DATA STRUCTURES FROM SCRATCH

A brisk introduction to essential, advanced, and persistent data structures

Alejandro Miralles

Preface

Have you ever had to implement a hash table and didn't know where to start?

Have you ever got stuck trying to balance a binary search tree?

Do you prefer reading working code over math notation?

If the answer is yes, then this book is for you!

Data Structures From Scratch was designed to help software developers to build commonly used data structures from the ground up using the Ruby programming language.

While you might never have to build a hash table at work, knowing how to do it will help you to pass the coding interview at most tech companies.

From the "Big Five" to early-stage start-ups, knowledge of data structures is required for most high-paying programming jobs.

After reading this book you won't be ready to apply for a job at Google, but you will find advanced material on the subject accessible and easy to digest.

What's inside

The journey begins with ***essential*** data structures like linked lists and stacks, then moves to ***advanced*** data structures like binary search trees and graphs, and finally, it introduces the concept of ***persistent*** data structures, such as the ones you find in functional programming languages like Clojure or Haskell.

The prose in this book is casual and jargon-free. Except for the chapters on sets and hash tables, there are almost no theoretical aspects discussed in this book. Each data structure is explained in plain English, using brief, code-centric demonstrations, on how to build them and how they work.

All chapters have roughly the same structure:

- A brief description of the data structure.
- A table that explains the data structure’s interface.
- A section that discuss implementation details. (In-depth analysis of code and time complexity.)
- Closing thoughts. (When to use the data structure, mostly.)

About the scope of this book

This book covers eleven data structures and is divided into three sections.

Essential

- Singly Linked Lists
- Doubly Linked Lists
- Circular Linked Lists
- Queues
- Stacks
- Hash Tables
- Sets

Advanced

- Binary Trees
- Binary Search Trees
- Graphs

Persistent

- Persistent Lists

How to read this book

The book was designed to be read from cover to cover.

After a first read, this book could also be used as reference material.

What do you need to follow along

The code targets Ruby 2.3.1 and it has no external dependencies. So, to follow along, you will need:

- Ruby (2.3.1+).
- You favorite text editor.
- A terminal that allows you to run Ruby programs.

About the audience

This book is intended for self-taught programmers looking for a warm introduction to data structures, developers who might have used singly linked lists before but don't know how they work internally. It's not advanced, nor academic material. You won't find formal definitions nor optimal algorithms on this book. It's a curated selection of articles from my blog that explain basic principles in simple terms.

About the code

The code in this book was optimized for "readability" not for production use. You won’t see fancy optimizations, lots of guards, or exhaustive error checking.

The code in this book will help you to understand how data structures work, but it won’t provide drop-in replacements for Ruby’s core data structures.

About the code style

The code was structured to fit nicely small devices (Even in portrait mode). So, my radical approach was to indent the code using two spaces and split most one-liners into code blocks.

To see this coding style in action, let's say that we have to implement a method that iterates over a collection of usernames and prints a greeting message for each user.

As a big fan of one liners, I would normally write the code this way:

```
def greet users
  users.each { |user| puts "Hello, #{user.name}!" }
end
```

Unfortunately, the code in the previous section don't look well on small devices. So, I decided to use code blocks instead.

```
def greet users
  users.each do |user|
    puts "Hello, #{user.name}!"
  end
end
```

I did the same thing for long lines with postfix conditionals. Instead of writing the code this way:

```
def greet_by_email user
  return send_email_to(user.name, user.email) if user
end
```

I wrote it this way:

```
def greet_by_email user
  if user
    send_email_to user.name, user.email
  end
end
```

There are few exceptions here and there, but reading the code on your phone won't be a problem.

Getting the source code

The source code and its test suite are hosted at github. You can get it by cloning this repo.

About the author

Ale Miralles is a software developer who has been working with computers from the past half of his life. He works at software development company in Argentina, where he builds compilers, IDEs, and web-based solutions for investment banks and international trading companies. He is a polyglot programmer who works mostly in C, C#, Ruby, and JavaScript.

On his spare time, he likes to read, travel with his family, and learn new stuff.

He lives in Buenos Aires with his wife and his beautiful son.

Table of Contents

Singly Linked Lists

A singly linked list is a data structure that allows us to manage variable-sized collections of elements.

Unlike C style arrays, singly linked lists can grow or shrink accordingly to the number of objects they have to store. This property makes them a nice fit for those cases where the number of elements is unknown beforehand.

On a singly linked list each element is represented by a node that contains two attributes:

Name	Summary
data	Current node's value.
next	Pointer to the next node in the list.

The first node is the ***head*** of the list and the last one is the ***tail***.

To mark the end of the list, we have to set tail's next pointer to nil.

The following section is an ASCII representation of a singly linked list.

```
d = Node's data.
n = Pointer to the next node.
x = nil

[d|n] -> [d|n] -> [d|n] -> x
```

Note: In this book we are going to use the terms pointer and reference interchangeably. Keep in mind that at Ruby's VM level a pointer and a reference might not be the same thing.

Since nodes on singly linked list don't have back pointers, the only way to traverse them is from ***head to tail***.

Let's start by taking a look at the singly linked list interface.

Singly linked lists interface

The next two tables describe singly linked list attributes and methods. I have included some methods that are no present in most programming books because in practice they show up quite frequently. Like ***cat***, ***each***, et. al.

Attributes

Name	Summary
head	Pointer to the first node of the list.
tail	Pointer to the last node of the list.
length	Number of elements in the current list.

Methods

Name	Summary	Complexity
initialize	Initializes an empty list.	O(1)
insert(item)	Inserts a new item into the list.	O(1)
remove(item)	Removes an item from the list.	O(n)
cat(list)	Catenates a list to the current list.	O(1)
clear	Removes all items from the list.	O(n)
includes?(data)	Checks if a value is present on the list.	O(n)
find_first(&predicate)	Returns the first element that matches the predicate.	O(n)
each	Loops over the list yielding one element at a time.	O(n)
print	Prints the contents of the list.	O(n)

Implementations details

In this section we are going to focus on three core aspects of each method:

- What the method does and how it does it.
- Run-time complexity (Big O).
- Source code.

But before jumping into the code, let's talk a bit about ***Big 0 notation***.

Big O is a notation that allows us to describe the efficiency of an algorithm and predict how the variance in the size of the data affects the resources (usually time) that the algorithm requires to process that data. Since all the methods that we are going to work on in this chapter run in either O(1) or O(n) time, let's focus on those.

O(1) means that the algorithm runs in constant time independently of the size of the data that it has to process. On singly linked lists, operations like insert or cat run in constant time because it doesn't matter if the list has an element or a million, the time required to run those methods is always O(1).

O(n) means that the resources required to run an algorithm increase linearly with the size of the data that it has to process. Traversals on singly linked lists run in O(n) time, ***where "n" is the number of elements in the list***. If a list has one element, we need just one iteration. If it has a million, we need a million.

When we analyze the performance of an algorithm we usually refer to ***the worst case situation***. For instance, finding an element on a singly linked list is an O(n) operation because we have to account for those cases where the element we are looking for does not exist. In such a situation, we have to start from the head of the list visiting all of its elements until we get to its tail. Hence, O(n), where n is the number of elements in the list.

Now, if the element we are looking for happens to be the first element on the list, the actual run-time cost for ***that particular operation*** would be 1. But that's just good luck. The run-time complexity of the lookup operation remains at the slow pace of O(n).

Note: Unless stated otherwise, we usually don't care about the best or average case situations.

In futures chapters we will explore data structures that work well on large

datasets. As you might already figure out, the singly linked list is not one of them.

So, that's it for our brief introduction to Big O notation. Now it's time to crank some code. Let's dive in!

Initialize

This method is the list constructor, and it's responsible for its proper initialization. What this method does is set the head and tail pointers to nil, and the list's length to 0.

The time complexity of this method is O(1).

```
def initialize
  self.head   = nil
  self.tail   = nil
  self.length = 0
end
```

Insert

This method creates a new node and inserts the specified value into the list. If the list has no nodes, the new node becomes the head of the list; otherwise, we insert it at the tail.

Since the list keeps pointers to its head and its tail, the time complexity of this method is O(1).

```
def insert data
  node = Node.new data
  unless head
    self.head = node
  else
    self.tail.next = node
  end
```

```
  self.tail = node
  self.length += 1
end
```

Remove

This method removes the given node from the linked list and adjusts pointers to keep the elements together.

This method is probably the most complex method on singly linked lists because depending on the location of the target node and the number of elements in the current list we have to cover three different situations.

- The target node is the only element in the list: we have to set head and tail to nil to remove it.
- The target node is the head of the list: the node that is next to it becomes the new head of the list, and the target node goes out of scope and becomes eligible for garbage collection.
- The list has many elements and the target node is not the first one: we have to traverse the list to find the target node and the node that precedes it. Once we have both nodes, we set the next pointer of the node that precedes the target node to point to the node that is next to it. After doing that, the target node is effectively removed and ready to be garbage collected.

Note: Unlike in languages like C, in Ruby, we don't have to release allocated memory manually. When an object goes out of scope, and no references are pointing to it, the object becomes eligible for garbage collection; from there on is up to the garbage collector when to reclaim those allocated bytes.

Since we might have to walk the whole list to remove the node, the time complexity of this method is O(n).

```
def remove node
  return unless node
  return unless self.length > 0
```

```
  succeed = false
  if node == head
    if head.next.nil?
      self.head = self.tail = nil
    else
      self.head = self.head.next
    end
    succeed = true
  else
    tmp = self.head
    while tmp && tmp.next != node
      tmp = tmp.next
    end
    if tmp
      succeed = true
      tmp.next = node.next
    end
  end

  self.length -= 1 if succeed
end
```

Cat

This method joins the specified list to the current list.

The work required to concatenate the lists consists in two steps:

1. Point the tail's next pointer of the current list to the head node of the list that we want to append.
2. Combine the length's from both lists and update length of the current list.

The time complexity of this method is O(1).

```
def cat list
  return unless list
```

```
  return unless list.length > 0

  if self.tail
    self.tail.next = list.head
  else
    # the current list is empty.
    self.tail = list.head
  end

  self.length += list.length
end
```

Clear

This method removes all elements from the list.

