

T5. Let's double check that the root has indeed been set to `r`.

```
>>  t.root
=>  Value: CEO, left child: VP1, right child: VP2
```

T6. To make it easier to understand the current structure of the tree, we can visualize it.

```
>>  view_tree(t)
=>  true
```

T7. Currently, the nodes `n1` and `n2` do not have any child. Let's practice adding a few more nodes into the tree.

```
>>  n3 = Node.new("M1")
=>  Value: M1, left child: nil, right child: nil

>>  n1.setLeft(n3)
=>  Value: VP1, left child: M1, right child: nil

>>  n4 = Node.new("M2")
=>  Value: M2, left child: nil, right child: nil

>>  n1.setRight(n4)
=>  Value: VP1, left child: M1, right child: M2
```

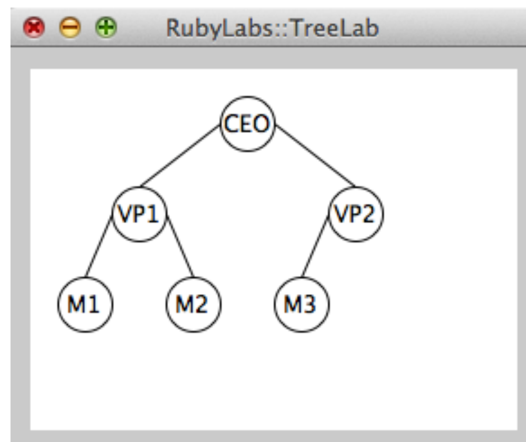



**Figure 2.3:** *Visualization of tree after the first tutorial project*

```
>>  n5 = Node.new("M3")
=>  Value: M3, left child: nil, right child: nil

>>  n2.setLeft(n5)
=>  Value: VP2, left child: M3, right child: nil

>>  n6 = Node.new("M4")
=>  Value: M4, left child: nil, right child: nil

>>  n2.setRight(n6)
=>  Value: VP2, left child: M3, right child: M4
```

Look at the tree visualization. Is the structure of the tree as expected?

T8. Sometimes, we may need to remove a node from the tree. For example, the company may be downsized, and the manager "M4" may be let go.

```
>>  n2.delRight
=>  true
```

Note that to remove a node, we simply call `delLeft` or `delRight` method from the parent of the node to be removed. In this case, "M4" is node `n6`, which is the right child of node `n2`.

T9. Do you see how the visualization changes to indicate that "M4" is no longer part of the tree? The visualization of the tree up to this step is shown in Figure 2.3. Does yours look the same?

T10. We can also find out the height of this tree, as well as the number of nodes.

```
>>  t.heightOfTree
=>  3

>>  t.noOfNodes
=>  6
```

T11. Try a few more tree operations on your own.

## 2.3 Traversals of a Binary Tree

Suppose that we would like to examine each element in the data structure exactly once. This is useful for many applications, such as searching for an element of a specific value. This process is called *traversing* a data structure.

For linear data structures, there is usually only one way, which is to iterate through the index positions of all the elements. However, in a non-linear data structure, we traverse elements by their relationships, and therefore there could be more than one traversal method.

During traversal, each node is visited exactly once. The node is "processed" during the visit. For instance, during a search, visiting a node may mean checking for its value.

For binary trees in particular, there are three common traversal methods: *preorder*, *inorder*, *postorder*, which we will explore in the following.

### preorder

In preorder traversal, we visit a node first, followed by visiting its left child, and then its right child.

An implementation of preorder traversal is shown in Figure 2.4(a). It is expressed in terms of a recursive algorithm. The root node is visited *before* the recursive calls to the left and right subtrees.

For an example of preorder traversal, we will traverse the same tree in Figure 2.3 that we created earlier in the previous tutorial project. We call `preorder_traverse` on the root node, which is "CEO". This node is visited first (no. 1). Next, we will visit the left subtree, rooted at "VP1" (no. 2). Recursively, we will visit the left subtree of "VP1", rooted at "M1" (no. 3), followed by the right subtree of "VP1", rooted at "M2" (no. 4). Since all the nodes in the left subtree of "CEO" have been visited, we now turn to the right subtree rooted at "VP2" (no. 5), which recursively calls its own left subtree rooted at "M3" (no. 6).

By now we have visited all the nodes in the following order: "CEO", "VP1", "M1", "M2", "VP2", "M3". The order is indicated in Figure 2.4(a) as sequence numbers below each node.

### inorder

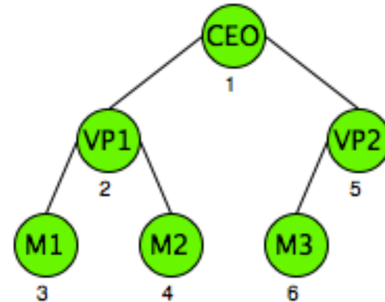
In inorder traversal, we visit the root's left subtree first, then the root, and finally its right subtree.

An implementation of inorder traversal is shown in Figure 2.4(b). It is also expressed in terms of a recursive algorithm. Note that the `visit` is called *in between* the recursive calls to the left subtree and the right subtree.

For example, we will traverse the same tree in Figure 2.3. We call `inorder_traverse` on the root node, which is "CEO". However, we do not visit this root node yet, and instead recursively traverse its left subtree rooted at "VP1". Again recursively, we traverse the left subtree of "VP1" rooted at "M1". Since "M1" has no child, there is no further recursive call, and "M1" is visited (no. 1). Having visited "M1", "VP1" is then visited (no. 2), followed by its right child "M2" (no. 3). The left subtree of "CEO" has been visited, so now we visit "CEO" itself (no. 4). This is then followed by visits to "M3" (no. 5) and "VP2" (no. 6).

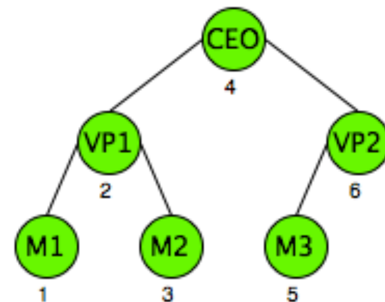
The inorder traversal sequence is "M1", "VP1", "M2", "CEO", "M3", "VP2". The order is indicated in Figure 2.4(b) as sequence numbers below each node.

```
def preorder_traverse(node)
  if (node != nil)
    visit(node)
    preorder_traverse(node.left)
    preorder_traverse(node.right)
  end
end
```



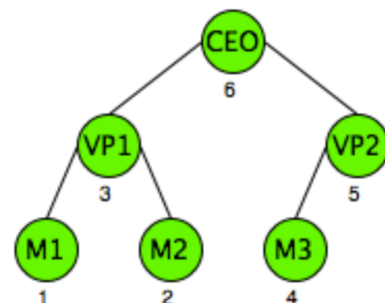

(a) preorder traversal

```
def inorder_traverse(node)
  if (node != nil)
    inorder_traverse(node.left)
    visit(node)
    inorder_traverse(node.right)
  end
end
```




(b) inorder traversal

```
def postorder_traverse(node)
  if (node != nil)
    postorder_traverse(node.left)
    postorder_traverse(node.right)
    visit(node)
  end
end
```




(c) postorder traversal

**Figure 2.4:** *Various Ways of Traversing a Binary Tree*

### postorder

In postorder traversal, we visit the root after visiting its left and right subtrees.

An implementation of postorder traversal is shown in Figure 2.4(c) It is also expressed in terms of a recursive algorithm. Note that the `visit` is called *after* the recursive calls to the left subtree and the right subtree.

We will again traverse the tree in Figure 2.3. We call `postorder_traverse` on the root node, which is "CEO". Immediately, we do a recursive call to its left subtree rooted at "VP1", followed by another recursive call to subtree rooted at "M1". Since "M1" has no child, there is no recursive call to its children. "M1" becomes the first node visited (no. 1). This is followed by "M2" (no. 2), and then "VP1" (no. 3) after its two children have been visited. We now go to the right subtree of "CEO", visiting "M3" (no. 4), "VP2" (no. 5), and finally "CEO" (no. 6).

The postorder traversal sequence is "M1", "M2", "VP1", "M3", "VP2", "CEO". The order is indicated in Figure 2.4(c) as sequence numbers below each node.

## Tutorial Project

The tutorial exercises are based on Ruby code provided in a file 'treelab.rb' that you can download from eLearn. Ensure that the file 'treelab.rb' is in the same directory as when you start the IRB session. Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'treelab.rb'
=>  true

>>  include TreeLab
=>  Object
```

To do traversal, first we need to create a binary tree. If you still have the `irb` session for the previous tutorial project, we will use that. Otherwise, you may want to re-do the first tutorial project in this chapter to create a tree first.

T12. Traverse the tree `t` using preorder:

```
>>  traversal(t, :preorder)
=>  Visited: CEO, VP1, M1, M2, VP2, M3
```

Do you see the animation in which the nodes are visited?

T13. If the animation is too fast for you, you can specify the time steps in seconds. Suppose the desired time step is 2 seconds, we call the method as follows.

