Data Structures

Andrew Rosen

Contents

Part I

Preliminaries

Chapter 1

Introduction

1.1 What is a Data Structures Course

Data Structures is all about defining the different ways we can organize data.

1.2 Why This Book?

This textbook is free.

It is both Java and Python.

1.2.1 Where Does This Book Fit Into a Computer Science Curriculum

Education in Computer Science is based around three core topics: translating the steps of solving a problem into a language a computer can understand, organizing data for solving problems, and techniques that can be used to solve problems. These courses typically covered in a university's introductory course, data structures course, and algorithms course respectively, although different universities decide exactly what content fits in which course. Of course, there is are lot more concepts in computer science, from operating systems and low level programming, to networks and how computers talk to each other. However, all these concepts rely on the knowledge gained in the core courses of programming, data structures, and algorithms.

This textbook is all about Data Structures, the middle section between learning how to program and the more advanced problem solving concepts we learn in Computer Science. Here, we focus on mastering the different ways to organize data, recognize the internal and performative differences between each structure, and learn to recognize the best (if there is one) for a given situation.

1.2.2 What Are My Base Assumptions about the Reader?

This textbook assumes that the student has taken a programming course that has covered the basics. Namely: data types such as ints, doubles, booleans, and strings; if statements, for and while loops; and object orient programming. This book is also suitable for the self taught programmer who has not learned much theoretical programming

1.3 To The Instructor

You'll note that this textbook lacks some of the features found in commercially available textbooks. The biggest of these is slides.

For the most part, slides are too static to help students understand how to code.

Does the lack of varied exercises make cheating on assignments easier as semesters go on? Yes, but that bridge was burned long ago. The cheating student can plagiarize from various websites or anonymously hire another to do their work for them. However, the student who cheats isn't exactly clever and certainly hasn't been exposed to much game theory. They will often cheat from the same source.

In addition, during the writing of this text, technologies such as GPTChat were released. This hasn't so much burned the bridge as dropped napalm on the entire surrounding forest. Newer technologies will then salt that earth. I recommend an open an honest dialogue with your students and at least 50% of their grade being the result of evaluations and assessments you do in class. This can range from proctored exams to flipping the classroom and giving students the chance to work on homework in class, where they are much more likely to turn to you or their peers for help.

1.3.1 How to Use

1.4 To The Student

1.4.1 How to use

Chapter 2

Functions and How They Work

This will be an extremely short chapter, but an important one.

2.1 The Runtime Stack

2.2 Argument vs Parameter

An argument is the actual value you pass in, the parameter is the variable that accepts it.

<Programming example?>

2.2.1 Does anyone actually care?

I cared enough to look it up, but I also had to look it up to double check that I'm correct. In a casual situation or talking with another programmer, I don't actually think anyone would care, but I would take care to get it correct for your assignments and exams, much like you would take care to avoid using "ain't" in a formal essay.

2.3 Passing Arguments

The vast majority of programming languages are *pass by copy* with a huge honking asterisk.

- Pass by copy means that when something is input as the argument to a function, the function gets a copy of the thing you are passing to it.

- The *huge honking asterisk* is that you are almost always passing a *reference* or *pointer* to an object, not the object itself. The reason for this is that if we had a super mega huge object, copying it would take up a super mega huge amount of time and memory.

2.3.1 How it Works in Java

In Java, we have two broad categories of data types: primitives and objects.

When you pass a primitive, such as an `int` or `double`, the value gets copied from where it is stored in memory and copied into the argument.

2.3.2 How it works in Python

Chapter 3

The Array

3.1 Why Arrays

- because new language: Since this is a data structures course, I assumed that students have had exposure to arrays or array like objects. This chapter goes into a bit of a deeper detail that may have been glossed over and Introduces the topic in the appropriate language if need be.

  In other words, I assume you know what an array is , but not necessarily how to use it in Java or Python[1].