Since we have to walk the whole list, the time complexity is O(n).

```
def clear
  while self.length > 0
    remove self.head
  end
end
```

Note: In Ruby, the previous operation could be rewritten to run in constant time by removing the loop and setting head and tail to nil. However, I think that a language agnostic approach is a better way to explore the data structure and its algorithms. Incidentally, that's the route I'll take in the following chapters.

Includes?

This method checks if a given value is present on the list.

To find out if the specified data belongs to the list, we have to walk the list from head to tail and check at every iteration if the value of the current node matches

the value we are looking for. If it does, the value belongs to the list; otherwise we keep moving until we find the value or run out of nodes.

Since we might have to walk the whole list, the time complexity for this method is O(n).

```
def includes? data
  return false unless self.length > 0

  current = self.head
  while current
    if current.data == data
      return true
    end
    current = current.next
  end
  false
end
```

Find First

This method allows us to get the first element of the list that satisfies a given predicate.

In this context, a predicate is a function that takes an element from the list as its only argument and returns a boolean value indicating whether the item satisfies the conditions expressed in that function or not.

The way the list finds the first element that matches the predicate is by walking its elements and applying the predicate to each one of them until the result is true or the list gets exhausted. Whatever comes first.

The time complexity of this method is O(n) because we might have to walk all of the elements in the list.

```
def find_first &predicate
  return unless block_given?
```

```
  return unless self.length > 0

  current = self.head
  while current
    if predicate.call(current)
      return current
    end
    current = current.next
  end
end
```

To see how this method works, let's say we have a list of integers and we want to find the first occurrence of the number 3.

The "rudimentary" way could be something like this:

```
def find_3
  e = nil
  current = list.head
  while current
    if current.data == 3
      e = current.data
      break
    end
    current = current.next
  end
  return e
end
```

By using the helper method find_first we get the same result in a Ruby idiomatic way.

```
def find_3
  list.find_first do |e|
    e.data == 3;
  end
```

```
end
```

Each

This method is a common Ruby idiom for objects that support enumeration. What it does is to yield items from the list, one at a time, until the list is exhausted.

If you are not familiar with the language, this might seems odd at first, but in Ruby, methods take arguments and a block of code. Typically, enumerators yield items to a given block of code (or lambda).

The time complexity to yield the next item is O(1). The time complexity to yield the whole list is O(n).

```
def each
  return unless block_given?
  return unless self.length > 0

  current = self.head
  while current
    yield current
    current = current.next
  end
end
```

Print

This method prints the contents of the current list.
(Incidentally, a good use-case to show how the each method works.)

The time complexity of this method is O(n).

```
def print
  if self.length == 0
    puts "empty"
```

```
  else
    self.each do |node|
      puts node.data
    end
  end
end
```

When to use singly linked lists

Singly linked lists are well suited to manage variable-sized collection of objects. So, whenever we have to handle collections of unknown sizes, singly linked lists will certainly do. But you might want to keep in mind that singly linked lists work well when:

- Sequences are small.
- Sequences are significant, but we don’t care about lookup times.
- You can sacrifice read performance for the sake of optimal inserts.
- You can afford to walk the list from head to tail, only.

To-do lists, shopping cart items, and append-only logs are good fits for singly linked lists; whereas carousels, windows managers and multi-million data sets are not.

Doubly Linked Lists

A doubly linked list is a member of the linked list family where each node is connected by two pointers. The main difference with singly linked lists is that in doubly linked traversals are bidirectional and removals run in constant time. These features are possible because each node has a pointer to the previous and next node in the list.

On a doubly linked list each element is represented by a node that contains three attributes:

Name	Summary
prev	A pointer to the previous node in the list.
data	Current node's value.
next	A pointer to the next node in the list.

As well as with singly linked lists, the first node is the head of the list, and the last one is the tail.

Doubly linked lists are delimited by the previous pointer of the first element and the next pointer of the last one. Both pointers must be set to nil.

The following section is an ASCII representation of a doubly linked list.

```
p = Pointer to the previous node.
d = Node's data.
n = Pointer to the next node.
x = nil

x <-[p|d|n] <-> [p|d|n] <-> [p|d|n] -> x
```

Doubly linked lists interface

Except for a few methods, doubly and singly linked list share the same interface.

Note: Aditions and overrides are highlighted in bold text.

Attributes

Name	Summary	Complexity
head	Pointer to the first node of the list.	O(1)
tail	Pointer to the last node of the list.	O(1)
length	Number of elements in the current list.	O(1)

Methods

Name	Summary	Complexity
initialize	Initializes an empty list.	O(1)
insert(item)	Inserts a new item into the list.	O(1)
remove(item)	Removes an item from the list.	O(1)
cat(list)	Catenates a list to the current list.	O(1)
clear	Removes all items from the list.	O(n)
includes?(data)	Checks if a value is present on the list.	O(n)
find_first(&predicate)	Returns the first element that matches the predicate.	O(n)
each	Loops over the list yielding one element at a time.	O(n)
print	Prints the contents of the list.	O(n)
find_last(&predicate)	Returns the last element that matches the predicate.	O(n)
reverse_each	Loops over the list backwards yielding one element at a time.	O(n)
reverse_print	Prints the content of the list backwards.	O(n)

Implementation details

To save us some time, I'm not going to discuss methods and attributes inherited from the singly linked list (you can see those in the previous chapter). The analysis focuses on doubly linked list exclusively.

Insert

This method creates a new node and inserts the specified value into the list. If the list has no nodes, the new node becomes the head of the list; otherwise, we have to append it at the tail of the list.

Since the list keeps pointers to its head and its tail, the complexity of this method is O(1).

Note that when the list contains elements we have to set the previous pointer of the node we are about to insert to point to the tail of the list.

```
def insert data
  node = Node.new data

  unless head
    self.head = node
  else
    node.prev = self.tail
    self.tail.next = node
  end

  self.tail = node
  self.length += 1
end
```

Remove

This method removes the given node from the linked list and adjusts pointers to keep the elements together.

As it happens with singly linked lists, we have to cover three mutually exclusive situations.

- The target node is the only element in the list: we have to set head and tail to nil to remove it.
- The target node is the head of the list: the node that is next to it

becomes the new head of the list. In this case, the target node goes out of scope and becomes eligible for garbage collection.
- The list has many elements, and the target node is not the first one: we have to set the next pointer of the node that precedes the target node to point to the node that is next to it; then set the previous pointer of the node that is next to the target node to point to the node that precedes it.

Once we complete those steps, the target node goes out of scope and becomes eligible for garbage collection.

```
def remove node
  return unless node
  return unless self.length > 0

  # This methods assumes that the target
  # node belongs to the list.
  if node == head
    if head.next.nil?
      self.head = self.tail = nil
    else
      self.head = self.head.next
    end
  else
    p = node.prev
    n = node.next
    p&.next = n
    n&.prev = p
  end

  self.length -= 1
end
```

In case you didn’t notice it, this time we didn’t have to walk the list to find the node that precedes the element that we want to remove. The time complexity of this method is O(1).

Cat

This method allows us two join two lists together and is divided into 4 steps:

1. Point the previous pointer of the head of the list we are appending to the tail of the current list.
2. Point the next pointer of the tail of the current list to the head of the list we are appending.
3. Set the tail of the current list to the tail of the list we are appending.
4. Adjust current list length.

The time complexity of this method is O(1).

```
def cat list
  return unless list
  return unless list.length > 0

  if self.tail
    list.head.prev = self.tail
    self.tail.next = list.head
    self.tail      = list.tail
  else
    self.head = list.head
    self.tail = list.tail
  end

  self.length += list.length
end
```

Find Last

This method allows us to get the last element of the list that satisfies a given predicate. (Or the first occurrence starting from the tail.)

The way we find the last element is by walking the list from tail to head applying the predicate to each element until the result of that operation is true or the list gets exhausted.

The time complexity of this method is O(n) because we might have to walk the

whole list.

```
def find_last &predicate
  return unless block_given?
  return unless self.length > 0

  current = self.tail
  while current
    if predicate.call(current)
      return current
    end
    current = current.prev
  end
end
```

To see this method in action, suppose we have a list of numbers, and we want to find the last occurrence of the number 3.

As well as we did with singly linked lists, let’s try the “rudimentary” way first:

```
def find_last_3
  e = nil
  current = list.tail
  while current
    if current.data == 3
      e = current.data
      break
    end
    current = current.prev
  end
  return e
end
```

Now, let's take a look at the Ruby “idiomatic” way:

```
def find_last_3
```

```
  list.find_last do |e|
    e.data == 3
  end
end
```

Reverse Each

This method traverses the list from tail to head yielding one item at a time until it reaches the end of the list.

Unsurprisingly, the implementation of this method is similar to what we did with singly linked lists; the difference is that in this case instead of starting from to the head of the list we start from the tail and move backward until the current node is nil.

The time complexity to yield the previous element is O(1).

The time complexity to yield the whole list is O(n).

```
def reverse_each
  return unless block_given?
  return unless self.length > 0

  current = self.tail
  while current
    yield current
    current = current.prev
  end
end
```

Reverse Print

This method prints the contents of the current list backward.

The time complexity of this method is O(n).

```
def reverse_print
  if self.length == 0
    puts "empty"
  else
    self.reverse_each do |node|
      puts node.data
    end
  end
end
```

When to use doubly linked lists

Doubly linked lists work great when you have to move back and forth over a sequence of elements without wraparound.

The frame manager of a media player where the user can move back and forth (frame by frame) from the beginning to the end could be a good fit for this data structure.

Also, last but not least, since removals are O(1), doubly linked lists could be a good choice for those cases where you have to handle lots of delete operations.

Circular Linked Lists

The first time I heard about the circular linked list was in an Anders Hejlsberg's (*) interview when the host of the show asked him to draw his favorite data structure.

After pulling a joke about the geeky request, the creator of C# came up with a sketch of linked list with no tail that allowed you to insert elements in constant time.

The trick he did catch my attention because, as far as I knew, the only way to insert elements in constant time was by keeping a pointer to the tail of list; otherwise, you have to find the list's tail first, and then add the new element. The searching step transforms the insertion into an O(n) operation.

Besides being one of Anders favorites data structures, a circular linked list is also another member of the linked list family whose distinguishing property is that its last node points back to the first one.