```
>>  traversal(t, :preorder, 2)
=>  Visited: CEO, VP1, M1, M2, VP2, M3
```

The result is the same, but the animation goes slower now, doesn't it?

T14. Traverse the tree `t` using inorder:

```
>>  traversal(t, :inorder)
=>  Visited: M1, VP1, M2, CEO, M3, VP2
```

T15. Traverse the tree `t` using postorder:

```
>>  traversal(t, :postorder)
=>  Visited:  M1, M2, VP1, M3, VP2, CEO
```

T16. Let's create a new tree, this time using numbers as values of nodes.

```
>>  t2 = BinaryTree.new
>>  m1 = Node.new(29)
>>  t2.setRoot(m1)
>>  m2 = Node.new(14)
>>  m1.setLeft(m2)
>>  m3 = Node.new(2)
>>  m2.setLeft(m3)
>>  m4 = Node.new(24)
>>  m2.setRight(m4)
>>  m5 = Node.new(27)
>>  m4.setRight(m5)
>>  m6 = Node.new(50)
>>  m1.setRight(m6)
>>  m7 = Node.new(71)
>>  m6.setRight(m7)
>>  view_tree(t2)
```

Do you get a tree that looks like Figure 2.5?

T17. What will be the sequence of preorder traversal on this tree `t2`? Work it out by hand first before calling the Ruby method to confirm your answer.

T18. What will be the sequence of inorder traversal on this tree `t2`? Work it out by hand first before calling the Ruby method to confirm your answer.

T19. What will be the sequence of postorder traversal on this tree `t2`? Work it out by hand first before calling the Ruby method to confirm your answer.

T20. Let us attach a probe to count the number of nodes visited in any traversal. Since all the traversal methods use the `visit` method, we will attach a counting probe there.

```
>>  Source.probe("visit", 2, :count)
=>  true
```

T21. Let us now count how many nodes are visited by each traversal method.

```
>>  count { traversal(t2, :preorder) }
=>  7

>>  count { traversal(t2, :inorder) }
=>  7

>>  count { traversal(t2, :postorder) }
=>  7
```

All the traversal methods visit 7 nodes, which is the total number of nodes in the binary tree `t2`. This is because traversal means iterating through all the nodes, just in different orders.

T22. Create your own binary trees, and test your understanding of the various orders of traversal.

## 2.4 Binary Search Tree

Suppose that we are searching for a value in a binary tree. We can use any traversal order introduced in the previous section. In the worst case, we have to traverse all the nodes before we find the node we are looking for, or before we realize no such node exists. The complexity of this approach would be $O(n)$, for a binary tree of $n$ nodes.

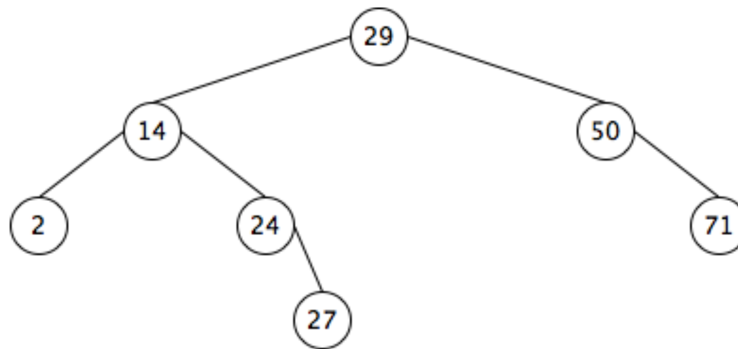



**Figure 2.5:** *Binary Search Tree*

One lesson that we keep encountering is that search could be much more efficient if the data is organized. For instance, the difference between linear search, with complexity $O(n)$, and binary search with complexity $O(log\ n)$, is that binary search assumes that the underlying array is sorted. Can we apply a similar paradigm of searching an array to searching a binary tree?

The answer is yes, if we keep the nodes in a binary tree in a "sorted" order. What does this mean? See the binary tree in Figure 2.5. Note that for all the nodes in the left subtree of the root node have smaller values than the root node. The nodes in the right subtree have larger values than the root node. Recursively, this applies for any subtree of this binary tree. Searching such a binary tree would be more efficient, because at any branch, we only need to traverse **either** the left subtree **or** the right subtree, but not both.

A **binary search tree** is a special binary tree, where for each node $n$ in the tree, its value is greater than all nodes in its left subtree, and is smaller than all nodes in its right subtree. All subtrees of this tree are themselves also binary search trees.

A node in the tree may be associated with several different attributes. For example, if a node represents a person, we may wish to store the name, as well as other attributes such as profession, date of birth, etc. In this case, we designate one of these attributes to be the *search key*, and this search key is unique.

### Searching a Binary Search Tree

In Figure 2.6, we show a recursive algorithm to search a binary tree rooted at `root` for the value `key`. This implementation makes use of preorder traversal. First, we check whether the root node has the value we are searching for. If not found and there is a left subtree, it recursively searches the left subtree first, followed by searching the right subtree. If the key is not found at all, it returns `nil`.

Note that although the order is similar to preorder traversal, it is not necessary to visit all the nodes, which would have been an $O(n)$ operation. `searchbst` finishes as soon as the node with the correct value is found. The method visits at most one node at each level, which means that the complexity of `searchbst` is $O(h)$, where $h$ is the height of the binary search tree. If the binary search tree is full, and $n$ is the number of nodes, then we have $h = log_2\ (n+1)$, which makes the complexity similar to binary search over an array.

```
def searchbst(root, key)
  if root.value == key
    return root
  elsif root.left != nil and root.value > key
    return searchbst(root.left, key)
  elsif root.right != nil and root.value < key
    return searchbst(root.right, key)
  else
    return nil
  end
end
```

**Figure 2.6:** *A Recursive Method to Search a Binary Search Tree*

```
def insertbst(root, key)
  if root.value == key
    return root
  elsif root.value > key
    if root.left != nil
      return insertbst(root.left, key)
    else
      node = Node.new(key)
      root.setLeft(node)
      return node
    end
  else
    if root.right != nil
      return insertbst(root.right, key)
    else
      node = Node.new(key)
      root.setRight(node)
      return node
    end
  end
end
```

**Figure 2.7:** *A Recursive Method to Insert a new Value to a Binary Search Tree*

### Inserting a New Node to a Binary Search Tree

Because a binary search tree keeps its nodes in a certain order, we have to be careful when inserting a node so that that order is preserved.

Suppose we want to insert new node with value 34 to the binary search tree in Figure 2.5. We first need to find the right parent for this new node. From visual inspection, we can figure out that 34 should be the left child of 50, because it is larger than 29, but smaller than 50.

Algorithmically, how do we find the "correct" position to insert the new node? Note that once the new node has been inserted, calling `searchbst` to search for this node will lead us to this "correct" position. Therefore, we should place the new node in a position where `searchbst` expects to find it in the first place. That would be where `searchbst` would have failed to find the node before it was inserted.

Figure 2.7 shows a recursive algorithm to insert a new node with value `key` to a binary search tree rooted at `root`. Note the similarity in structure to `searchbst`.

- It first inspects whether the current node already has the value `key`. If yes, no new node is inserted, and the current node is returned.
- If the `key` is less than the current node's value, we inspect the left subtree. If there is a left subtree, it recursively tries to insert the new value to the left subtree. If there is no left subtree, a new node with value `key` is inserted as the left child of the current node.
- Otherwise, we inspect the right subtree, either to make another recursive call, or to insert the new node as the right child of the current node.

### Challenge

Can you think of how to remove a node from a binary search tree, such that after the removal the nodes in the tree will still be ordered?

*Hint: There are three possible scenarios, which need to be handled differentlly. In the first scenario, the node to be removed is a leaf node. In the second scenario, the node to be removed has only one child. In the last scenario, the node to be removed has two children.*

## Tutorial Project

The tutorial exercises are based on Ruby code provided in a file 'treelab.rb' that you can download from eLearn. Ensure that the file 'treelab.rb' is in the same directory as when you start the IRB session. Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'treelab.rb'
=>  true

>>  include TreeLab
=>  Object
```

T23. Let us create a new binary search tree object, and visualize it.

```
>>  bst = BinarySearchTree.new

>>  view_tree(bst)
=>  true
```

This binary search tree is currently empty.

T24. Our first exercise is to re-create the binary search tree in Figure 2.5. To do this, we begin by setting the root node to a new node with value 29.

```
>>  bst.setRoot(Node.new(29))
=>  Value: 29, left child: nil, right child: nil
```

T25. Because a binary search tree is a special form of binary tree, it has all the binary tree operations. In addition, it also allows us to insert and search for a specific key value. To insert a new node, we specify the value that the new node will have.

```
>>  bst.insert(14)
=>  Value: 14, left child: nil, right child: nil
```

The operation `insert` will call the recursive helper method `insertbst` shown in Figure 2.7. It works by first searching for the "correct" position to insert the new value, and then creating a new node in that position. In this case, because 14 is smaller than 29, it is inserted as the left child of the root node.

T26. Insert the other nodes as well. Observe the visualization window. Note how for each insertion, we first search for the right "position" to place the new node. This results in one or more nodes being visited during the search, which are indicated by the animation in the visualization window.

```
>>  bst.insert(2)
=>  Value: 2, left child: nil, right child: nil

>>  bst.insert(24)
=>  Value: 24, left child: nil, right child: nil

>>  bst.insert(27)
=>  Value: 27, left child: nil, right child: nil

>>  bst.insert(50)
=>  Value: 50, left child: nil, right child: nil

>>  bst.insert(71)
=>  Value: 71, left child: nil, right child: nil
```

T27. Do you think the structure of the tree will be the same if we insert the nodes in a different order? Why?