- because internal memory lookup

- Because we need to make sure internal knowledge is cohesive (eg arrays of objects are arrays of pointers/references)

3.2 Java and Arrays

The Array is a built in class in Java, but the syntax is a bit unique [2]

To create an array in Java we do:

```
Type[] myArray = new Type[sizeOfArray]
```

Here, every item in the array is of whatever `Type` we want, which could be a Class or primitive. Arrays can be whatever integer size we desire, but once set it cannot be changed. This is because to create an array, the computer allocates a contiguous block of memory. If we wanted to resize it, there is no guarantee that this chunk of memory won't have things directly before or after it, preventing us from safely extending its range.

[1] Although we use lists in python

[2] Enough so that I constantly had to look up how to do it my first two years of undergraduate studies, so don't feel too bad if you have to do the same

3.3 Python and Arrays

Python doesn't really do arrays. It instead uses Lists, as we'll see in Chapter 5. `myNotArray`[3] `= []` does not actually make an array like you assume it would in some other language. Instead it makes A list (specifically an arraylist) to contain these items. This works exactly like an array in other languages, but you get some nifty operations that allow us to dynamically resize this array if we need it bigger or smaller.[4]

However, if you really want or need to use an array in python, you can. There are two ways to accomplish this.The first way is the built in `array` package. The python package `numpy` contains

Why would we want to use a

3.3.1 Cool Ways to Build an Array in Python

3.4 How an Array Works

3.4.1 Operations

Retrieving a item stored at an index

Setting the value of an index

3.4.2 Array Internals and the Memory Formula

3.5 Common Array Algorithms

3.5.1 Finding Values in an Array

Finding the Minimum

Finding the Average

3.5.2 Limitations

Most frequent characters Resizing

[3]On styles: Java convention is to use camel case for variable types (`myVariableName`), while python convention is to use underscores (`my_variable_name`). I will be using the Java style camel-casing for variables throughout the book for consistency and because it is my preference.

[4]We cover the specifics in Chapter **??**

Chapter 4

Analyzing Algorithms

4.1 Cost

Every function, operation, algorithm, or what have you that a computer performs has a *cost*. In fact, there are always multiples costs; we often just focus on the most important one or two costs.

What is most important depends on context. However, in the vast majority of cases, the most important cost to focus on is **time**. When our program is eating away at our storage resources like a hungry child slurping up spaghetti, we can always go out and buy more memory/storage/RAM. If our program requires a large amount of energy consumption, energy is readily available from a variety of sources: batteries, power plugs, internal combustion engines, the giant fusion reactor in the sky.

Time is different. We cannot got out and buy another weeks worth of time. Yes, processors get faster as technology marches on, but they get faster slowly and Moore's law ostensibly has its limits. The only way to make our programs realistically run faster is to make them more efficient.

Measuring Cost

When we measure cost, we need to do abstractly. When we measure the amount of time that an algorithm takes, we look at the number of operations that will be executed, not the overall elapsed time.

4.1.1 Time

A time cost is a measure of not just how long it takes a program to finish executing, bit also how the length of execution is affected by adding additional item.

Time is almost always *the most important cost.*

4.1.2 Space

4.1.3 Energy

4.2 Big O Notation

- What is big O
- how to read it
- Aside about big omega and theta
- How wrong usage annoys mathematician
- refers to cost in general, but used for time usually
- space complexity
- Common runtimes
- runtimes we'll focus on now
- runtimes we focus on later

4.2.1 Space Complexity

4.3 Examples with Arrays

- Retrieval - refer back to earlier chapter for address lookup
- Replacement
- Linear Search
- Binary Search

4.3.1 Selection Sort

4.3.2 Bubble Sort

4.3.3 Insertion Sort

4.3.4 Other Sorting Algorithms

4.4 The Formal Mathematics of Big O Notation

4.5 Other Notations

4.6 When To Ignore Costs

Part II

Lists

Chapter 5

Array Lists

The first data structure we will be studying is the list. The list is by far the most relatable data structure, as humans deal with lists on a regular basis.

5.1 What is a List?

When you get right down to it, lists are defined by order. We don't have to take advantage of this order, but its there. Populated lists have a first item and they have a last item.