Circular linked lists are named after the circular pattern that shows up when we run traversals on them. Since this kind of lists have no tail, traversals wraparound at the last element and restart from the head of the list forming a circle.

Each element on a circular linked list is represented by a node that contains two attributes:

Name	Summary
data	Current node's value.
next	Pointer to the next node in the list.

In contrast with regular linked lists, circular linked lists have no "hard" boundaries. The last node on the list points back to the first one.

Note: Circular linked lists can be singly or doubly linked. For the sake of simplicity, in this chapter, we are going to work on the singly linked variant.

(*) Anders Hejlsberg is prominent language designer creator of Turbo Pascal, Delphi, C# and TypeScript.

Circular linked lists interface

The “C-stylish” interface of circular linked list diverges from what we have seen on previous chapters. Let's take a look at it.

Note: Additions and methods that significantly diverge from regular linked lists are highlighted in bold text.

Attributes

Name	Summary	Complexity
head	Pointer to the first node of the list.	O(1)
current	Pointer to the current node.	O(1)
length	Number of elements in the current list.	O(1)

Methods

Name	Summary	Complexity
initialize	Initializes an empty list.	O(1)
insert(item)	Inserts a new item into the list.	O(n)
insert_next(prev, item)	Inserts a new item next to the previous node. (Anders's trick)	O(1)
remove(item)	Removes and item from the list.	O(n)
remove_next(prev)	Removes the element that is next to the previous node.	O(1)
clear	Removes all items from the list.	O(n)
includes?(data)	Checks if a value is present on the list.	O(n)
find_first(&predicate)	Returns the first element that matches the predicate.	O(n)
move_next	Points the current node to the next node in the list.	O(1)

full_scan	Traverses the list starting from the head node.	O(n)
print	Print the contents of the current list.	O(n)

Implementation details

As opposed to what we did in the previous chapter, this time we are going to start from scratch.

Extending singly linked list would save us some work, but some methods from it make no sense on circular linked ones. For instance, what would happen if someone calls the each method to traverse the list from head to tail?

That is right. Stack overflow!

Since the circular linked list has no tail, the each method runs until the stack blows up. Not a nice way to retain users for our library.

So, to prevent unwanted behavior, we are going to build our data structure from the ground up.

Initialize

This method is the list constructor. The only thing it does is to create an empty list setting the head to nil and the list length to 0.

The time complexity of this method is O(1).

```
def initialize
  self.head   = nil
  self.length = 0
end
```

Insert

Creates a new node to insert a value into the list. If the list has no nodes, the new node becomes the head of the list. Otherwise, we have to append it at the end of

the list.

Since circular lists keep no pointers to their last node, the time complexity of this method is O(n).

```
def insert data
  if (self.length == 0)
    return insert_next(nil, data)
  end

  if (self.length == 1)
    return insert_next(self.head, data)
  end

  self.current = head

  i = 0;
  while ((i += 1) < self.length)
    move_next
  end
  return insert_next(self.current, data)
end
```

Insert Next

Since regular inserts in circular linked are O(n) operations, counting with method that allows us to insert items in constant time is a nice feature to have.

That is what insert_next does.

As long as we hold a pointer to the node that will precede the new node once inserted, we can insert elements in constant time.

(Usually, it'll be a pointer to the last node, but that's not required. We can insert new elements in any place we want.)

If the list has no nodes, the argument "previous node" is ignored, and the newly

created node becomes the head of the list. In this particular case, the next pointer of the head node points to itself.

If the list contains nodes already, we have to do two things:

Set the next pointer of the new element to point to the next pointer of the node that precedes it.

Set the next pointer of the previous node to point to the new node.

And finally, we have to adjust the list's length before return.

The time complexity of this method is O(1).

```
def insert_next prev, data
  new_node = Node.new data
  if self.length == 0
    self.head = new_node.next = new_node
  else
    new_node.next = prev.next
    prev.next = new_node
  end
  self.length += 1
end
```

Remove

Removes the given node from the linked list and adjusts pointers to keep the elements together.

If the node to be removed is not the of the list, this method has to find the node that precedes the node to be deleted and then call remove_next.

The time complexity of this method is O(n).

```
def remove node
  return nil unless node
```

```
  return nil unless self.length > 0

  if self.head == node
    return remove_next self.head.next
  end

  prev  = nil
  found = false
  # finds the node that precedes the
  # target node.
  full_scan do |nd|
    if (nd == node)
      found = true
      break
    end
    prev = nd
  end

  remove_next prev if found
end
```

Remove Next

This method allows us to remove elements in constant time as long as we have a pointer to the node that precedes the target node.

If the element the target node is the only element in the lits, we only need to point head to nil.

If the argument is nil, we have to set head to point to the node that is next to it.

If the element to remove is not the only element in the list, nor the previous node is nil, we have to:

1. “Capture” the next to the previous node.
2. Point the next pointer of the captured node to that follows the next pointer of the previous node.

3. Check if the node we captured before is the head of the list. If so, we have to point the head of the list to the node that is next to the captured node.

Since we have a pointer to the element that precedes the target element and a pointer to the element that follows it, the time complexity of this method is O(1).

```
def remove_next prev
  return nil unless self.length > 0

  unless prev
    self.head = self.head.next
  else
    if prev.next == prev
      self.head = nil
    else
      old = prev.next
      prev.next = prev.next&.next
      if (old == self.head)
        self.head = old.next
      end
    end
  end

  self.length -= 1
end
```

Clear

This method removes all elements from the list. Since we have to walk the whole list.

The time complexity of this method is O(n).

```
def clear
  while self.length > 0
```

```
    remove self.head
  end
end
```

Includes?

This method checks if a given value is present on the list.

To find out if the specified data belongs to the list, we have to walk the list and check at every iteration if the value of the current node matches the value we are looking for. If it does, the value belongs to the list, otherwise we keep moving until we find the value or get back to the head of the list.

Since we might have to walk the whole list, the time complexity for this method is O(n).

```
def includes? data
  return false unless self.length > 0

  self.current = self.head
  loop do
    if current.data == data
      return true
    end
    if move_next == self.head
      return false
    end
  end
end
```

Move Next

Points the current node to the node that follows it.

Since we already have a pointer to the current node, the time complexity of this method is O(1).

```
def move_next
  self.current = self.current&.next
end
```

Full Scan

It traverses all elements in the list without wrapping around.

This method starts from the head of the list and moves next (yielding one element at a time) until it gets back to the head node.

It works like each on regular linked list.

The time complexity of this method is O(n).

```
def full_scan
  return nil unless block_given?
  return nil unless self.length > 0

  self.current = self.head
  loop do
    yield self.current
    break if (move_next == self.head)
  end
end
```

Find First

This method allows us to get the first element of the list that satisfies a given predicate.

The way the list finds the first element that matches the predicate is by walking its elements and applying the predicate function to each one of them until one matches, or it gets back to its head.

The complexity of this method is O(n).

```
def find_first &predicate
  return nil unless block_given?
  return nil unless self.length > 0

  self.current = self.head
  loop do
    if predicate.call(self.current)
      return self.current
    end
    break if (move_next == self.head)
  end
end
```

Print

This method prints the contents of the current list. It’s a nice helper method and also a neat way to show full_scan in action.

The time complexity of this method is O(n).

```
def print
  if self.length == 0
    puts "empty"
  else
    self.full_scan do |item|
      puts item.data
    end
  end
end
```

When to use a circular linked list

A windows manager that allows users to cycle thru windows by pressing Cmd+Tab could use a circular linked list to store windows handles.

```
def on_key_press e
  if (e.Super && e.Tab)
    windows.move_next
  end
end
```

Note: If you need to support backward cycling (i.e., when users press Cmd + Shift + Tab), you can use doubly circular linked lists instead.

Another interesting aspect of the circular linked list is that inserts and deletes run in constant time making them a nice fit for those cases where you have to perform lots of insert/remove operations on a reasonably sized data set.

Queues

A queue is a special kind of singly linked list designed to efficiently store and retrieve elements in first in, first out basis (FIFO).

Queues return elements in the same order they were stored and are represented by nodes that contain two attributes:

Name	Summary
data	Current node's value.
next	Pointer to the next node in the queue.

Contrary to what happens with other list-based data structures; queues won’t allow us to insert/remove elements at random locations. All elements are inserted at the head of the queue and removed from its tail.

The following section is an ASCII representation of a queue.

```
[1|next] -> [2|next] -> X (nil)
```

Now let’s say we want to store a new element:

```
[1|next] -> [2|next ] -> [3|next] -> X (nil)
```

And now we want to remove one:

```
[2|next] -> [3|next] -> X (nil)
```

Notice how queues grow to their tail and shrink from their head.

Queues interface

To keep the semantics of this data structure I've removed helper methods like cat

and find_first. Although from time to time they might come handy, if we find ourselves using them, the chances are that the queue is not the right data structure for the problem at hand.

The following table describes the methods on the queue.

Methods

Name	Summary	Complexity
initialize	Initializes an empty queue.	O(1)
queue(item)	Inserts a new item into the queue.	O(1)
dequeue	Removes an item from the queue.	O(1)
peek	Returns the node that is at the head of the queue without removing it.	O(1)
size	Returns the number of elements on the queue.	O(1)
clear	Removes all elements from the queue.	O(n)
each	Loops over the queue yielding an element at a time.	O(n)
print	Print the contents of the queue.	O(n)

Implementation details

An important difference between queues and the rest of the list we have implemented so far is that on queues there are no public attributes. To keep this data structure safe, we use private variables to store the head, tail, and length of the queue. By doing that, we can guarantee the semantics of the public methods and prevent unwanted mutations.

Initialize

This method is the queue constructor. It creates an empty queue, sets the head and tail nodes to nil, and the length of the queue to 0.

The time complexity of this method is O(1).

```
def initialize
```

```
  @head   = nil
  @tail   = nil
  @length = 0
end
```

Enqueue

This method creates a new node to insert a value into the queue. If the queue has no nodes, the new node becomes the head of the queue; otherwise, we append it at the tail of the queue.

Since the queue keeps pointers to its head and its tail, the time complexity of this method is O(1).

```
def enqueue data
  node = Node.new data
  unless @head
    @head = node
  else
    @tail.next = node
  end
  @tail    = node
  @length += 1
end
```

Dequeue

This method removes an element from the queue. In contrast to what happens on regular linked lists, removals on queues are straightforward because they always happen at the head of the queue.