T28. We will now search for value 50 in this tree.

```
>>  bst.search(50)
=>  Value: 50, left child: nil, right child: 71
```

The operation `search` will call the recursive helper method `searchbst` shown in Figure 2.6. It first visits the root node, and because the key (50) is greater than the value of the root node (29), it goes to the right subtree and finds the node with value 50 there.

T29. We will now try searching for a non-existent value.

```
>>  bst.search(51)
=>  nil
```

As expected, it returns `nil`, after navigating to the leaf level and still not finding the key.

T30. Let's attach a counting probe to see how many nodes are visited in order to search for a particular value.

```
>>  Source.probe("searchbst", 2, :count)
=>  true

>>  count { bst.search(50) }
=>  2
```

We only need to visit two nodes. The first one is the root node. The second one is the node with value 50.

T31. Let's search for a value that does not exist.

```
>>  count { bst.search(90) }
=>  3
```

In this case, it goes to root node, the right child, and finally the leaf node of the right subtree, and still does not find anything.

T32. Let's search for a non-existent value that would have been found in the left subtree.

```
>>  count { bst.search(25) }
=>  4
```

Do you know why the answer is 4? What is the maximum number of nodes visited in this binary search tree?

T33. We will now experiment with creating a new binary search tree with random values. Start by setting a root node to this new tree.

```
>>  bst2 = BinarySearchTree.new

>>  view_tree(bst2)

>>  bst2.setRoot(Node.new(rand(100)))
```

The call `rand(100)` generates a random number between 1 and 100. So each time you run this method, it will give you a different number.

T34. Experiment by repeatedly inserting a new node with random values.

```
>>  bst2.insert(rand(100))
```

T35. Try to search for different values in the tree you have just created. Anticipate which nodes will be visited in the search.

## 2.5 Summary

In this chapter, we learn about a new data structure, tree, used to store hierarchical relationships among data elements. In particular, we focus on binary trees, where each parent node can have at most two children nodes.

There are several ways to traverse a tree to visit all the nodes. Preorder traversal visits the current node *before*, inorder traversal does so *in between*, and postorder traversal does so *after* visiting the left and right children.

We also learn about binary search tree, which is a special type of binary tree with its elements always in order. Searching a binary search tree is more efficient than traversing all the nodes. However, inserting a new node has to be done carefully so as to preserve the ordering among data elements.

### Advanced Material (Optional)

For students interested in additional reading, including removal of nodes from a binary search tree, as well as implementations of binary tree and binary search tree in Java, please refer to Chapter 11 of the supplementary textbook [Prichard and Carrano(2011)].

## Exercises

1. Give 3 examples each of real life applications of binary trees and binary search trees respectively.
2. Andy wants to draw his family tree. Andy's parents are Bob and Mary. Andy's paternal grandparents are Bill and Melinda. Andy's maternal grandparents are Bert and Molly. Bert's parents are Bruce and Melanie. Melinda's parents are Bernie and Martha. Draw a binary tree that represents these relationships.
3. Refer to the binary tree in Figure 2.5.
   a) Which is the root node?
   b) Which are the leave nodes?
   c) Which nodes are the parents of 14?
   d) Which nodes are the children of 14?
   e) Which nodes are the descendants of 14?
   f) Which nodes are the ancestors of 14?
   g) Which nodes are the siblings of 14?
4. Suppose we run the following operations on `irb`. Do not actually run them on `irb`. Work out the answers by hand instead.

```
>> t = BinaryTree.new
>> t.setRoot(Node.new("A"))
>> t.root.setLeft(Node.new("B"))
>> t.root.setRight(Node.new("C"))
>> t.root.left.setLeft(Node.new("D"))
>> t.root.right.setRight(Node.new("E"))
>> t.root.right.right.setLeft(Node.new("F"))
>> t.root.left.setRight(Node.new("G"))
```

   a) What is the output of calling `t.root.right`?
   b) What is the output of calling `t.root.left.right`?
   c) What is the height of the tree?
   d) What is the parent of "E"?
   e) What is the order of nodes visited for preorder traversal?
   f) What is the order of nodes visited for inorder traversal?
   g) What is the order of nodes visited for postorder traversal?

5. Andy starts a chain email. He sends an email to two of his friends, with an instruction to either forward it to two other friends (who have never received it) each, or not to forward it at all. The chain of email forwarding can be represented by a binary tree. If there are 100 people who end up sending an email:
   a) What is the maximum possible height of this binary tree?
   b) What is the minimum possible height of this binary tree?
6. Implement a non-recursive versions of the following traversal methods:
   a) preorder
   b) inorder
   c) postorder

   *Hint: you may use any data structure that you have learnt previously.*
7. Build a binary search tree (based on alphabetical ordering) for the words `Lexus`, `Mercedes`, `BMW`, `Audi`, `Lamborghini`, `Chevrolet`, `Porsche`, `Maserati`, `Lotus`.
   a) Draw the binary search tree obtained by inserting the words in the order they are listed above.
   b) Draw the binary search tree obtained by inserting the words in alphabetical order. We will first insert `Audi`, followed by `BMW`, `Chevrolet`, and so on in alphabetical order.
8. For each of the above binary search trees, how many comparisons are needed to locate the following words:
   a) `Audi`
   b) `Lamborghini`
   c) `Maserati`
   d) `Porsche`
9. Instead of preorder visits, is it possible to implement `searchbst` by using postorder or inorder visits? In other words, for postorder, we check the current node's value after checking the children's. For inorder, we check in between.

   Among the possible traversal orders, which is most efficient?
10. If a binary search tree with $n$ nodes is not full, what is the worst case complexity of `searchbst` in terms of `n`? What is the shape of the tree for this worst case?
11. We would like to design a sorting algorithm called `treesort`. It works by inserting each data element into a binary search tree. To print the data elements in a sorted order, we traverse the tree in a certain order.
    a) Which traversal method do we use for this purpose?
    b) Write the Ruby code for such an algorithm which takes as input an array `a` of size `n`. You may use the functions we have defined earlier in this chapter.
    c) What is the complexity of the above algorithm?
12. Design implementations of the `enqueue` and `peek` operations of a priority queue with a binary search tree as the underlying data structure.

# 3

# It's A Small World After All

### *Modeling networks with graphs*

In the previous chapter, we learn how a tree allows us to represent hierarchical relationships. However, a tree cannot represent all relationships, because many relationships are simply connecting pairs of entities together (without a hierarchical sense). For example, while parent-child relationships are hierarchical, friendships often are just about which two people are related to one another.

In this chapter, we will learn about a new type of non-linear data structure called *graph*, which allows us to represent many different types of relationships.

## 3.1 Real-world Networks

Many things that we experience in the real world are naturally inter-dependent and interconnected. Here are just some examples of such connected structures that we usually call "networks":

**Transportation.** Networks are commonly found in many transportation infrastructures. Airlines fly from one city to the next. The connectivity of departure and arrival cities can be represented as a network. When you take a train, it goes from one stop to the next stop, and we can link together each pair of previous and next stops, resulting in a train network.

**Communication.** Similarly, in our telecommunications infrastructure, telcos around the world maintains servers that are connected to one another to support our communication needs, be it phone calls, television, or Internet services.

**Web.** When we view a web page on our browser, the page has hyperlinks that link to other pages on the Web. When you click on a hyperlink, you go on to the next page, which again has a number of other hyperlinks connecting to yet other pages. The collective

pages on the Web, and how they link to one another, represent one of the largest networks that exist today.

**Dependency.** In a university degree program, to take a particular course, there may be one or more pre-requisite courses. This precedence or dependency relationship can be represented as a network, where we create a link from a course A to a course B, if A is a prerequisite for B. We will revisit this structure later in Section 3.4.

**Social** The type of network that most of us are probably familiar with is social network. When we visit Facebook, we go there to see the postings of our friends. When we visit Twitter, we see the new tweets that are posted by people that we follow. A social network connects each person to other people whom we have indicated as friends (Facebook), or people whom we follow (Twitter).

These are just but a subset of network structures that we encounter in real life. There are many others. Can you think of a few more?

### Six Degrees of Separation

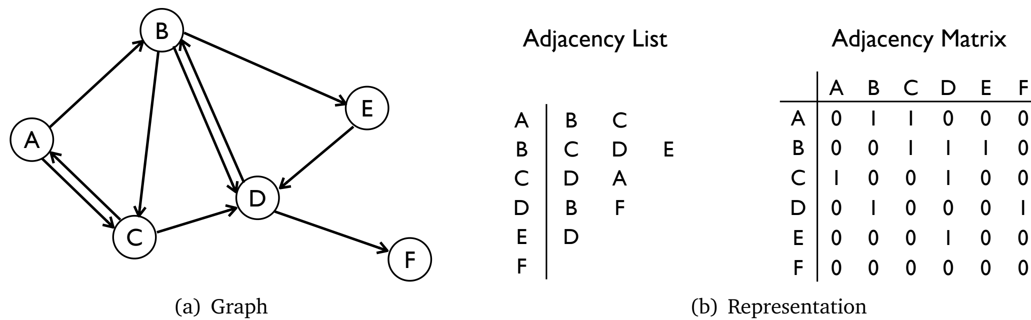
An interesting finding by social network scientists is the "small world" property of many real world networks. What it means is that on average, two entities in a network are connected to each other by a surprisingly small number of sequential edges.