Take a look at this quest below from a hypothetical fantasy game:

Quest: Slay the Dragon of Doom

- Get Sword of Dragonslaying
- Locate the map to Dragon Lair of Doom
- Travel to the Dragon Lair of Doom
- Slay the Dragon of Doom
- Return to the Castle

Here, the order is implied by the contents of the list - you can't beat the dragon without the macguffin and you certainly can't fight it without being able to find it. Generally speaking, going up against a dragon without any preparation is foolhardy in the extreme, but I digress.

Thus, you must get the special sword[1] first, and you must get the map to find the lair before you can physically travel there.

shopping list example

While lists are defined by order, we don't necessarily ascribe any meaning to the order. Take a look at the shopping list below:

[1] What if its possible to get the map before the sword? We'll see much later this kind of quest and it's requirements are much better handled by a directed acyclic graph in Chapter 17, but this example is fine for teaching lists.

<Shopping List>

While bread is the first item on this list, being the first item in the shopping list in this case has no special meaning. It's not the most important item on the list[2], nor is it necessarily the item I'm going to pick up first.

Where arrays and lists differ is that lists can grow to an arbitrary size, whereas arrays are static. Arrays can't get bigger, lists can.

A note on terminology

An **array list** is a type of list. These are sometimes called dynamic arrays.

As mentioned in Section 3.3, Python doesn't have arrays. If you've been programming in Python, you've been using an array list the entire time you've declared `[]`. They are usually just called lists rather than array lists.

I will be using the Java nomenclature for the majority of the book as this allows me to be clear about the types and implementations of data structures.

5.1.1 Lists in Java

So what does

An Aside about interfaces

Here is the source code

5.2 Generics

5.2.1 What are they?

Before we get to deep into lists, we need to have a discussion about generics. Generics are a way of restricting and specifying what types can go into a collection.

5.2.2 But Why?

5.3 List Operations

5.3.1 Size

We need some easy way of knowing how big our lists are, if for no other reason than to make sure our add and remove methods can figure our their valid indices.

5.3.2 Add

By default, we add items to the end of the list, but we can also add items to any index we want.

When we add an item at some specific index i, the item at i and all indices to the right shift over one. In other words, what was at i is now at $i + 1$, what was at $i + 1$ is at $i + 2$, and so on.

---

[2] obviously, that's the cookies

This also an understandable restriction to adding items to a list - we cannot an item to any index greater than `myList.size() +1`. Anything greater wouldn't be at the end of the list; it would be beyond it. The same goes for negative indices[3].

<possible picture showing a legal an illegal add>

We will cover this operation in more detail when we implement the add method for the arraylist

5.3.3 Remove

We can remove items from a list much in the same way we can add them. When removing an item at index i, the

For example, in the image below, we are removing the item at index 3, the word"cookie," from the list.

<image before>

C is for cookie that's good enough for me

<image after>

C is for that's good enough for me

5.3.4 Get

Get is how we retrieve our items from the list. Given an index, get will give us the value that has been stored at that index.

5.3.5 Set

5.4 ArrayLists

An array list, as you might have guessed, are lists built using *arrays*.[4] They work by growing or shrinking the array[5] automatically as items are added or removed from the list, giving the illusion that the data structure can hold an arbitrary amount of data.

We'll go into the specifics of how this works in Section 5.6.

Java's ArrayLists

Java's arrayList

Python's Lists

Python's lists, such as below:

```
l = [1,2,3] # this is a list, not an array!
```

are actually array lists!

Python uses a different vocabulary for some of the methods we'll be implementing below. For example, take the action of adding an item to a list. Python

[3]Python does allow negative indexes, but we will ignore that for now

[4]Shockingly, many of the names we give things at this point actually make sense.

[5]A lie. As you'll see we don't actually change the size of an array; we create a new array of the appropriate size and copy everything over

uses the **append** method to add an item to end of the list and **insert** to put an item into the middle of the list. Java (who's vocabulary we'll be following), uses **add** for both these contexts.