To remove an element from the queue, we point the head to the node that is next to it and set tail to nil if the queue contains just one element.

The time complexity of this method is O(1).

```
def dequeue
  return nil unless @length > 0
  @head = @head.next
  @tail = nil if @length == 1
  @length -= 1
end
```

Peek

This method returns the node that’s at the head of the queue without removing it, or nil if the queue is empty.

Since dequeue doesn’t return the target element, peek is the way to go if you have to do something with that element.

The time complexity of this method is O(1).

```
def peek
  @head
end
```

Size

This method returns the number of elements on the queue.

The time complexity of this method is O(1).

```
def size
  @length
end
```

Clear

This method dequeues all elements from the queue.

The time complexity of this method is O(n).

```
def clear
  while peek
    dequeue
  end
end
```

Each

This method walks the queue yielding one at a time until it reaches the last element.

Because of the semantics of queues, I’m not a big fan of having methods like this one on them, but I’ve to admit that from time to time they might be handy. (i.e., during debugging sessions.)

The time complexity to yield the next item in the queue is O(1). The time complexity to yield the whole queue is O(n).

```
def each
  return nil unless block_given?
  current = @head
  while current
    yield current
    current = current.next
  end
end
```

Print

This method prints the contents of the queue by walking its items.

The time complexity of this method is O(n).

```
def print
  if @length == 0
    puts "empty"
  else
    each do |node|
      puts node.data
    end
  end
end
```

When to use queues

We can use queues to implement components like task schedulers, IO buffers, printing spoolers, or interrupt handlers, to name a few.

Stacks

A stack is a special kind of singly linked list designed to efficiently store and retrieve elements in last in, first out basis (LIFO).

Elements on a stack are retrieved in the opposite order they were stored and are represented by a nodes that contain two attributes:

Name	Summary
data	Current node's value.
next	Pointer to the next node in the stack.

The first node is the head of the stack, and the last one is its tail. To mark the end of the stack, we must set the tail's next pointer to nil.

As well as it happens with queues, stacks won’t allow us to insert/remove elements at random locations.

Let’s start by seeing the way a stack grows as we store (push) elements into it.

```
push(1) push(2) push(3)
 ---     ---     ---
  1       2       3
 ---     ---     ---
         1       2
         ---     ---
                 1
                 ---
```

Now let’s take a look at the way it shrinks when we remove (pop) items out of it.

```
pop  pop  pop
---  ---  ---
 3    2    1
```

```
---   ---   ---
 2     1
---   ---
 1
---
```

Stacks interface

As well as we did with queues, to keep the semantics pure we are not going to add helper methods to this data structure.

Methods

Name	Summary	Complexity
initialize	Initializes an empty stack.	O(1)
push(item)	Inserts a new item into the stack.	O(1)
pop	Removes an item from the top of the stack.	O(1)
peek	Returns the node that is at the top of the stack without removing it.	O(1)
size	Returns the number of elements on the stack.	O(1)
clear	Removes all elements from the stack.	O(n)
print	Print the contents of the stack.	O(n)

Implementation details

To implement the stack, we are going to apply the sample principles we use will building the queue. Don't expose attributes to guarantee the semantics of the public methods.

Initialize

This method is the stack constructor. It creates an empty stack, sets the head and the tail node to nil, and the length of the stack to 0.

The time complexity of this method is O(1).

```
def initialize
  @head   = nil
  @tail   = nil
  @length = 0
end
```

Push

This method creates a new node to insert a value into the stack. The new node moves the element that’s at the head of the list and becomes the new head of the list.

Since the stack keeps pointers to its head and its tail, the time complexity of this method is O(1).

```
def push data
  node = Node.new data
  if @length == 0
    @tail = node
  end
  node.next = @head
  @head = node
  @length += 1
end
```

Pop

This method removes an element from the stack. As it happens with queues, removals are straightforward because they always happen at the head of the stack.

To pop an element from the stack, we point the head to the node that is next to it and set tail to nil if the stack contains just one element.

The time complexity of this method is O(1).

```
def pop
  return nil unless @length > 0

  @head = @head.next
  @tail = nil if @length == 1
  @length -= 1
end
```

Peek

This method returns the node that’s at the head of the stack without removing it, or nil if the stack is empty.

Since pop doesn’t return the removed element, peek is the way to go if you have to do something with that element.

The time complexity of this method is O(1).

```
def peek
  @head
end
```

Size

This method returns the number of elements on the stack.

The time complexity of this method is O(1).

```
def size
  @length
end
```

Clear

This method pops all elements from the stack.

The time complexity of this method is O(n).

```
def clear
  while peek
    pop
  end
end
```

Print

This method prints the contents of the stack by walking its items.

The time complexity of this method is O(n).

```
def print
  if @length == 0
    puts "empty"
  else
    current = @head
    while current
      puts current.data
      current = current.next
    end
  end
end
```

When to use stacks

We can use stacks to implement Undo/Redo functionality, to manage activation frames, to parse expressions, and solve operators precedence, to name a few.

Ruby’s virtual machine (YARV) use stacks to store and eval bytecode instructions when we run our ruby programs.

Hash Table

A hash table is a data structure optimized for random reads where entries are stored as key-value pairs into an internal array. Since we access elements by hashing their key to a particular position within the internal array, (assuming approximation to uniform hashing) searches run in constant time.

If you have to remember just one thing about hash tables, remember that “a hash table maps a key to an index of an array.”. By doing that you’ll be able to “guestimate” the complexity of operations based on how well an array would do. For instance, insert, update, read, and delete, are all O(1) operations on arrays; unsurprisingly, that’s also true for hash tables.

Before jumping into the code, let's gloss over some theoretical aspects that will help us to understand why tables hash tables behave the way they do.

In practice, there are two kinds of hash tables:

Chained Hash Tables

- Dynamic size. Can grow to accommodate more data.
- Data is stored in buckets.
- Solves collisions by putting colliding keys into the same bucket.

Open-Addressed Hash Tables

- Fixed size.
- Data is stored in the table.
- Solves collisions by probing the table.
- Offers better data locality (This property depends on the probing method.)

Since at the time of this writing the Ruby core team is rewriting Ruby's hash table implementation using an open-addressed approach we are going to follow that route, too. Keep in mind, though, that in most cases, any approach would do

just fine.

Hash Code

A hash code is a result to apply a hash function to a given key. While keys could be of any type, hash codes are always integers.

An important aspect to remember about hash codes is that they can collide. While proper hash functions are designed to produce as few collisions as possible, in practice they still happen. It’s entirely possible for a hash function to produce the same hash code for two different keys. It doesn’t matter how strong the function is; we should always account for collisions.

Probing methods

A proving method is an efficient way to find an empty slot for a given key in an open-addressed table. That’s it.

These are the most common probing methods and their distinguishing properties:

Linear Probe

- Good cache performance.
- Easy to implement.
- Suffers from primary clustering.

Quadratic Probe

- Average Cache performance.
- Less clustering that linear probe.

Double hashing

- Poor cache performance.
- Avoids clustering.
- Need more CPU cycles.

Since double hashing is one of the most effective approaches (and also the one I'll use in this chapter), let's focus on that one.

Here is the definition for a double hashing function:

```
h(k,i) = (h1(k) + i * h2(k)) mod m
```

Variable	Description
m	Number of positions in the hash table.
k	The key we want to hash.
i	An integer that goes from 0 to m-1.
h1	First auxiliary hash function.
h2	Second auxiliary hash function.

Considerations on hash functions and the number slots

To ensure that we visit all positions before visiting any position twice, we need to constrain the number of slots to be a prime number and design the secondary hash function to return a positive number less than the number of slots.

Another approach is to choose a number of spots that is a power of 2 and design the secondary hash function to return an odd number.

A typical auxiliary hash function for double hashing might look like this:

```
k = key
m = slots
let h1(k) = k mod m
let h2(k) = 1 + (k mod (m-1))
```

Load factor

The load factor of a hash table is the number of occupied positions relative to the whole number of slots.

```
n  = entries
m  = slots
factor = n / m
```

Since open-addressed hash tables can’t contain more elements than slots, their load factor is less than or equal to 1. (The load factor for a full occupied table is 1.)

Keeping a healthy (below the average) load factor is crucial for open-addressed hash tables because as the table becomes full, the numbers of positions we have to probe increases drastically and make search operations costly and slow.

Load Factor (%)	Expected Probes	Number of traversals
50	1 / (1 - 0.5)	2
80	1 / (1 - 0.8)	5
90	1 / (1 - 0.9)	10
95	1 / (1 - 0.95)	20

As we see in the table above, once the table’s load factor exceeds 80%, the number of positions we have to probe sends lookups performance to the dumpster.

A trick we can pull to keep a healthy load factor is to allocate twice as many positions as we expect to use. (Assuming we can sacrifice memory footprint for the sake of speed, doubling the number of slots is a good way to go.)

Now that we have reviewed the theoretical aspect of hash tables is finally time to jump into the code!

Hash Tables interface

The public interface of this data structure is dead simple; it only has an attribute and five methods:

Attributes

Name	Summary
size	Number of entries in the hash table.

Methods

Name	Summary	Complexity
initialize	Creates an empty hash table.	O(m)
upsert(key, val)	Inserts or updates an entry.	O(1)
get(key)	Returns the value of the given key or nil if the key does not exists.	O(1)
delete(key)	Removes an entry from the hash table.	O(1)
print	Print the contents of the hash table.	O(m)

Implementation details

While on from the surface hash tables are straightforward, implementation details are quite complicated to say the least. Let’s start by taking a look at how entries are represented in the hash table.

Slot (hash table entry)

Attribute	Summary
key	Entry's unique identifier.
value	Entry's value.
vacated	A flag that indicates if the slot is free or not.

It’s important to note that we keep the original key for each table entry.

Now you might wonder: If we store entries by hashing their keys, why do we need to keep the original key?

The answer: Collisions!

While proper hash functions are designed to produce as few collisions as possible, in practice they still happen. So, when a collision occurs, the only way

to “tiebreak” the hashcode is by looking at the actual key, and that is why we have to keep them.