In particular, the term "six degrees of separation" was popularized by an experiment[1] conducted in the 1960's by a psychologist by the name of Stanley Milgram. In this experiment, Milgram sent a number of packets to people in two US cities (Omaha, Nebraska, and Wichita, Kansas). Inside the packet was the name of a target person in Boston, Massachusetts. Each recipient was instructed to forwad the packet to the target person if he or she knows the target person. Otherwise, the recipient was to forward it to an acquaintance who were more likely to know the target person. Out of 296 packets sent, eventually 64 reached the target person. The average number of hops from a recipient to the target person was found to be around six, thus suggesting that people in the US were separated by merely six degrees[2].

This finding was popularized further by a game, called "Six Degrees of Kevin Bacon"[3]. Kevin Bacon, being one of the most prolific actors in Hollywood, had collaborated with many other actors. Two actors are directly related if they appear in the same movie together. The game revolves around who can name how an actor (e.g., Johnny Depp) is related to Kevin Bacon, through a series of actor relationships. For example, Johnny Depp and Helena Bonham Carter appeared in Dark Shadows. Helena Bonham Carter and Kevin Bacon appeared in Novocaine. In this case, we say that Bacon number of Johnny Depp is 2, because he is connected to Kevin Bacon in two hops (through Helena Bonham Carter).

In this chapter, we will explore how networks can be modeled by graph data structures, as well as how some operations on the graph allow us to explore the connectivity of different entities.

[1] S. Milgram, The small world problem, Psychology Today, 1967.
[2] `http://en.wikipedia.org/wiki/Small-world_experiment`
[3] `http://en.wikipedia.org/wiki/Six_Degrees_of_Kevin_Bacon`




| | | | |
|---|---|---|---|
| A | B | C | |
| B | C | D | E |
| C | D | A | |
| D | B | F | |
| E | D | | |
| F | | | |

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 | 0 | 0 |
| B | 0 | 0 | 1 | 1 | 1 | 0 |
| C | 1 | 0 | 0 | 1 | 0 | 0 |
| D | 0 | 1 | 0 | 0 | 0 | 1 |
| E | 0 | 0 | 0 | 1 | 0 | 0 |
| F | 0 | 0 | 0 | 0 | 0 | 0 |

(a) Graph (b) Representation

**Figure 3.1:** *Example of a directed graph*

## 3.2 Graph

We will first explore the terminologies associated with graph data structures, before we look at how to represent graphs, as well as identify some variants of graph.

### Terminologies

A graph $G$ contains a set of **vertices** $V$ and a set of **edges** $E$. Figure 3.1(a) shows an example of a graph. This graph contains seven vertices. We refer to each vertex by its label. In this case, $V$ is the set of vertices $\{A, B, C, D, E, F\}$. The set of edges $E$ encode the connectivity structure between pairs of vertices. In this graph, we are modeling **directed edges**, i.e., there is a **source vertex** and a **target vertex** for each edge. When an edge exists from a vertex to another vertex, we draw a line from the source vertex to the target vertex. For example, $A$ (source) has two outgoing edges, to $B$ and to $C$ (targets) respectively.

### Adjacency List vs. Adjacency Matrix Representation

Figure 3.1(a) shows a visual representation of a graph with nodes as vertices and lines as edges. To represent graphs in computations, we frequently use one of the following representations.

**Adjacency List.** The first representation of a graph is adjacency list. In this representation, each vertex (source) has a list of other vertices that it connects to (a list of targets). The left half of Figure 3.1(b) shows this representation for the graph in Figure 3.1(a). For instance, $A$'s list consists of $B$ and $C$; $B$'s list has $C$, $D$, and $E$; and so on.

The ordering of vertices in this adjacency list does not affect the connectivity structure in terms of which vertex is connected to which other vertex. However, as we will see later, the ordering in the list may affect the running of some graph algorithms.

**Adjacency Matrix.** The second common representation of a graph is adjacency matrix. For a graph with $n$ vertices, we use a square matrix of dimension $n \times n$. Every row corresponds to each source vertex. Every column corresponds to each target vertex. Each element has a value of 1 if there exists an edge between the source and the target vertices. Otherwise, the element has a value of 0. For example, the first row corresponds to $A$. Since $A$ is connected to $B$ and $C$, we see 1's in $B$ and $C$'s column of this row, and 0's in other columns.

### Operations

A graph consists of a set of vertices and edges, and therefore it supports a number of operations for addition and removal of vertices and edges.

The `Graph` object has the following operations:

- `new` creates an empty graph
- `vertices` returns the list of vertices in this graph
- `addVertex(vertex)` inserts a new vertex into the graph's list of vertices
- `deleteVertex(vertex)` removes the specified vertex and its edges from the graph
- `addEdge(vertex1, vertex2)` inserts a new edge from `vertex1` to `vertex2`
- `deleteEdge(vertex1, vertex2)` removes the edge from `vertex1` to `vertex2`

The `Vertex` object has the following operations:

- `new(label)` creates a new vertex with the given label
- `adjList` returns the list of target vertices that this vertex has edges to

### Connectivity and Paths

The **indegree** of a vertex is the number of incoming edges that the vertex has. Its **outdegree** is the number of outgoing edges it has. For example, in Figure 3.1(a), $B$ has an indegree of 2 (corresponding to edges from $A$ and $D$), and outdegree of 3 (corresponding to edges to $C$, $D$, $E$). We refer to the set of target vertices in a source vertex's adjacency list as its neighbors.

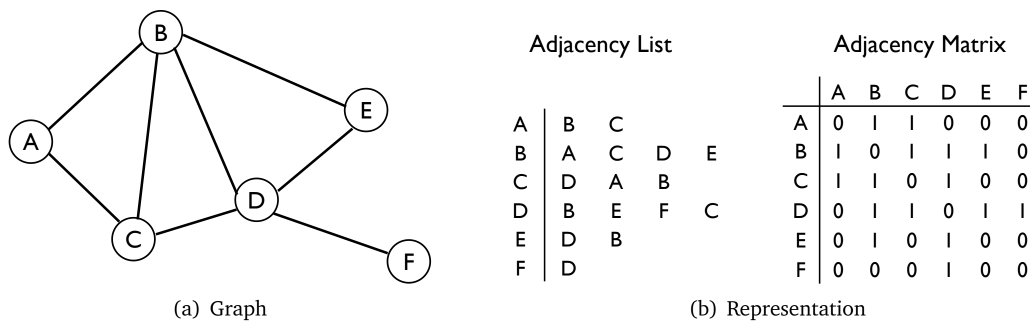
A **path** is a sequence of vertices such that for every pair consecutive vertices in the sequence, there exists an edge from the first vertex to the second vertex in the pair. In Figure 3.1(a), ($A$, $B$, $E$) is a path of length 2 (because it involves 2 edges). ($A$, $C$, $D$, $B$, $E$) is another path, with length 4. There could be multiple paths from a vertex to another vertex. For instance, both of these example paths begin at $A$ and end at $E$.

A path that begins and ends with the same vertex is a **cycle**. Figure 3.1(a) contains cycles, such as ($A$, $C$, $A$), ($A$, $B$, $C$, $A$), ($B$, $E$, $D$, $B$), and several others.

### Undirected vs. Directed

The discussion up to now has assumed that the graph models directed edges. An alternative graph model may have edges that are undirected. Figure 3.2(a) shows an example of such a graph. Notably, there is no "arrow" indicating direction. Such graphs are commonly used to represent relationships that are symmetrical in nature.

To illustrate, a social network graph may use directed edges if a vertex $v_1$ may indicate $v_2$ as a friend, but $v_2$ may not indicate $v_1$ as a friend. An example of this would be the Twitter follow network. Another social network graph may use undirected edges if both vertices have to agree before a friendship relationship is created. An example of this would be the Facebook friendship network.




Adjacency List

| | | | | |
|---|---|---|---|---|
| A | B | C | | |
| B | A | C | D | E |
| C | D | A | B | |
| D | B | E | F | C |
| E | D | B | | |
| F | D | | | |

Adjacency Matrix

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | 0 | 1 | 1 | 0 | 0 | 0 |
| B | 1 | 0 | 1 | 1 | 1 | 0 |
| C | 1 | 1 | 0 | 1 | 0 | 0 |
| D | 0 | 1 | 1 | 0 | 1 | 1 |
| E | 0 | 1 | 0 | 1 | 0 | 0 |
| F | 0 | 0 | 0 | 1 | 0 | 0 |

(a) Graph (b) Representation

**Figure 3.2:** *Example of an undirected graph*

To represent undirected graphs, we can similarly use either an adjacency list or an adjacency matrix. The main difference is that every edge is represented twice, once in the first vertex's adjacency list or row in adjacency matrix, and another time in the second vertex's adjacency list or row in adjacency matrix. An alternative way of viewing an undirected graph is to view it as a directed graph, whereby every undirected edge corresponds to two directed edges in opposite directions. Figure 3.2(b) shows the representations for the undirected graph in Figure 3.2(a).