5.5 Example Algorithms

```
	public static <E> boolean isPermutation(List<E> listA,
↪	List<E> listB) {

		if(listA.size() != listB.size()) {
			return false;
		}
		for(int i  = 0; i < listA.size() ; i++){
			E item =  listA.get(i);
			int countA = 0;
			int countB = 0;

			for (E element : listA) {
				if(item.equals(element)){
					countA++;
				}
			}
			for (E element : listB) {
				if(item.equals(element)){
					countB++;
				}
			}
			if(countA != countB) {
				return false;
			}
		}
		return true;
	}
```

5.6 Building an ArrayList

To truly understand how a data structure works we need to implement it ourselves.

5.6.1 Caveats

MyArrayList.java

We will not be implementing the `List` interface. We don't need to implement all the functions to get an understanding of how the fundamentals of an arraylist work. Implementing the list interface would take up a hideous amount of physical space and get in the way of actually understanding the code.

myArrayList.py

For python, this will require some suspension of disbelief, as our array list will require using an array, and as previously discussed, arrays are shirked in favor of arraylists in python. We'll be using a list and pretending it's an array. Silly? Yes. But it will keep our code compact and easier to understand.

5.6.2 Instance Variables

Believe it or not, we only need to keep track of three instance variables to get our arraylist working.

theData We need an array to actually store the items. This is it.

size Size here refers to the total number of items we have stored in the array.

capacity This is the number of items the underlying `array` in our list can hold. It is the maximum size of the list before we have to do something about it. This is not strictly necessary as we can get it by querying `array`'s length. However, making it it's own variable will help with the readability.

It is very easy to confuse size and capacity since they both deal with counting how many elements. When I talk about `size`, I am talking about the number of items we have stored in the list we are making. Capacity, on the other hand, depends on the length of the built-in array.

Java

```
public class MyArrayList<E> {

        private E[] theData
        private int size;  // how many items are in the list
        private int capacity;  // how many items the underlying
 ↪  array can hold
```

First, note the `<E>` after `MyArrayList`. This means that we're saying:

- MyArrayList is designed to hold a specific type of object.
- Every `E` we see is a placeholder for some type, which will be that same across the entire lifespan of the object.

Python

In python, we will be creating our instance variables in the constructor.

```
class MyArrayList(object):
        pass
```

5.6.3 Constructor

We need to set the variables to their initial values upon creating the arraylist. The `size` will be 0, since we won't have any objects stored in it yet. We will set the `initial` capacity to 10. It's a small number and thus won't create much wasted space if we don't fill up `theData`. `theData` will be an empty array of `capacity` length. If `theData` becomes full, we will create a bigger array to hold our items using the `reallocate()` method (Section **??**)

Java

With our constructor, we have one line of weird black magic in order to create an Array of `E[]`'s.

```
public MyArrayList(){
        size = 0;
        capacity = 10;
        theData = (E[]) new Object[10]; // this generates a
↪   warning
}
```

So what's going on with the last line? Typically, when creating an array, we would just say:

```
//doing this in the constructor gives us an error.
TYPE[] myArray  = new Type[desired_size];
```

However, Java won't let you create new `E` objects since there's no telling what the constructors will be. This rule extends to arrays of `E`, like so:

```
theData =  new E[10];
```

However, when creating a new empty array of objects of any type, we're just making an array of nulls which will eventually be replaced by references to objects. Thus, even though the Java compiler will yell at us about Type safety, we can instead create an array of `Object` and then tell , since all references to any types are the same size.

```
// creating one array of nulls and telling Java
// its another type of array of nulls.
theData = (E[]) new Object[10];
```

Remember how Java and most modern programming languages deal with objects; if you're assigning an object to a variable, like in `Object o = new Object()`, we are storing a reference to that object.

Thus, when we add an item to a list, what really happens is we'll be adding a reference to it - the instructions on how to find it in memory.