Another reason for what we need the original key is for those cases where we need to rebuild the table.

So, now that we know how hash table entries look like let’s take a look at the hash table methods.

Initialize

This method is the hash table constructor and the one in charge of proper initialization.

These are the steps the constructor executes when we create an instance of a hash table:

- Set the initial number of slots to 5.
- Initialize all of the slots with nil values.
- Set size (occupied positions) to 0.
- Set the number of rebuilds to zero. (If the table grows past certain threshold we have to rebuild it.)
- Set up auxiliary functions, h1 and h2.

Since we have to initialize all of the slots in the internal storage, the time complexity of this method is O(m).

```
def initialize
  @slots    = 5
  fill_table @slots
  self.size = 0
  @rebuilds = 0
  @h1 = -> (k) { k % @slots }
  @h2 = -> (k) { 1 + (k % (@slots - 1)) }
end
```

Note: If you are not familiar with Ruby’s syntax the code we use to

Upsert

This method inserts a new entry into the table or updates an existing one. Probably, this is the most complicated method in this data structure because it does many things. (I know that this sounds bad, but believe me, in this case, is not.) Let's analyze the code step by step.

The first thing we need to do is to check if there is already an entry for the given key. If there is one, we have to update the value and return.

If the entry does not exist, we have to check if there is enough room to add it to the table. If the number of slots is too small, we have to trigger a rebuild operation to ensure that the new entry fits into the internal storage. (Remember that in open-addressed hash tables the number of entries is fixed. When we run out of space we have to rebuild the table to add more room.)

Once we are sure that there is enough space, we have to hash the key and map it to a position into the internal storage. In our case, we do that by double hashing the key's hash code until we find an empty slot or run out of positions. If the latter happens, the combination of hash functions we choose was to way too weak.

Notice that we use the key's hash code and not the key itself. We do this because (internally) we need integer keys and Ruby's hash method is a reliable way to get them. It doesn't matter if users of this data structure use symbols or strings key, by using the hashcode we get the kind of values that we need.

Once we successfully mapped a key to a position in the internal array, we increase the table size and return.

Assuming proper hash functions and proper distribution, the time complexity of this method is O(1).

```
def upsert key, value
  if (slot = find_slot(key))
    slot.value = value
    return
  end
```

```
  rebuild if self.size > (@slots / 2)

  0.upto(@slots - 1) do |i|
    index = double_hash key.hash, i
    slot  = @table[index]
    if slot.nil? || slot.vacated
      @table[index] = Slot.new key, value
      self.size += 1
      return
    end
  end
  raise "Weak hash function."
end
```

Get

This method returns the value of on entry or nil if the entry doesn’t exist.

This method also applies double hashing to map a key to an index into the internal array. If there is an entry at the computed index, this method also checks the original key before return the entry’s value. (Remember that it’s possible for a hash function to produce the same code for two different keys. Unlikely, but entirely possible.)

As it happens with upsert, assuming proper hash functions and proper distribution, the time complexity of this method is O(1).

```
def get key
  0.upto(@slots - 1) do |i|
    index = double_hash key.hash, i
    slot  = @table[index]
    if slot.nil? || slot.vacated
      return nil
    end
    if slot.key == key
```

```
      return slot.value
    end
  end
  nil
end
```

Delete

This method searches a slot that matches the given key and marks it as free if it finds it.

As it happens with upsert and get, the time complexity of this method is O(1).

```
def delete key
  find_slot(key)&.free
end
```

Find Slot

This method's almost identical to get but instead of returning the entry's value returns the entry itself.

While is not part of the public API, this method plays a critical role on this hash table because upsert and delete heavily depend on it.

The time complexity of this method is O(1).

```
def find_slot key
  0.upto(@slots - 1) do |i|
    index = double_hash key.hash, i
    slot  = @table[index]
    return nil  if slot.nil?
    return slot if slot.key == key
  end
  nil
end
```

Print

This method prints the contents of the hash table. (Mostly for debugging purposes.)

The time complexity of this method is O(m).

```
def print
  @table.each do |e|
    if e
      puts "#{e.key}: #{e.value}"
    else
      puts "empty"
    end
  end
end
```

Rebuild

This method rebuilds the table's internal storage to make room for new entries.

In the code above the constant **PRIMES** contains appropriate prime numbers of slots to use when doing incremental rebuilds.

(On production grade hashtables the maximum number of entries should be up to a couple of millions, but for demonstration purposes, I guess 509 is a reasonable limit.)

The time complexity of this method is O(m).

```
def rebuild
  if @rebuilds >= MAX_REBUILDS
    raise "Too many entries."
  end
```

```
  old    = @table
  @slots = PRIMES[@rebuilds]
  self.size = 0
  fill_table @slots
  old.each do |e|
    upsert e.key, e.value if e
  end
  @rebuilds += 1
end

def fill_table slots
  @table = []
  0.upto(slots - 1) { @table << nil }
end
```

When to use hash tables

Hash tables work great when we can identify elements by unique keys, and we don’t need to traverse all entries. Caches, indexes, and routing tables are all good use-cases for hash tables.

Limitations

Since elements might be spread all over the place, hash tables should be avoided in situations where we need sequential reads (i.e., cursors.)

Note: The code in the apendix was adapted to reuse find_slot on get operations. The rest of the code stayed the same.

Sets

A set is an unordered sequence of unique elements called members, grouped because they related to each other in some way.

We can define sets by using sets notation:

```
S = { 1, 2, 3 }
```

A common way to describe sets is by their operations and properties. So let's take a look at those.

Set Operations

Empty

A set with no elements is the empty set.

```
{} => true
```

Equal

Two sets are equal if they contain the same members. (Remeber that in the case of sets the order doesn’t matter.)

```
{1, 2, 3} = {3, 2, 1} => true
```

Subset

Set “a” is a subset of “b” if “b” contains all of the elements that are present in “a”.

```
{3, 4} C {2, 3, 4} => true
```

Union

A union of sets “a” and “b” produces a set “c” that contains all the elements from “a” and “b”.

```
{1, 2, 3} U {4, 5, 6} => { 1, 2, 3, 4, 5, 6 }
```

Intersect

An intersection of two sets produces a new set that contains only the elements that are present in both sets.

```
{1, 2, 3} N {2, 3, 4} => { 2, 3 }
```

Difference

A difference between two sets is a set that contains all the elements for the first set except the ones that are present in the second.

```
{1, 2, 3} - {2, 3, 4} => { 1 }
```

Set Properties

- The intersection of a set with the empty set is the empty set.
- The union of set the “a” and the empty set is the set “a”.
- The intersection of a set with itself is the original set.
- The union of a set with itself is the original set.

Note: There are a few more properties to sets, but they are outside the scope of this book.

Sets are widely used in math, probability, and combinatorics. All subjects

equally interesting and way out of my league. So, let's focus on the data structure instead.

Sets interface

The following table shows us that the time complexity of operations significantly diverts from what ‘ve seen in previous chapters. The main reason behind that difference is that most operations involve two sets O(mn).

Name	Summary	Complexity
initialize	Initializes an empty set.	O(1)
insert(member)	Inserts a member into the set.	O(n)
remove(member)	Removes a member from the set.	O(n)
union(other)	Returns a set that contains all members from the current and the other set.	O(mn)
intersect(other)	Return a set that contains all elements from current set that are also members of the other set.	O(mn)
diff(other)	Returns a set that contains all members of the current set that are not present in the other set.	O(mn)
contains? (member)	Return true is the current set contains the given member.	O(n)
subset?(other)	returns true if the other set contains all of the members in the current set.	O(mn)
equal?(other)	Returns true the current set and the other set contains the same members.	O(mn)
count	Returns the numbers of members in the current set.	O(1)
each(block)	Loops over all of the members of the current set applying the given predicate.	O(n)
print	Prints the contents of the current set.	O(n)

Implementation Details

To save us some work we are going to store set's members on an instance of the

singly linked list we built in the first chapter.

Note: We could also have used hash tables which would give us better insert/remove performance at a cost a bit slower full scans. As most things we do in our field is all about trade-offs.

Initialize

The only thing this method does is to initialize the set's internal storage. The time complexity of this method is O(1).

```
def initialize
  @list = SinglyLinkedList.new
end
```

Insert

This method inserts a new member into the set. Since sets can't contain duplicates, the first thing we have to do is to check if the member already exists. If it does, we do nothing; otherwise, we have to insert the new member into the set.

Since we have to check if the member is already there, the time complexity of this method is O(n)

```
def insert member
  return if contains? member
  @list.insert member
end
```

Remove

This method removes a member from the current set, or it does nothing if the element doesn't exist.

Since we might have to walk the whole list the time complexity of this method is

O(n).

```
def remove member
  node = @list.find_first do |nd|
    nd.data == member
  end
  @list.remove node if node
end
```

Union

This method returns a set that contains all members of the current and the other set.

Since this operation involves walking two sets its time complexity is O(mn).

```
def union other
  res = Set.new
  @list.each do |nd|
    res.insert(nd.data)
  end
  other.each do |data|
    res.insert(data)
  end
  res
end
```

Intersect

This method returns the intersection of the current set with the other set.

Since this operation involves walking two sets, its time complexity is O(mn).

```
def intersect other
  res = Set.new
```

```
  @list.each do |nd|
    if other.contains?(nd.data)
      res.insert(nd.data)
    end
  end
  res
end
```

Diff

This method returns a set that contains all of the elements that are present in the current set but not in the other.

As well as it happens with union an intersect, this method requires walking the two sets, its time complexity is O(mn).

```
def diff other
  res = Set.new
  @list.each do |nd|
    unless other.contains?(nd.data)
      res.insert(nd.data)
    end
  end
  res
end
```

Contains

This method returns true if the set includes the given member. To find a member this method walks the internal list and check each node data until it finds the member or the list runs out of nodes.

To complete this operation we might have to walk the whole set. Therefore its time complexity is O(n).

```
def contains? member
```

```
  @list.find_first do |nd|
    nd.data == member
  end
end
```

Subset

This method returns true if the current set is a subset of the other set and false otherwise.

In this case, we use two strategies: If the current set is larger than the other set, there is no way that if a subset of it. So we can take the expressway an exit earlier. The result is false.