In subsequent discussions, unless otherwise specified, we will use directed graph by default. However, most of the concepts will also be transferable to undirected graphs.

## Tutorial Project

The tutorial exercises are based on Ruby code provided in a file 'graphlab.rb' that you can download from eLearn. Ensure that the file 'graphlab.rb' is in the same directory as when you start the IRB session. Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'graphlab.rb'
=>  true

>>  include GraphLab
=>  Object
```

T1. Create a new graph `g`:

```
>>  g = Graph.new
=>  []
```

Currently, the graph is empty. It does not have any vertex yet.

T2. Make a new vertex with label 1 and insert it into the graph:

```
>>  v1 = Vertex.new(1)
=>  1

>>  g.addVertex(v1)
=>  [1 ]
```

Upon adding a new vertex, the new vertex will appear on the visualization canvas. The placement of the vertex on the canvas would be random, unless we specify the x and y coordinates of the new vertex, e.g., `Vertex.new(label, x-coord, y-coord)`. The canvas has a size of approximately 600 x 600, so you may specify any coordinates from 0 to 600.

T3. Let's create another vertex, and place it on the canvas at the coordinates (100, 500).

```
>>  v2 = Vertex.new(2, 100, 500)

>>  g.addVertex(v2)
```

T4. Create another six vertices and insert them into the graph. You may optionally specify the coordinates of each vertex to have a better visualization layout.

```
>>  v3 = Vertex.new(3)

>>  g.addVertex(v3)

>>  v4 = Vertex.new(4)

>>  g.addVertex(v4)

>>  v5 = Vertex.new(5)

>>  g.addVertex(v5)

>>  v6 = Vertex.new(6)

>>  g.addVertex(v6)

>>  v7 = Vertex.new(7)

>>  g.addVertex(v7)

>>  v8 = Vertex.new(8)

>>  g.addVertex(v8)
```

The graph now has eight vertices that are not connected to one another.

T5. We can remove an existing vertex using `deleteVertex` operation:

```
>>  g.deleteVertex(v8)
=>  [1 , 2 , 3 , 4 , 5 , 6 , 7 ]
```

T6. At any time, we can check the list of vertices of this graph:

```
>>  g.vertices
=>  [1 , 2 , 3 , 4 , 5 , 6 , 7 ]
```

T7. Let us now add an edge to this graph:

```
>>  g.addEdge(v1, v4)
=>  true
```

You can now see in the visualization window that there is a directed edge between the vertex with label 1 to the vertex with label 4.

T8. Let us now practise adding more edges to this graph:

```
>>  g.addEdge(v1, v7)
=>  true

>>  g.addEdge(v7, v4)
=>  true

>>  g.addEdge(v4, v5)
=>  true

>>  g.addEdge(v4, v2)
=>  true

>>  g.addEdge(v5, v2)
=>  true

>>  g.addEdge(v5, v6)
=>  true

>>  g.addEdge(v2, v6)
=>  true

>>  g.addEdge(v6, v3)
=>  true

>>  g.addEdge(v6, v7)
=>  true

>>  g.addEdge(v3, v7)
=>  true

>>  g.addEdge(v3, v1)
=>  true
```

T9. Sometimes we need to delete an edge, which can be done using `deleteEdge` operation.

```
>>  g.deleteEdge(v3, v1)
=>  true
```

Figure 3.3 shows how the graph we created above may look like. You may not get exactly the same layout, because the positioning of the vertices are random. However, you should still see the same number of vertices, with the same labels and the same connectivity.

T10. In this implementation, we use an adjacency list representation of the graph. We can check the list of other vertices that a vertex is connected to by calling the `adjList` operation.

```
>>  v1.adjList
=>  [4, 7]
```

This shows that from vertex 1, there are two directed edges, to vertex 4 and vertex 7 respectively.

T11. Check the adjacency lists of other vertices as well.

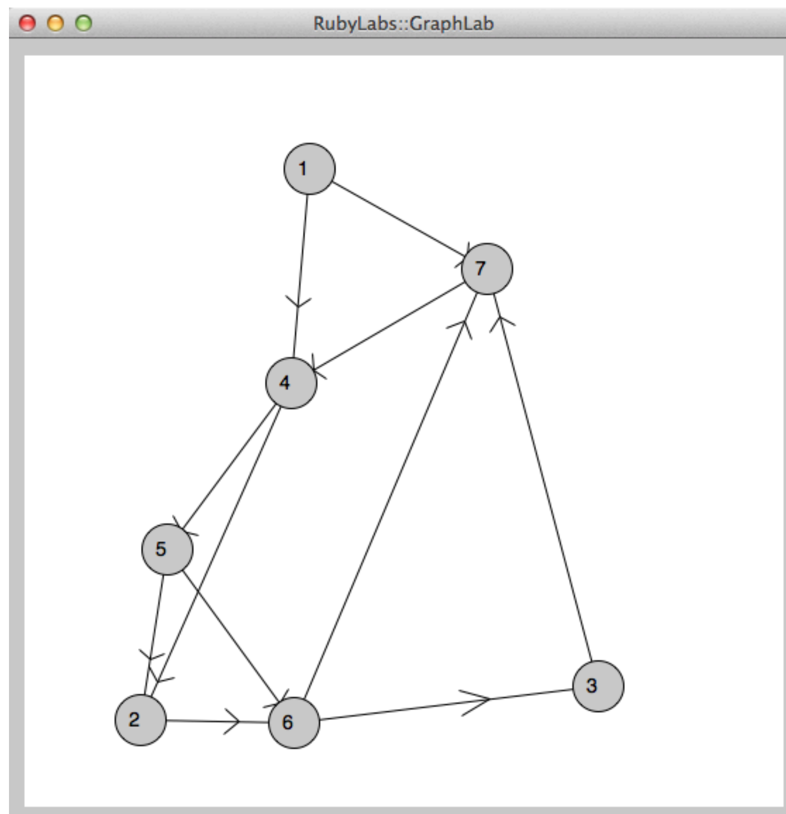



**Figure 3.3:** *Visualization of graph after the first tutorial project*

```
>>  v2.adjList
=>  [6]

>>  v3.adjList
=>  [7]

>>  v4.adjList
=>  [5, 2]

>>  v5.adjList
=>  [2, 6]

>>  v6.adjList
=>  [3, 7]

>>  v7.adjList
=>  [4]
```

T12. What would be the adjacency matrix representation of this graph?

```
def traversal(graph, vertex, option)
  graph.clearAll
  case option
  when :dfs
    dfs_traversal(vertex)
  when :bfs
    bfs_traversal(vertex)
  else
    return "Invalid option."
  end
  return nil
end
```

(a) `traversal`

```
def visit(vertex)
  vertex.visited = true
  p vertex
end
```

(b) `visit`

**Figure 3.4:** *Shared methods for traversing a graph*

## 3.3 Traversals of a Graph

Like a tree, a graph is a non-linear structure. What it means is that there could be multiple ways to "iterate" or to traverse a non-linear structure. Graph traversal is even more complex than tree. In a tree, every traversal begins with the root node, and every non-root node is "reachable" from the root node. In a graph, there is no single "root node". Instead, traversal may begin from any vertex in the graph.

There are two main ways to traverse a graph from any particular vertex: *depth-first search* and *breadth-first search*. In the following, we will explore how to write both traversal algorithms. First of all, in Figure 3.4(a), we show a common `traversal` method that takes in a `graph`, a `vertex` where we will begin the traversal from, as well as an `option`. At the beginning of each traversal, the line `graph.clearAll` clears or resets the list of vertices that we have visited. If the specified `option` is `:dfs`, we will call the depth-first search traversal method `dfs_traversal`. Alternatively, if the option is `:bfs`, we will call the breadth-first search traversal method `bfs_traversal`.

### Depth-First Search (DFS)

In depth-first search, we prioritize going deep into the graph. Each time we encounter several choices of vertices to explore (e.g., $v_1$ and $v_2$), we will pick one vertex (e.g., $v_1$) and