Python

Python is fairly straightforward, with the caveat that we are pretending `theData` is an array, and not a list.

```
class MyArrayList(object):
        def __init__(self):
                self.size = 0
                self.capacity = 10
                self.theData = [None]*self.capacity
```

Since built-in lists in Python grow and shrink like we would expect a list to, we initialize `theData` with 10 `None` objects[6] to mimic the way an array would be initialized.

5.6.4 Size

Now, we will add a size method to our list; fairly straightforward in `Java`.

```
public int size() {
        return size;
}
```

In Python, we can go ahead and use the built in `__len__` method.

```
def __len__(self):
        return self.size
```

Which can then be invoked with `len(myList)`.

It is worth reinforcing that by keeping track of the size in a variable, retrieving the size of our list is always $O(1)$

5.6.5 The Add Method

Now it's time to dig into the bulk of our code: adding items to our list. To do this, I'll be creating two methods: one for adding to the end of the list (an extremely common operation) and one for adding at any index in the list.

In Java, we will overload these two methods and call them both add. We will have an `add( E item )` for adding to the end and an `add(int index, E item)` for every other case. In Python, these two `add` methods are called `append` and `insert` respectively, as Python does not support method overloading.

Cases

The `add` method has 5 basic parts, only three of which involve actual thinking about how to code:

1. Check index to see if our index in bounds
    - If it is, crash the program.
2. Check to see if our array list has room to add a new item.
    - If there is no room, make some!
    - How we do this is covered in Section 5.6.5.

[6] This is the Python equivalent to the Java `null`.

3. Shift all the existing items from `index` to the end of the list over one index to the right.

The last steps are important but not complicated: actually storing the item and then incrementing the size. We will go ahead and handle that now and put in comments for the other parts.

```
content...

content...
```

Overloading and Method Names

As previously mentioned, adding to the end is an extremely common operation, so we will overload our `add` method. If our list is provided with only an `item`, as opposed to an `item` and an `index`, we will just add that `item` to the end. Since we already wrote a perfectly good `add` method already that we know works, we'll just have our new method call that one.

```
public boolean add(E item) {
        this.add(size, item); // size is the last valid index
        return true; // What?
}
```

Why are we returning `true` here? The short answer is practice and consistency with future data structures. The long answer is any `Collection` in Java has must have an `add` method and a `List` is type of `Collection`[7].

`Collection` specifies that `add` must take in an `item` and return a `boolean`. A `true` signals the `add` is successful. A `false` signals that we could not add the `item`. For example, this might happen with a `Set` (Chapter 13)

On the other hand, our Adding at a specific index is unique to lists, and not part of collections, and will always work. Therefore, there's no need to return a boolean.

Reallocation

When we need to grow our arraylist

```
private void reallocate(){
        //doubles or 1.5x capacity
        //don't do +1 capacity
        capacity = 2 * capacity;
        E[] newData = (E[]) new Object[capacity];
        for(int i = 0; i < theData.length; i++) {
                newData[i] =  theData[i];
        }
        theData = newData;
}
```

[7]Our `MyArrayList` isn't technically a `Collection` since we did not implement the `List` interface, but I digress.

We want to double our capacity or at least increase it by 50%, rather than increasing it by a static number. Consider if we increase the capacity by one each time we reallocated. If we did that, we would have to reallocate every time we added a new item to the list. This would mean that every time we add an item to list, add becomes a linear time - $O(n)$ - operation.

Finished Code

```
public void add(int index, E item) {
        if(index < 0 || index > size) {
                throw new IndexOutOfBoundsException("Index " +
↪   index + " out of bounds.");
        }

        if(size == capacity) {
                this.reallocate(); // O(n) time...sometimes.
↪   Amortized over the cost of adding
        }

        for(int i = size - 1; i >= index; i--) { //If adding to
↪   the end... constant
                E temp = theData[i];  // Store the item from
                theData[i+1] = temp;  // Move the item from
        }

        theData[index] = item;
        size++;
}

        def insert(self, index, item):
        self.size += 1
```

5.6.6 toString and _str_

Now that we supposedly have a method for adding items into the list.