If the previous condition is not true, we have to go the slow path instead and walk over all members in the current set.

Potentially, we might have to walk two sets; therefore the time complexity of this method is O(mn).

```
def subset? other
  if self.count > other.count
    return false
  end
  @list.each do |nd|
    unless other.contains?(nd.data)
      return false
    end
  end
  true
end
```

Equal

This method returns true if the current set is equal to the other set and false otherwise. As well as we did with the subset method, we do a quick check on

set’s length first to see if we can exit fast if that check fails, we have to go the slow path instead.

As it happens with the subset method, we might have to walk two sets to complete this operation which makes its time complexity O(mn).

```
def equal? other
  if self.count != other.count
    return false
  end
  subset? other
end
```

Count

This method returns the number of elements in the current set. The time complexity of this method is O(1).

```
def count
  @list.length
end
```

Each

This method walks the members in the current set yielding them one a time. The time complexity of this method is O(n).

```
def each
  return nil unless block_given?

  current = @list.head
  while current
    yield current&.data
    current = current.next
  end
```

```
end
```

Print

This method prints the contents of the current set. The time complexity of this method is O(n).

```
def print
  @list.print
end
```

When to use sets

Sets are commonly used to find relationships between different groups of data.

Also, as we will see in when we get to the advanced data structures section, sets are widely use as a support data structure for graphs and graphs algorithms.

Binary Trees

A tree is a data structure that allows us to represent different forms of hierarchical data. The DOM in HTML pages, files and folders in our disc, and the internal representation of Ruby programs are all different forms of data that can be represented using trees. In this chapter we are going to focus on ***binary trees***. A particular kind of three that has only two branches.

The way we usually classify trees is by their branching factor, a number that describes how nodes multiply as we add more levels to the tree.

Unsurprisingly, the branching factor of binary trees is two.

The following section is as ASCII representation of a binary tree.

```
     3
    / \
   8   17
  /\   /\
 1 11 15 25
```

Notice that we have one node at the root level, then two nodes one level below, then four nodes two levels below, and so on…

Note: The tree in the previous section is a balanced binary tree. We will explore that concept in future chapters, but for now, let's say a tree is balanced when its nodes are arranged in such a way that tree is as short as it could be.

Since many search algorithms depend on the height of the trees, keep them short is crucial for performance reasons.

Each entry on the tree is represented by a node that has four attributes:

Name	Description
parent	Parent node of the current node.

data	Current node's value.
left	Current node's left subtree.
right	Current node's right subtree.

Every node has a parent and up to two children. The only exception to this rule is the first node (called root) which has no parent.

As it happens with other hierarchical data structures, there are different ways to traverse trees. ***Depth-first*** and ***breadth-first*** are the most common ones:

Binary Trees Interface

The interface of a binary tree is tightly coupled with the way we are going to use it. For instance, an AVL Tree (binary search) have methods to rotate and balance subtrees whereas a regular one has not.

In this chapter, we are going to build a general purpose binary tree that we could use to store in-memory representations of binary expressions like 1 + 2 * 3 and a=b.

The following section contains an ASCII representation of the same expressions in its tree form:

```
  +           =
 / \         / \
1   *       a   b
   / \
  2   3
```

Name	Summary	Complexity
initialize()	Initializes an empty tree.	O(1)
insert_left(node, data)	Inserts a new node at the left child of the specified node.	O(1)
insert_right(node, data)	Inserts a new node at the right child of the specified node.	O(1)
	Removes the subtree rooted at the left child of	

remove_left(node)	the specified node.	O(n)
remove_right(node)	Removes the subtree rooted at the right child of the specified node.	O(n)
merge(left, right)	Produces a new tree by merging left and right.	O(1)
print	Prints the contents of the current tree.	O(n)

Implementation Details

Initialize

This sets the root node of the tree an initializes its size.

The time complexity of this method is O(1).

```
def initialize root
  @root = root
  @size = 1
end
```

Insert Left

This method inserts a new node rooted at the left child of the specified node.

The time complexity of this method is O(1).

Note that this method adds a check to prevent mutations to the left subtree. While it’s application-specific, when we change a node we usually have to rearrange the structure of the tree.

```
def insert_left(node, data)
  return unless node
  if node.left
    raise "Can't override nodes."
  end
  @size += 1
```

```
  node.left = Node.new node, data
end
```

Insert Right

This method works like insert_left but uses the right child instead. The time complexity of this method is O(1).

```
def insert_right(node, data)
  return unless node
  if node.right
    raise "Can't override nodes."
  end
  @size += 1
  node.right = Node.new node, data
end
```

Remove Left

This method removes the subtree that’s rooted at the left child of the given node.

Notice that as opposed to what happens with languages like C, in Ruby, the only thing we need to do to remove the subtree is set it to nil. From there, it’s up to the garbage collector to release allocated nodes.

The time complexity of this method is O(n) where n is equal to the number of nodes in the subtree.

```
def remove_left(node)
  if node&.left
    remove_left(node.left)
    remove_right(node.left)
    node.left = nil
    @size -= 1
  end
end
```

Remove Right

This method removes the subtree that’s rooted at the left child of the given node.

As well as it happens with remove_left, the time complexity of this method is O(n).

```
def remove_right(node)
  if node&.right
    remove_left node.right
    remove_right node.right
    node.right = nil
    @size -= 1
  end
end
```

Merge

This is a class method that creates a new binary tree by merging two existing ones.

Optionally, this method allows us to specify a value for the root node of the merged tree.

These are the steps to merge the two trees:

1. Create a root node for the merged tree.
2. Create an empty tree to hold the merge.
3. Take the merged tree and point the left child of its root node to the root node of the left tree.
4. Take the merged tree and point the right child of its root node to the root node of the right tree.
5. Adjust the size in the merged tree.

The time complexity of this method is O(1).

```
def self.merge tree1, tree2, data = nil
  unless tree1 && tree2
    raise "Can't merge nil trees."
  end

  root = Node.new(nil, data);
  res  = BinaryTree.new root

  res.root.left  = tree1.root
  res.root.right = tree2.root

  res.size = tree1.size + tree2.size
  res
end
```

Print

This method prints the contents of the current tree recursively applying a “pretty” format (actually, it’s just indentation, but still…)

The time complexity of this method is O(n).

```
def print
  print_rec @root
end

private
def print_rec node, indent=0
  print_node node, indent
  if node&.left
    print_rec node.left, indent + 1
  end
  if node&.right
    print_rec node.right, indent + 1
  end
end
```

```
def print_node node, indent
  data = node&.data&.to_s || "nil"
  puts data.rjust indent * 4, " "
end
```

When to use trees

Trees are one of the most used data structure in CS. They can be found in the source code of database systems, user interfaces, data compression algorithms, parsers, and schedules, to name a few. They are pretty much everywhere.

Binary Search Trees

Binary search tree, or BTS for short, is a variation of a binary tree data structure designed to perform fast lookups on large datasets. The subject of this chapter is AVL trees, a special kind of self-balancing BST named after its creators Adelson-Velskii and Landis.

On AVL trees nodes arranged in descendant order, and the height difference between the left and right subtrees (the balance factor) is always -1, 0 or 1.

As it happens with sets, AVL trees can't contain duplicate keys.

Each node on the AVL Tree contains these attributes:

Name	Summary
key	Node's unique identifier.
data	Node's value.
deleted	This flag indicates if the node has been "removed".
height	Node's height.
left	Node's left subtree.
right	Node's right subtree

Let’s start by looking at how lookups work.

How lookups work

On this data structure lookups are top-down recursive-descendant operations; which means that they start from the root node and move all the way down until they find the key they are looking for or run out of nodes.

Of course, inspecting the whole tree without a strategy would be prohibitively expensive, but since nodes are guaranteed to be sorted, we don’t have to do that. We start from the top, descending left or right based on how our key compares to the current node’s.

If the key of the current node is higher than the key we are looking for, we move to the left; otherwise, we go to the right. In any case, we have to repeat this operation until we find the key we are looking for, or we get to the end of the branch (EOB).

The following section is an ASCII representation of an AVL Tree.

```
      4
     / \
    /   \
   2     6
  / \   / \
1   3 5   9
          /\
         7  11
```

Now suppose we want to find the number 5.

Starting from the root node, the way to go would be:

```
1. Is 5 == 4?
No

2. Is 5 < 4?
No. Move right.

3. Is 5 == 6?
No

4. Is 5 < 6?
Yes. Move left.

5. Is 5 == 5
Yes. We found it!
```

Note: To simplify the lookup process I've omitted the check for "end of the

branch.”

So, to find the number 5 on an AVL tree we need only 2 traversals from the root; the same operation on a singly linked list would require 5.

For a small number of elements, it doesn’t look like a big difference; however, what is interesting about AVL trees, is that lookup operations run O(log2 n) time; which means that they get exponentially better than linear search as the dataset grows.

For instance, looking for an element out of a million on a singly linked list could require up to 1,000,000 traversals. The same operation on AVL trees would require roughly 20!

How inserts work

Inserts and searches are tightly coupled operations. So, now that we know how searches work let’s take a look at inserts. Or better yet, rotations, which are the interesting bits of inserts.

Rotation is a mechanism to rearrange parts of a tree to restore its balance without breaking these properties:

- left’s key < parent’s key
- parent’s key < right’s key

Once the rotation is complete, the balance factor of all nodes must be in the balanced range (-1..1).

Depending on the location of the node that puts the tree into an unbalanced state, we have to apply one of these rotations (LL, LR, RR, RL). Let’s take a look at each one of them.

LL (left-left)

Suppose we want to add the number 1 to the following subtree:

```
              /
```

```
        6
       / \
      3   7
     / \
    2   4
   /
  x
```

By following the rules we use for lookups operations, we descend from top to bottom, moving left or right, until we find an empty spot. In this case, that spot is the left subtree of the node that contains the number 2. Once we are there, we insert the new node.

```
        /
       6
      / \
     3   7
    / \
   2   4
  /
 1
```

So far, so good; but now we have a problem. After inserting the new node, the balance factor of the subtree's root node (6) went to +2, which means that we must balance the tree before going on.