```
def dfs_traversal(vertex)
  visit(vertex)
  for i in 0..(vertex.adjList.length-1)
    neighbor = vertex.adjList[i]
    if !neighbor.isVisited?
      dfs_traversal(neighbor)
    end
  end
end
```

**Figure 3.5:** *Recursive implementation of depth first search*

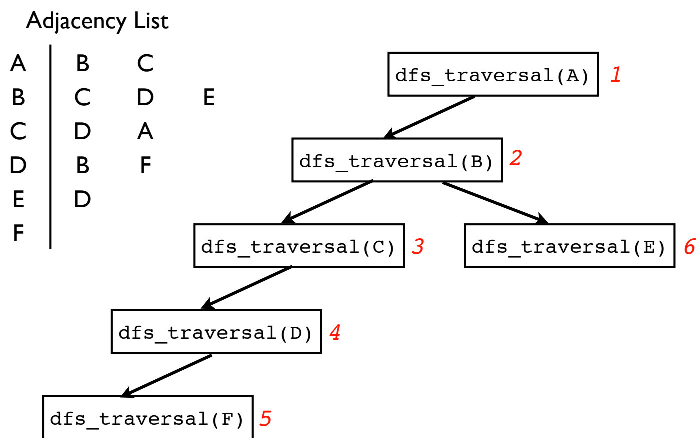



**Figure 3.6:** *Trace of depth first search for graph in Figure 3.1(a)*

keep exploring all other connected vertices, before moving on to the second vertex $v_2$, and so on.

Figure 3.5 illustrates a recursive method for depth-first search. Every time we encounter a vertex, we will visit this vertex. We will then iterate through each neighbor (according to the ordering in the vertex's adjacency list), and recursively traverses this neighbor (before moving on to the next neighbor).

The definition of "visit" here typically depends on each application scenario. It may involve a search, a comparison, or an operation with the visited vertex. In our context here, the method `visit` is illustrated in Figure 3.4(b), where we simply indicate that this vertex has been visited, and print out its label.

To illustrate the working of `dfs_traversal`, we will trace the execution of this recursive algorithm with the graph in Figure 3.1(a), beginning with the vertex $A$. This trace of recursive calls is shown in Figure 3.6. `dfs_traversal(A)` results in first visiting $A$ (1st). It then leads to recursive calls to $A$'s neighbor according to the ordering in $A$'s adjacency list. We visit $A$'s first neighbor $B$ (2nd). This recursively goes on to $B$'s neighbor $C$ (3rd), and $C$'s neighbor $D$ (4th). We then visit $D$'s neighbor $F$ (5th), since $D$'s other neighbor $B$ has been visited before. By this time, $D$'s neighbors are all visited. Same with $C$'s neighbors.

```
def bfs_traversal(vertex)
  q = Queue.new
  visit(vertex)
  q.enqueue(vertex)
  while(!q.isEmpty?)
    v = q.dequeue
    for i in 0..(v.adjList.length-1)
      neighbor = v.adjList[i]
      if !neighbor.isVisited?
        visit(neighbor)
        q.enqueue(neighbor)
      end
    end
  end
end
```

**Figure 3.7:** *Queue-based implementation of breadth first search*

We then visit the remaining unvisited neighbor of $B$, which is $E$ (6th). Finally, the recursion ends, resulting in order of traversal $A$, $B$, $C$, $D$, $F$, $E$.

## Breadth-First Search (BFS)

Unlike depth-first search where we prioritize going deep into the graph, in breadth-first search we prioritize visiting vertices closer to the starting vertex. For instance, suppose we traverse Figure 3.1(a), starting from $A$. After visiting $A$, we will visit all vertices reachable from $A$ through paths of length 1 (direct neighbors), followed by vertices reachable by paths of length 2 (neighbors of neighbors), and so on.

Adjacency List

| | | | |
|---|---|---|---|
| A | B | C | |
| B | C | D | E |
| C | D | A | |
| D | B | F | |
| E | D | | |
| F | | | |

| step | dequeued | visited/enqueued | queue |
|---|---|---|---|
| 0 | | A | [A] |
| 1 | A | B, C | [B, C ] |
| 2 | B | D, E | [C, D, E] |
| 3 | C | | [D, E] |
| 4 | D | F | [E, F] |
| 5 | E | | [F] |
| 6 | F | | [ ] |

**Figure 3.8:** *Trace of breadth first search for graph in Figure 3.1(a)*

Figure 3.7 shows an implementation using a queue. In each loop of the iteration, we dequeue a vertex, and visit all its neighbors, and then enqueue them so that we can determine their neighbors to be visited later. Hence, the queue maintains the order in which vertices have been visited. Every dequeue returns the earliest vertex, whose neighbors are yet to be visited.

The trace of execution for Figure 3.1(a) starting from $A$ is shown in Figure 3.8. Initially, the queue contains only the starting vertex $A$. In the first iteration, we dequeue $A$, and visit as well as enqueue vertices directly reachable from $A$, which are $B$ and $C$. Next, we dequeue $B$ and visit/enqueue its unvisited neighbors $D$ and $E$. When dequeueing $C$, we realize that $C$'s neighbors were already visited or are already in the queue. By now, we have visited all vertices reachable through path length of 2 from $A$. The next vertex visited is the only one with path length 3, which is $F$. The traversal order is $A$, $B$, $C$, $D$, $E$, $F$.

## Tutorial Project

The tutorial exercises are based on Ruby code provided in a file 'graphlab.rb' that you can download from eLearn. Ensure that the file 'graphlab.rb' is in the same directory as when you start the IRB session. Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'graphlab.rb'
=>  true

>>  include GraphLab
=>  Object
```

T13. We will begin by traversing the graph constructed in the previous tutorial project (Figure 3.3) with depth first search, starting from vertex 1.

```
>>  g.dfs(v1)
 1
 4
 5
 2
 6
 3
 7
=>  nil
```

Watch the animation. Is it as you expected?

We begin from vertex 1 (1st), followed by visiting its first neighbor 4 (2nd). Next is 4's first neighbor 5 (3rd), which in turn visits its first neighbor 2 (4th). From 2, we go to 6 (5th), then 3 (6th), and finally 7 (7th). By now, all the vertices are already visited.

T14. Try depth-first search again but with other starting vertices. First, work out by hand what you expect to be the sequence. Then, run it on `irb` to verify your answer.

```
>>  g.dfs(v2)

>>  g.dfs(v3)

>>  g.dfs(v4)

>>  g.dfs(v5)

>>  g.dfs(v6)

>>  g.dfs(v7)
```

What do you observe? Does every depth first search always visit all the vertices?

T15. We now traverse the graph in Figure 3.3 using breadth first search, starting from vertex 1.

```
>>  g.bfs(v1)
 1
 4
 7
 5
 2
 6
 3
=>  nil
```

Note how the sequence is different from the previous depth first search traversal.

We first visit 1 (1st), and enqueue 1's neighbors 4 and 7. We dequeue and visit 4 (2nd), followed by enqueuing 4's neighbors 5, 2. Next, we visit 7 (3rd), but do not put anything in the queue since its neighbor 4 is already visited. We then visit 5 (4th), which enqueues its remaining neighbor 6. Afterwards, visit 2 (5th), but no new insertion to the queue, because 6 is already in the queue. After visiting 6 (6th), we enqueue 3. Finally, we visit 3 (7th).

T16. Try breadth-first search again but with other starting vertices. First, work out by hand what you expect to be the sequence. Then, run it on `irb` to verify your answer.

```
>>  g.bfs(v2)

>>  g.bfs(v3)

>>  g.bfs(v4)

>>  g.bfs(v5)

>>  g.bfs(v6)

>>  g.bfs(7)
```

What do you observe? Does every breadth first search always visit all the vertices?

T17. To better understand the algorithms for depth first search and breadth first search traversal, code your own Ruby methods. Put together the algorithms in Figure 3.4(a) to Figure 3.7 into a text file, and name the file "traversal.rb".

```
>>  load "traversal.rb"
=>  true
```

The traversal method takes in the graph, the starting vertex for traversal, as well as an option, which could be `:dfs` for depth first search, and `:bfs` for breadth first search.

```
>>  traversal(g, v1, :dfs)
 1
 4
 5
 2
 6
 3
 7
=>  nil

>>  traversal(g, v1, :bfs)
 1
 4
 7
 5
 2
 6
 3
=>  nil
```

Do they give you the same sequence of visits as the animation you have seen previously?

Try the traversal starting from different vertices. Add and remove vertices, and see if the traversals are as expected.

## 3.4 Topological Sorting

Having looked at the basics of graph, and different methods of traversal, we are now ready to look at one of the graph-based applications.

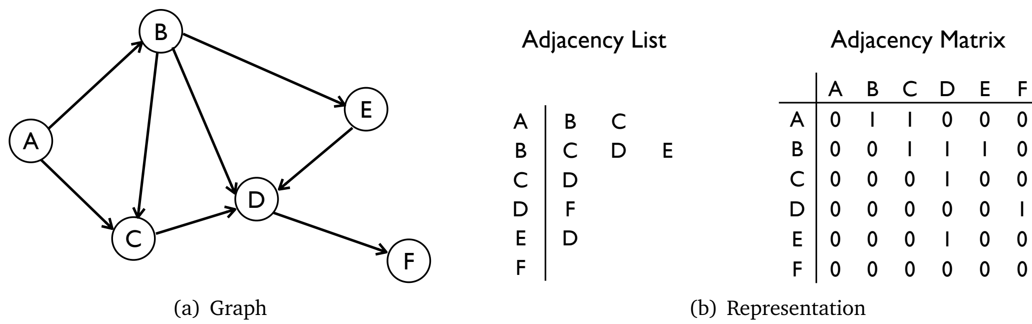
### Directed Acyclic Graph (DAG)

Earlier, we have defined a cycle as a path that begins and ends with the same vertex. If a directed graph has no cycle, we call it a *directed acyclic graph* or DAG. Figure 3.9(a) illustrates an example of a DAG, with adjacency list and matrix representations shown in Figure 3.9(b).

Such a graph is commonly used to represent dependency or precedence relationships. This is the case when to complete a task, we need to first complete another task. We can represent this with a DAG, with tasks as vertices, and edges connecting the preceding task to the following task. For example, if vertices are university courses, a DAG may have edges leading to a course from its prerequisites. Suppose that such a graph were to contain a cycle, e.g., to take a course $c_1$ you would first need to take $c_2$, but to take $c_2$ you would first need to take $c_1$, then it would not be possible to complete the courses in the cycle while still respecting the precedence relationship.