5.6.7 Get and Set

The `get` and `set` methods are fairly straightforward:

`get` - Given an `index`, retrieve the item stored at that `index`.

`set` - Given an `index`, replace the old item stored at that `index` with the provided `item`.

`set` has one additional quirk, we also want to return the old item we're replacing, just in case the programmer wants to doing something with the old item. This would obviate the need for pairing a `get` and `set` call with each other if we want to replace the old item, but do something else with it.

For both `get` and `set`, we want to throw some kind of error if the provided index is out of bounds.

Java

Our `get` is fairly straightforward, but feel free to give more information with the error.

```
public E get(int index) {
        if(index  < 0 || index >= size) {
                throw new IndexOutOfBoundsException("Index " +
↪  index + " out of bounds.");
        }
        return theData[index];
}
```

The same goes for our `set` method.

```
public E set(int index, E item) {
        if(index < 0 || index >= size) {
                throw new IndexOutOfBoundsException("Index " +
↪  index + " out of bounds.");
        }
        E oldItem  = theData[index];
        theData[index] = item;

        return oldItem;
}
```

Python

Python supports negative indices. It's up to you whether or not you want to support that, but if you do you need to account for the `capacity` and the `size` of the internal list.

We can take advantage of some of the method calls built into python to make our `myarraylist` support indexing.

```
def __getitem__(self, index):
        return self.theData[index]
```

However, this has some flaws. This method, as written, will return `None` if the user tries to access an index that is within in the bounds of the capacity but above the size. The same thing will happen if we use negative indices.

While this is fine for our pedagogical programming purposes, prudence posits proactive protection. That is to say, we should ask "how do we prevent out users from accidentally getting the wrong data when they should be getting an error."

Below, we will add two index checks.

5.6.8 Remove

5.7 Analysis

5.7.1 Constructors

Java's `ArrayList` can optionally take in an integer as an argument. This will start the underlying array's length at that value, rather than the default of 10.

This is useful if you know exactly how big your List will be. However, if you aren't removing any items when populating your list, consider using an array instead.

5.7.2 Trim

The `trim()` exists to mitigate the fact that arraylist capacity and size don't often match up.

On the other hand, Python will automatically optimize lists for you CITATION NEEDED.

For other languages, check to see if the equivalent exists

5.7.3 Adding Multiple Items in One Invocation

One common operation is to move or copy all the items from one list to another. In Java, we can use the `addAll()` method, which takes any Java collection as a parameter and all the items in that collection to the object.

```
List<Integer> a =  new ArrayList<>();
List<Integer> b =  new ArrayList<>();
for(int i = 0; i <3; i++) {
        a.add(i);
}
for(int i = 3; i <6; i++) {
        b.add(i);
}
a.addAll(b);
System.out.println(a); // 0 to 5 inclusive
System.out.println(b); // [3, 4, 5]
```

In Python, we can use the `extend()` method on anything that is iterable or use some clever slicing. However, I would always recommend using the method call over the slice, since a method invocation is always more readable.

```
a = [0, 1, 2]
b = [3, 4, 5]
c = a + b # creates a new list, which is not extend
a.extend(b) # adds all of b's items to a
a[len(a):] = b # does the same thing but unreadable.
```

A common beginner mistake in Python is to try to extend a list by calling `append` on the list like so.

```
a = [0, 1, 2]
b = [3, 4, 5]
a.append(b) # a is now [0, 1, 2, [3, 4, 5]]
```

This adds the entire list a single item in the list.