To know which kind of rotation we have to apply, we can follow the path from the unbalanced node to the node we just added. Let's annotate the tree to visualize this path:

```
          /
         6
        / \
  (L) 3   7
     / \
(L) 2   4
```

```
      /
     1
```

In this case, the node is rooted at the left subtree of the left subtree of the unbalance node; hence, we have to do a left-left rotation (LL).

LR (left-right)

Now let’s take a look at a left-right rotation (LR). A rotation we have to apply when the new node lies on the right subtree of the left subtree of the node that got unbalanced.

In the following three that rotation will take place if we have to insert the number 5 into the tree.

```
        /
       6
      / \
     3   7
    / \
   2   4
        \
         5 (*)
```

RR (right-right)

The third kind of rotation is the right-right rotation (RR). This rotation happens when the new node is on the right subtree of the right subtree of the unbalanced node. For instance, if we insert the number 11 into the following tree.

```
      /
     4
    / \
   3   6
      / \
     5   9
```

```
            \
            11 (*)
```

RL (right-left)

The last kind of rotation is the right-left rotation (RL). A rotation that happens when the new node lies on the right subtree of the left subtree of the node that got unbalanced.

After inserting the number 4 in the following tree, we must perform an RL rotation to restore its balance.

```
      /
    2
   / \
  1   6
     / \
    5   9
   /
  4 (*)
```

Interface for AVL trees

The public API for this data structure is straightforward. It only has four methods.

Name	Summary	Complexity
insert(key, data)	Inserts a new node into the current tree.	O(log2 n)
remove(key)	Removes an entry from the current tree.	O(log2 n)
search(key)	Searches for an entry based on its key.	O(log2 n)
print	Print the contents of the current tree.	O(n)

Implementation details

As opposed to what we did in previous chapters, this time we are going to focus

the private methods of the data structure. By doing so, we can cover important details that otherwise would be unnoticed.

Also, we are not going to reuse code from the previous chapter. AVL Trees had a bunch of logic that would be hard to follow if we don't start from scratch.

Insert

Adds a new node to the tree and reset its root. This method relies on insert_and_balance that takes care of applying the right rotations to keep the tree in balance.

The time complexity of this method is O(log2 n).

```
def insert key, data = nil
  @root = insert_and_balance(@root, key, data)
end
```

Insert And Balance

This method looks recursively to find the right spot for the given key. Once it finds the spot and inserts the new node, it calls the balance method to ensure that the tree remains balanced.

The time complexity of this method is O(log2 n).

```
def insert_and_balance node, key, data = nil
  return Node.new key, data unless node

  if (key < node.key)
    node.left = insert_and_balance(node.left, key, dat
  elsif(key > node.key)
    node.right = insert_and_balance(node.right, key, d
  else
    node.data  = data
    node.deleted = false
```

```
  end
  balance(node)
end
```

Balance

This method balances the subtree rooted at the specified node. This method is probably the most complex in this data structure because it is responsible for selecting the right kind of rotation based on the “heaviness” of the tree.

A tree is considered left-heavy is its balance factor is higher than 1 or right-heavy if it's lower than -1.

The best way to follow this method is by looking at the ASCII representations of each use-case in previous sections.

The time complexity of this method is O(log2 n).

```
def balance n
  set_height n
  # *h* is a helper method that
  # calculates the height of the tree.
  if (h(n.left) - h(n.right) == 2)
    if (h(n.left.right) > h(n.left.left))
      return rotate_left_right(n)
    end
    return rotate_right(n)
  elsif (h(n.right) - h(n.left) == 2)
    if (h(n.right.left) > h(n.right.right))
      return rotate_right_left(n)
    end
    return rotate_left(n)
  end
  return n
end
```

Remove

This method finds the node to be removed and marks it as deleted. Performance wise, this is a neat way to handle deletions because the structure of the tree doesn’t change, and so we don’t have to balance it after removal.

This technique is called “lazy” removal because it doesn’t do much work.

The time complexity of this method is O(log2 n).

```
def remove key
  search(key)&.deleted = true
end
```

Search

This method starts a top-down recursive descendant search from the current tree’s root node.

The time complexity of this method is O(log2 n).

```
def search key
  node = search_rec @root, key
  return node unless node&.deleted
end
```

Search Rec

Searches for a key in the subtree that starts at the provided node. This method starts the search at a given node and descends (recursively) moving left or right based on the key’s values until it finds the key or gets to the end of the subtree.

The time complexity of this method is O(log2 n).

```
def search_rec node, key
  return nil unless node
```

```
  if (key < node.key)
    return search_rec(node.left, key)
  end

  if (key > node.key)
    return search_rec(node.right, key)
  end

  return node
end
```

Set height

This method calculates and sets the height for the specified node based on the heights of their left and right subtrees.

The time complexity of this method is O(1).

```
def set_height node
  lh  = h(node&.left)
  rh  = h(node&.right)
  max = lh > rh ? lh : rh
  node.height = 1 + max
end
```

Rotate right

This method performs a right rotation.

The time complexity of this method is O(1).

```
def rotate_right p
  q       = p.left
  p.left  = q.right
  q.right = p
```

```
  set_height p
  set_height q
  return q
end
```

Rotate Left

This method performs a left rotation.

The time complexity of this method is O(1).

```
def rotate_left p
  q       = p.right
  p.right = q.left
  q.left  = p
  set_height p
  set_height q
  return q
end
```

Rotate Left Right

This method points the left subtree of the given node to the result of rotating that subtree to the left and then rotates the specified node to the right. The last rotation is also its return value.

The time complexity of this method is O(1).

```
def rotate_left_right node
  node.left = rotate_left(node.left)
  return rotate_right(node)
end
```

Rotate Right Left

This method points the right subtree of the given node to the result of rotating that subtree to the right and then rotates the specified node to the left. The last rotation is also its return value.

The time complexity of this method is O(1).

```
def rotate_right_left node
  node.right = rotate_right(node.right)
  return rotate_left(node)
end
```

Print

This method prints the contents of a tree.

Since it has to visit all nodes in the tree, the time complexity of this method is O(n).

```
def print
  print_rec @root, 0
end

def print_rec node, indent
  unless node
    puts "x".rjust(indent * 4, " ")
    return
  end
  puts_key node, indent
  print_rec node.left, indent + 1
  print_rec node.right, indent + 1
end

def puts_key node, indent
  txt = node.key.to_s
  if node.deleted
    txt += " (D)"
```

```
    puts txt.rjust(indent * 8, " ")
  else
    puts txt.rjust(indent * 4, " ")
  end
end
```

When to use binary search trees

Binary search trees work well when we have to handle large datasets in applications where the read/write ratio is 10 > 1 or more. If the volume of data is small or the read/write ratio is even is better to use hash tables or even linked lists instead.

I recently used this data structure to implement the internal storage for a spreadsheet component, and it worked like a charm.

Graphs

A graph is a data structure that allows us to represent data in terms of objects and relationships.

Objects on a graph are called vertices (or nodes), and their connections to other vertices are called edges.

```
    A <-- (this is vertex/node.)
   / \
  B - C
   \ / <- (this is an edge.)
    D
```

A graph is said to be undirected where relationships between objects are bidirectional (like the one in the section above) or directed when they are not.

A directed graph is classified as cyclic when a node points back to the node that references it, and acyclic when there are no back-pointers. A compiler's code generators usually transform AST (abstract syntax trees) to DAGs (directed acyclic graphs) before emitting object code for the target platform.

The following section is an ASCII representation of a Directed Acyclic Graph.

```
     A
    / \
   /   \
  v     v
B ----> C
```

Interface for Graphs

The interface for general purposes graphs is quite simple. It's just a small set of operations that allows us to add, remove, connect, disconnect, and search for

vertices.

Depending on the specific use of the graph, you might have to add additional properties. For instance, you might have to add weight to the edges if you are trying to solve the shortest path from one node to the other.

Name	Summary	Complexity
initialize()	Initializes an empty graph.	O(1)
insert_vertex(key)	Adds a vertex (node) to the graph.	O(n)
insert_edge(key1, key2)	Connects two vertices by adding and edge to the graph.	O(n)
remove_vertex(key)	Removes a vertex from the graph.	O(n+e)
remove_edge(key1, key2)	Disconnects two vertices by removing the edge that connects them.	O(n)
adjacent?(key1, key2)	Tells if two vertices are adjacent or not.	O(n)
find_vertex(key)	Finds a vertex in the current graph.	O(n)
print	Prints the contents of the current graph.	O(n+e)

Implementation details

At its core, a graph is simply a collection of connected nodes. To store those nodes we are going to use a singly linked list where each entry on the list points to an instance of the Vertex class, the class we use to represent a node and its edges.

The singly linked list we are going to use this time is a slight variation of the one that we built in the first chapter that allows us to remove elements in constant time.

The following section is an ASCII representation of the data structure.

```
v = vertex
e = edges
b = next
x = nil
```

```
[v|{e}|n] -> [v|{e}|n] -> x
```

Each vertex (node) contains a key that uniquely identifies it and a set of edges attached to it.

To represent the edges on the graph, we use the set data structure that we built in a previous chapter.

Initialize

This method is the graph's constructor, and the only thing it does is initialize the list of vertices.

The time complexity of this method is O(1).

```
def initialize
  @vertices = SinglyLinkedList.new
end
```

Find Vertex

This method finds a vertex based on its key.

Since this method delegates the task to a singly linked list, the search ends up being linear, and so the time complexity is O(n).

```
def find_vertex key
  @vertices.find_first do |v|
    v.data.key == key
  end
end
```

Insert Vertex

This method adds a vertex to the graph.

Since this method has to guarantee that the graph contains no duplicates, its time complexity is O(n), where n is the number of vertices in the graph.

```
def insert_vertex key
  return if find_vertex key
  vertex = Vertex.new key
  @vertices.insert vertex
end
```

Insert Edge

This method connects two nodes by adding an edge to the graph.

The time complexity of this method is O(n), where n is the number of vertices in the graph.

```
def insert_edge key1, key2
  v1 = find_vertex key1
  return unless v1

  v2 = find_vertex key2
  return unless v2

  v1.data.edges.insert v2.data.key
end
```

Remove Vertex

This method removes a vertex from the graph.

Since this method is the most complex one, we are going to analyze it step by step.

The first thing we have to do is to look for the vertex that matches the key that we want to remove. This time we can’t use the standard find_vertex method because what we need is the node that contains the vertex on the vertices list and

also the one that precedes it. By having those nodes, we can remove the target vertex in constant time.