### Topological Ordering

Topological ordering in a DAG is an ordering of *all* vertices in the DAG, $(v_1, v_2, \ldots, v_n)$, such that for any edge involving two vertices, from $v_i$ to $v_j$, we have $i < j$. In other words, all the edges in this ordering “point forward”.




(a) Graph (b) Representation

**Figure 3.9:** *Example of a directed acyclic graph*

```
def topsort(graph)
  graph.clearAll
  s = Stack.new
  for i in 0..(graph.vertices.length-1)
    vi = graph.vertices[i]
    if !vi.isVisited?
      topsort_dfs(vi, s)
    end
  end
  return s
end
```

(a) `topsort`

```
def topsort_dfs(vertex, stack)
  visit(vertex)
  for i in 0..(vertex.adjList.length-1)
    neighbor = vertex.adjList[i]
    if !neighbor.isVisited?
      topsort_dfs(neighbor, stack)
    end
  end
  stack.push(vertex)
end
```

(b) `topsort_dfs`

**Figure 3.10:** *Topological Sort*

In any DAG, there is at least one topological ordering. For the DAG in Figure 3.9(a), ($A$, $B$, $E$, $C$, $D$, $F$) is one topological ordering. For any edge in the graph, the source vertex appears earlier than the target vertex in the topological ordering. However, a DAG could also have more than one topological ordering. ($A$, $B$, $C$, $E$, $D$, $F$) is the second topological ordering for the DAG in Figure 3.9(a).

To come first in a topological ordering, a vertex cannot have any incoming edge in the graph. In any DAG, there is always *at least* one such vertex.

### Topological Sorting

We will now explore an algorithm to find one of the topological orderings in a DAG. We begin with the observation that for any path that exists in the graph, the vertices that are part of that path must appear in the same sequential ordering as the path. However, there may be some other interleaving vertices as well.

The traversal method that explores a graph by following the sequential edges in a path is depth-first search. Note that when we call the recursive `dfs_traversal(v)` on a vertex $v$, we do not return from this call until we have recursively visited all the other vertices that $v$ has a path to. This suggests that by the time we return from $v$'s DFS traversal, we would already know which other vertices should appear *after* $v$ in a topological ordering.

Therefore, one way to discover a topological ordering is to ensure that vertices whose recursive DFS terminates earlier will appear later in the topological ordering. The right data structure for this purpose is a stack, by pushing each vertex into a stack as we complete the recursive DFS on that vertex.

In Figure 3.10(a), we present an algorithm `topsort`, which takes as input a `graph` (which has to be a directed acyclic graph), and produces as output a stack. This stack contains the vertices of the graph. If we pop the vertices from the stack one by one, they will appear according to a topological ordering. This algorithm first resets the visiting record of the graph (`graph.clearAll`). It then creates a new stack to contain the topological ordering. It then iterates over each vertex in the graph, and runs a special type of DFS `topsort_dfs` (Figure 3.10(b)) on each vertex. This DFS traversal is identical to the previously defined recursive algorithm for depth-first search, except that it pushes a vertex into the stack when its recursive call completes.

Note that this algorithm will return only one of the valid topological orderings. For the same graph, depending on the ordering of vertices in each adjacency list, the sequence of DFS may vary slightly, potentially resulting in a different topological ordering. For example, for $B$'s adjacency list in Figure 3.9(b), `topsort` will return the topological ordering ($A$, $B$, $E$, $C$, $D$, $F$). However, if $B$'s adjacency list were to consist of $E$, $C$, and then $D$, the output topological ordering would be ($A$, $B$, $C$, $E$, $D$, $F$).

## Tutorial Project

The tutorial exercises are based on Ruby code provided in a file ‘graphlab.rb’ that you can download from eLearn. Ensure that the file ‘graphlab.rb’ is in the same directory as when you start the IRB session. Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'graphlab.rb'
=>  true

>>  include GraphLab
=>  Object
```

T18. In this tutorial, we will explore topological sort for the DAG in Figure 3.9(a). To do this, we first create the vertices A to F.

```
>>  va = Vertex.new("A")

>>  vb = Vertex.new("B")

>>  vc = Vertex.new("C")

>>  vd = Vertex.new("D")

>>  ve = Vertex.new("E")

>>  vf = Vertex.new("F")
```

T19. Create a new graph, and insert the vertices into this graph.

```
>>  g2 = Graph.new

>>  g2.addVertex(va)

>>  g2.addVertex(vb)

>>  g2.addVertex(vc)

>>  g2.addVertex(vd)

>>  g2.addVertex(ve)

>>  g2.addVertex(vf)
```

T20. Let us now reproduce the connectivity in Figure 3.9(a).

```
>>  g2.addEdge(va, vb)

>>  g2.addEdge(va, vc)

>>  g2.addEdge(vb, vc)

>>  g2.addEdge(vb, vd)

>>  g2.addEdge(vb, ve)

>>  g2.addEdge(vc, vd)

>>  g2.addEdge(vd, vf)

>>  g2.addEdge(ve, vd)
```

Compare the visualization that you get now to Figure 3.9(a). The layout may be affected by

the random placement of the vertices in the visualization window. However, the connectivity should still be the same as Figure 3.9(a).

T21. Let us now run the topological sort algorithm on this graph.

```
>>  topsort(g2)
 A
 B
 C
 D
 F
 E
=>  [A , B , E , C , D , F ]
```

After calling `topsort`, the method prints out the order of nodes visited at each line. In this case, the sequence of visits is A, B, C, D, F, E.

However, the final line prints out the stack that contains the topological sequence [A , B , E , C , D , F ], which is not identical to the order of visits.

Is this a correct topological sequence?

T22. Let us now add a new node that has an outgoing edge to vertex A, and check the topological ordering.

```
>>  vh = Vertex.new("H")
=>  H

>>  g2.addVertex(vh)
=>  [A , B , C , D , E , F , H ]
```

Note that H comes last in the list of vertices of `g2`, because H is added last.

We now add an edge from H to A.

```
>>  g2.addEdge(vh, va)
=>  true
```

T23. Run the topological sort algorithm on this new graph.

```
>>  topsort(g2)
 A
 B
 C
 D
 F
 E
 H
=>  [H, A , B , E , C , D , F ]
```

Based on the sequence of visits, vertex H is visited last. This is because the first vertex in graph `g2` is A. The algorithm in Figure 3.10(a) runs depth first search from A, and visites the rest of the vertex, except for H, which is visited in the next depth first search.

However, the topological sequence [H, A , B , E , C , D , F ] is correct because H is before A.

T24. Let us now add a new node that is expected to be last in topological sort. Such a vertex would have an incoming edge from F.

```
>>  vj = Vertex.new("J")
=>  J

>>  g2.addVertex(vj)
=>  [A , B , C , D , E , F , H , J ]

>>  g2.addEdge(vf, vj)
=>  true
```

T25. Run the topological sort algorithm on this new graph.