Once we found the target node, we have to make sure that it doesn’t contain references to other vertices. If the node contains references, we can’t remove it.

If we pass the previous check, the node can be safely removed; To do so, we have to grab the node that precedes the target node (captured while searching) and called the method remove_next on the modified version of the singly linked list that we use to store vertices.

The time complexity of this method is O(n + e), where n is the number of vertices in the graph and e is the number of edges.

```
def remove_vertex key
  found  = false
  target = nil
  prev   = nil

  @vertices.each do |v|
    if v.data.edges.contains? key
      return
    end
    if v.data.key == key
      found  = true
      target = v.data
    end
    prev = v unless found
  end

  unless found
    return
  end

  unless target.edges.count == 0
    return
  end
```

```
  @vertices.remove_next prev
end
```

Remove Edge

This method removes the edge that connects the specified nodes from the graph.

To remove the edge we search for the vertex that matches the first key, and if we found it, we remove the second key form its edges collection.

The time complexity of this method is O(n), where n is the number of vertices in the graph.

```
def remove_edge key1, key2
  vertex = find_vertex(key1)&.data
  return unless vertex
  vertex.edges.remove key2
end
```

Adjacent?

This method tells if two vertices (nodes) are adjacent or not.

As we did with remove_edge, we first search for the vertex that matches the first key, and if we found it, we check if its edges collection contains the second key, if it does, the nodes are adjacent.

The time complexity of this method is O(n), where n is the number of vertices in the graph.

```
def adjacent? key1, key2
  vertex = find_vertex(key1)&.data
  if vertex&.edges.contains? key2
    return true
  end
```

```
  return false
end
```

Print

This method prints the contents of the current graph.

The time complexity of this method is O(n + e), where n is the number of vertices in the graph and e is the number of edges.

```
def print
  @vertices.each do |v|
    puts "#{v.data} (vertex)"
    v.data.edges.each do |e|
      puts "    #{e.data} (edge)"
    end
  end
end
```

When to use graphs

Graphs can be used to solve all sort of problems:

- To get the shortest path between two points on a map.
- To count network hops.
- To do topological sorting
- To represent dependencies on software systems.
- To orchestrate code generation on compilers.
- To represent commits on source content management systems like git.

If you want to dig a bit deeper into graphs, the git's codebase is a nice place to see practical applications of graphs.

Persistent Lists

In this chapter I want to introduce the concept of persistent data structures, a term coined by Driscoll et al. in an article from 1986, that these days it's at the core of most purely functional data structures.

The first time I heard the term was in a talk by Rich Hickey, while the creator of Clojure was showing how functional programming languages deal with mutations; or the lack of them.

In this chapter, we are going to implement a persistently linked list, which is a data structure that behaves almost the same as a regular linked list but is immutable and has a copy on write semantics.

Let's start by reviewing how mutations work.

How mutations work

On imperative programming languages, when you want to modify a data structure, you do it in such a way that the current version of the data structure is destroyed and replaced by a new one.

If you have a linked list that contains one item, and you add another one, you end up with a list that contains two items. The previous version of the list (the one that contained one item) is no longer available. Hence, regular linked lists are not persistent.

Note: Incidentally, that is why this kind of updates are called "destructive assignments" .

```
> list = (1)
> list.insert(2)
> list.print
(1, 2)
```

On functional programming languages though, things are a bit different, because variables are immutable and updates in place are not even an option;

If you have a linked list that contains an item and you add another one, you end up with two lists; the original version that contains the sole item, and the new version that contains two. This kind of data structure is said to be persistent because, as long as you hold a reference to them, different versions of the same data structure can coexist in our programs.

The way persistence is achieved by making a copy of the original data structure and applying the changes to that copy rather than modifying the original version. A common way to perform this kind of updates is by using a technique called COW (copy on write); which as its name implies, consists in making a copy of the original object every time we are about to apply modifications to it.

```
> list1 = List.new(1)
> list2 = List.insert(list1, 2)
> list1.print
(1)

> list2.print
(1, 2)
```

Notice how as long as we hold a reference to them, both versions of the linked list can coexist in the same program.

A nice thing about persistent data structures is that we can share them on concurrent programs without worrying about locks or unwanted mutations. Since they are immutable, and changes to one version are not visible to the other, they can be shared safely between multiple threads and execution contexts.

If you think that having to copy the entire list over and over again might introduce performance penalties, you are right. However, you don’t have to worry about that, because there are a couple of techniques we can apply to reuse unaffected nodes and amortize the whole process.

Persistent list interface

The interface of this data structure is strongly shaped after the singly linked list that we built in the first chapter, but there is an important difference, in this interface, all methods are class methods. We cannot make calls on instances of this data structure; all interactions happen thru class methods.

Name	Summary	Complexity
empty()	Initializes an empty list.	O(n)
insert(list, data)	Inserts a new item into a copy of the specified list.	O(n)
update(list, node, data)	Updates a value into a copy of the specified list.	O(n)
remove(list, node)	Removes an item from a copy of the specified list.	O(n)
cat(list1, list2)	Creates a list that contains a copy of all nodes from list1 and list2.	O(n)
len(list)	Returns the number of elements on the specified list.	O(1)
find_first(list, &predicate)	Returns the first element that matches the predicate on the specified list.	O(n)
each(list, &block)	Loops over the specified list calling the given block passing one element at a time.	O(n)
print(list)	Prints the contents of the specified list.	O(n)

Notice how in contrast to what happens on singly linked lists, the time complexity for most methods is O(n). That’s because before doing the actual work on the list we have to make a copy of it.

Implementation details

To store values on a persistently linked list, we need read-only nodes. Since Ruby doesn't have immutable objects, we are going to emulate them by calling the freeze method on nodes once its data has been set. This solution is not perfect, but it is good enough to prevent accidental mutations.

As it happens with regular linked lists, every entry is represented by a node that has two attributes.

Name	Summary
data	Current node's value.
next	Pointer to the next node in the list.

Now that we know how the storage works on persistent lists, let’s jump to the code and see how their methods work.

Empty

This method creates an empty list. Its time complexity is O(1).

```
def self.empty
  list = SinglyLinkedList.new
  list.freeze
end
```

Insert

This method inserts a new item into a copy on the given list and return that copy.

Since we have to copy the whole list the time complexity of this method is O(n).

```
def self.insert list, data
  ret = self.copy list
  ret.insert data
  ret.freeze
end
```

Update

This method updates an item into a copy of the given list and returns that copy.

Update is the first method where we apply amortization techniques to save time and space. The strategy we use is as follow:

1. Create a copy of each element until we get to the target node.
2. Update the value on a new version of the target node.
3. Reuse elements from the node that’s next to the target node until we get to the end of the list.

The time complexity of this method is O(n) where n is the number of elements from the head of the list to the target node.

```
def self.update list, node, data
  ret   = SinglyLinkedList.new
  reuse = false
  found = false
  list.each do |nd|
    unless found
      found = (nd.data == node.data)
      if found
        ret.insert(data)
        reuse = true
        next
      end
    end
    unless reuse
      ret.insert(data)
    else
      # Reuse nodes from target to tail.
      ret.reuse_from_node(nd)
      break
    end
  end
  ret.freeze
end
```

Remove

This method removes an item from a copy of the specified list and returns that copy.

The strategy we use here is similar to the one we use for updates, but instead of creating a new version of the target node, we skip it while copying the list.

The time complexity of this method is O(n) where n is the number of elements from the head of the list to the target node.

```
def self.remove list, node
  ret   = SinglyLinkedList.new
  reuse = false
  found = false
  list.each do |nd|
    unless found
      found = (nd.data == node.data)
      if found
        reuse = true
        next # skip the target node.
      end
    end
    unless reuse
      ret.insert(nd.data)
    else
      # Reuse nodes from target to tail.
      ret.reuse_from_node(nd)
      break
    end
  end
  ret.freeze
end
```

Cat

This method creates list that contains a copy of the two given lists. (LHS, RHS)

Cat is another method where we can apply amortization techniques, this time we only copy LHS and catenate RHS to its end. Since RHS remains untouched, it can be safely shared with the rest of our program.

The time complexity of this method is O(n) where n is the number of elements on LHS.

```
def self.cat lhs, rhs
  ret = self.copy lhs
  ret.cat rhs
  ret.freeze
end
```

Len

This method returns the length of the given list.

Since the internal list has an attribute that contains that information, the time complexity of this method is O(1).

```
def self.len list
  list&.length || 0
end
```

Find First

This method returns the first element of the specified list that matches the given predicate.

The time complexity of this method is O(n).

```
def self.find_first list, &predicate
  return nil unless list
  return list.find_first &predicate
end
```

Each

This method walks the specified list yielding its elements to the given block.

The time complexity of this method is O(n).

```
def self.each list, &block
  return nil unless list
  list.each &block
end
```

Print

This method prints the contents of the specified list.

The time complexity of this method is O(n).

```
def self.print list
  unless list
    print "empty"
  else
    list.print
  end
end
```

Copy

This method is not part of the public API, but I decided to include it as part of the analysis because it is the only method that can mutate the lists.

What this method does is copy elements from the source list into a new list and return that list.

Is important to note, that the list that this method returns is not read-only! We have to take extra care each time we use it, and never expose it to the public.

The time complexity of this method is O(n).

```
def self.copy src
  dst = SinglyLinkedList.new
```

```
  src.each do |node|
    dst.insert node.data
  end
  dst
end
```

When to use persistent data structures

If you find yourself wrestling with locks, semaphores, and mutexes in your code, the chances are that you will be better off using functional data structures; Their immutable nature makes them a good fit for multithreaded execution contexts.

Conclusion

This is the end of our journey. I hope you had enjoyed the ride.

I want to sincerely thank you for buying this book and for taking the time to read it!

Feedback is more than welcomed. If you feel to write a review, please do so, it will help me a lot!

For quick updates and brief programming articles, you can visit my blog at https://medium.com/@alemiralles.

If you want to get in touch, don't hesitate to contact me. Ping me on twitter @alemiralles or drop me an email at info@amiralles.com.ar.

With nothing further ado, thank you so much! I wish you the best in programming and in life.