```
>>  topsort(g2)
 A
 B
 C
 D
 F
 J
 E
 H
=>  [H, A , B , E , C , D , F , J ]
```

Note that in the topological sequence, J comes after F, which is as expected.

T26. Add a few more vertices and edges on your own, and explore the topological sort algorithm.

## 3.5 Shortest Path

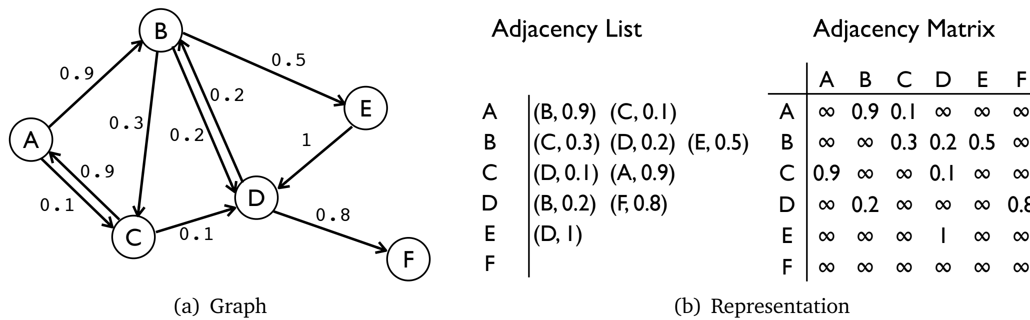
Previously, we have been looking at graphs where edges either exist or do not exist. Therefore, a graph's representation as an adjacency list or an adjacency matrix simply needs to reflect the existence of edges, where usually a binary value of 0 or 1 suffices. In addition, between two paths connecting the same source vertex and target vertex, we can only speak about the varying lengths of such paths (how many intermediate vertices exist in a path).

### Weighted Graph

In some applications, not all edges have equal importance. For example, when we use a graph to represent an inter-city road network, where vertices are cities, and edges are roads connecting two cities, we may want to associate each edge with some value that reflects either the distance travelled, the time taken, or the cost of travel along the corresponding road. In this respect, the previous binary representation is no longer sufficient.

We now define a weighted graph as a graph where each edge is associated with a weight. As indicated in the previous example, this weight may reflect various kinds of values (e.g., distance, time, cost), but we assume the weight has a consistent meaning for all edges. Figure 3.11(a) illustrates an example of a weighted graph. The numbers appearing alongside edges are their respective weights. For example, the edge from $A$ to $C$ has a weight of 0.1, but the edge from $C$ to $A$ has a higher weight of 0.9.

As shown in Figure 3.11(b), a weighted graph could also be represented as either an adjacency list or an adjacency matrix, with some minor variation. In an adjacency list, for each source vertex, its list of target vertices now also indicate the weight of each edge to a target vertex. In an adjacency matrix, we replace the previously binary values with real values reflecting the weights. However, we no longer simply use 0 to reflect an edge that does not exist, because 0 itself is a possible valid weight. In Figure 3.11(b), we set the weight of non-existent edge as infinity, because in this specific example, we model a graph where lower weight indicates lower cost (e.g., to go through a non-existent road has infinity cost).




(a) Graph

Adjacency List

| | |
|---|---|
| A | (B, 0.9) (C, 0.1) |
| B | (C, 0.3) (D, 0.2) (E, 0.5) |
| C | (D, 0.1) (A, 0.9) |
| D | (B, 0.2) (F, 0.8) |
| E | (D, 1) |
| F | |

Adjacency Matrix

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | ∞ | 0.9 | 0.1 | ∞ | ∞ | ∞ |
| B | ∞ | ∞ | 0.3 | 0.2 | 0.5 | ∞ |
| C | 0.9 | ∞ | ∞ | 0.1 | ∞ | ∞ |
| D | ∞ | 0.2 | ∞ | ∞ | ∞ | 0.8 |
| E | ∞ | ∞ | ∞ | 1 | ∞ | ∞ |
| F | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |

(b) Representation

**Figure 3.11:** *Example of a directed, weighted graph*

### Shortest Path

One of the advantages of modeling weight in a graph is the ability to associate different weights with paths in the graph. One way to compute the weight of a path is to simply sum the weights of individual edges in the path.

We can now speak about the "shortest" path, not simply in terms of which path has the smallest length, but also in terms of which path has the smallest weight. For example, in Figure 3.11(a), to go from $A$ to $B$, we could go through either path $(A, B)$, which has a length of only 1, and a weight of 0.9. Alternatively, we could use the path $(A, C, D, B)$, which has a length of 3, but a weight of only $0.1 + 0.1 + 0.2 = 0.4$. This gives us a lot more flexibility in modeling graphs, as well as modeling what it means for a path to be short.

### Algorithm for Finding Shortest Path (self-study)

Finding shortest paths connecting two vertices in a graph is an important problem with many applications. In an unweighted graph, finding shortest path is relatively simple, because by using breadth-first search traversal, we can find a way to go from the source vertex to the target vertex in as few hops as possible.

However, in a weighted graph, this simple heuristic alone is not sufficient because as we have seen previously, there could be a longer path with a smaller aggregate weight.

The most well-known algorithm for finding shortest paths in a weighted graph is *Dijkstra's shortest path algorithm*. There are many available references for this algorithm. The study of this algorithm is left to the students as self-study.

## 3.6 Summary

A graph data structure allows us to represent relationships in real-world networks. The basic principle of a graph is to related vertices with edges. There could be several variants of graph data structures, depending on the properties of these edges. For instance, whether edges are directed or undirected, as well as whether edges are binary or weighted.

To traverse a graph, we use two main strategies. Depth-first search or DFS prioritizes following edges in paths. Breadth-first search or BFS prioritizes visiting vertices closer to the starting vertex.

**Table 3.1:** *Population Sizes by Capital Cities*

| *Country* | *Capital* | *Population* |
|---|---|---:|
| Singapore | Singapore | 5.2M |
| Indonesia | Jakarta | 10.2M |
| Malaysia | Kuala Lumpur | 1.6M |
| Thailand | Bangkok | 8.3M |
| Philippines | Manila | 1.7M |
| Brunei | Bandar Seri Begawan | 0.1M |
| Vietnam | Hanoi | 6.6M |
| Cambodia | Phnom Penh | 1.5M |
| Laos | Vientiane | 6.5M |
| Myanmar | Naypyidaw | 0.9M |

If a directed graph does not contain any cycle, we call it a directed acyclic graph or DAG. DAG is commonly used to represent precedence or dependency relationships. In a DAG, there could be one or more topological orderings. In any such ordering, an edge always points forward, with the source vertex appearing earlier in the ordering than the target vertex of each edge.

If a graph (directed or undirected) has weighted edges, a path in the graph can be associated with a weight as well, which would be some form of aggregation (e.g., summation) of the component edge weights. This is a common application in shortest path problems, where we may need to find a sequence of vertices that can be visited with the lowest cost.

## Exercises

1. Scout Airways flies from Singapore to all the capitals in ASEAN countries. However, there is a return flight to Singapore only if the capital's population is larger than Singapore's. Otherwise, to return to Singapore, one needs to first fly to another capital with a return flight to Singapore. There is always a flight from any capital city to another capital city (other than Singapore) with the next largest population size (e.g., there is a flight from Bandar Seri Begawan (pop: 0.1M) to Naypyidaw (pop: 0.9M)).

   Table 3.1 lists the capital cities of each country and their respective population sizes.

   Draw a directed graph, where the vertices are the capital cities, and an edge exists from one city to another city only if there is a flight from the first to the second city.

2. You are attending a friend's birthday party. Suppose that you want to track who knows whom in the birthday party, what kind of graph: directed or undirected, will you use? Can it have cycles?

3. We can construct a call graph, with vertex representing phone numbers. An edge exists if there is at least one phone call in a month from a source vertex to a target vertex.

   Suppose that you are given two call graphs: one for the month of April, and another for the month of May. How would you use the two call graphs to determine the new phone numbers of people who have changed their phone numbers?

4. A graph has 10 vertices, $A$ to $J$. $A$ connects to $B$ and $I$. $B$ connects to $D$ and $H$. $C$ connects to $D$. $D$ connects to $H$. $E$ connects to $F$ and $G$. $F$ connects to $A$. $G$ connects to $J$. $H$ connects to $E$. $I$ connects to $B$ and $J$. $J$ connects to $A$.
   a) Represent this graph in adjacency list and adjacency matrix respectively.
   b) Which vertices have the highest indegreee?
   c) Which vertices have the highest outdegree?
   d) How many paths that do not contain any cycle (no vertex are visited twice in a path) exist from $A$ to $J$?
   e) How many cyclical paths exist in this graph?
   f) What is the sequence of vertices visited by breadth-first search beginning from $A$?
   g) What is the sequence of vertices visited by depth-first search beginning from $A$?
5. Please refer to the graph that can be found in `http://www.openarchives.org/ore/0.2/datamodel-images/WebGraphBase.jpg`.
   a) Represent this graph in adjacency list and adjacency matrix respectively.
   b) Which vertices have the highest indegreee?
   c) Which vertices have the highest outdegree?
   d) How many paths that do not contain any cycle (no vertex are visited twice in a path) exist from $A$ to $J$?
   e) How many cyclical paths exist in this graph?
   f) What is the sequence of vertices visited by breadth-first search beginning from $A$?
   g) What is the sequence of vertices visited by depth-first search beginning from $A$?
6. Implement depth-first search non-recursively using a stack.
7. A graph has $n$ vertices, with labels 0, 1, 2, $\ldots$, $n-1$. There is an edge from every vertex with label $i$ to another vertex with label $j$ if $i < j$.
   a) How many edges exist in this graph?
   b) Is there any cycle in this graph?
   c) How many topological ordering exist in this graph?
8. You are considering to take a certificate program in marketing. There are three levels of courses.
   - The first-level courses are 1A, 1B, and 1C. There are no prerequisites for first-level courses.
   - The second-level courses are 2A and 2B. The prerequisites for 2A are 1A and 1B. The prerequisites for 2B are 1B and 1C.
   - The third-level courses are 3A, 3B, and 3C. 2A is the prerequiesite for all third-level courses. In addition, to take 3B, you first need to take 2B. To take 3C, you first need to take 1C.

You need to complete the program in eight terms, taking one course per term. You have to propose the sequence of courses that you will follow over the eight terms. Figure out *all* the valid sequences of courses.

9. A graph has six vertices, labeled A to F. There are exactly two paths from A to F. The first path is (A, B, C, F). The second path is (A, D, E, F). All the edges in this graph can be found in either of this path.

   If you run topological sorting algorithm, what is the output topological ordering?

   Could there be more than one possible outputs? What affects which output you get?

10. Refer to Figure 3.11(a). Suppose we are interested in paths from $B$ to $F$.

    a) What is the shortest path from $B$ to $F$?

    b) What are the all other paths from $B$ to $F$ if we cannot visit the same vertex twice in each path?

    c) How many paths are possible if we allow visiting the same vertex more than once?

# Bibliography

[Conery(2011)] John S. Conery. *Explorations in Computing: An Introduction to Computer Science*. CRC Press, first edition, 2011.

[Prichard and Carrano(2011)] Janet Prichard and Frank Carrano. *Data Abstraction and Problem Solving with Java: Walls and Mirrors*. Prentice Hall, third edition, 2011.