5.8 Source Code

5.8.1 Java

```
package arraylists;

public class MyArrayList<E> {

    private int size;  // how many items are in the list
    private int capacity;  // how many items the underlying array
↪   can hold
    private E[] theData;

    public MyArrayList(){
        size = 0;
        capacity = 10;
        theData = (E[]) new Object[10];
    }


    public int size() {  // O(1)
        return size;
    }


    public boolean isEmpty() {
        return (size == 0);
    }

    public boolean add(E item) {
        this.add(size, item);
        return true;
    }


    public void add(int index, E item) {
        if(index < 0 || index > size) {
            throw new IndexOutOfBoundsException("Index " +index+
↪   " was out of bounds. What are you doing???");
        }

        if(size == capacity) {  // O(n) time...sometimes.
↪   Amortized over the cost of adding
            this.reallocate();
        }

        for(int i = size - 1; i >= index; i--) { //If adding to
↪   the end... constant
            E temp = theData[i];
            theData[i+1] = temp;
```

```
        }

        theData[index] = item;
        size++;
    }

    private void reallocate(){
        //doubles or 1.5x capacity
        //don't do +1 capacity
        capacity = 2 * capacity;

        E[] newData = (E[]) new Object[capacity];
        for(int i = 0; i < theData.length; i++) {
            newData[i] =  theData[i];
        }

        theData = newData;


    }



    public E remove(int index) {
        if(index  < 0 || index >= size) {
            throw new
↪   IndexOutOfBoundsException("WE ALREADY WENT OVER THIS! IT'S OUT OF BOUNDS!!!");
        }
        E item =  theData[index];

        for(int i = index  + 1; i < size; i++) {  //O(n), unless
↪   we remove last item in the list
            theData[i-1] =  theData[i];
        }

        size--;
        return item;
    }

    public E get(int index) {
        if(index  < 0 || index >= size) {
            throw new
↪   IndexOutOfBoundsException("WE ALREADY WENT OVER THIS! IT'S OUT OF BOUNDS!!!");
        }
        return theData[index];
    }


    public E set(int index, E item) {
        if(index < 0 || index >= size) {
```

```
            throw new
↪   IndexOutOfBoundsException("WE ALREADY DID THIS JOKE!");
        }
        E oldItem  = theData[index];
        theData[index] = item;

        return oldItem;
    }

    public int indexOf(E item) {
        for (int i = 0; i < size; i++) {
            if(item.equals(theData[i])){
                return i;
            }
        }
        return -1;
    }


    public boolean contains(E item) {
        for (int i = 0; i < size; i++) {
            if(item.equals(theData[i])){
                return true;
            }
        }
        return false;
    }


    public String toString(){
        String output = "["+theData[0];
        for (int i = 1; i < size; i++) {
            output+= ", " + theData[i];
        }


       return output + "]";
    }


    public static void main(String[] args) {
        MyArrayList<Integer>  list =  new MyArrayList<Integer>();
        for(int i = 0 ; i < 5; i++){
            list.add(i);
            System.out.println(list);
        }
        list.remove(1);
        System.out.println(list);
        list.add(5);
        System.out.println(list);
```

```
    }

}
```

5.8.2 Python

```
from doctest import OutputChecker


class MyArrayList(object):
    def __init__(self) -> None:
        self.size = 0
        self.capacity = 10
        self.theData = [None]*self.capacity

    def __len__(self):
        return self.size

    def insert(self, index: int, item):
        if not isinstance(index, int):
            raise IndexError(index + " is not an integer.")
        if index < 0 or index > self.size:
            raise IndexError("Index " + str(index) +
            ↪  " is out of range.")
        if self.size ==  self.capacity:
            self.__reallocate()

        for i in range(self.size -1, index -1, -1):
            temp =  self.theData[i]
            self.theData[i+1] = temp


        self.theData[index] =  item
        self.size += 1

    def append(self, item):
        self.insert(self.size,item)

    def __reallocate(self):
        self.capacity = self.capacity * 2
        newData =  [None] * self.capacity
        for index, item in enumerate(self.theData):
            newData[index] = item
        self.theData = newData

    def remove(self, index: int):
        if index < 0 or index >= self.size:
            raise IndexError("Index " + str(index) +
            ↪  " is out of range.")

        item = self.theData[index]

        for index in range(index+1,self.size):
            self.theData[index -1] = self.theData[index]
```

```
        self.size = self.size - 1
        return item

    """
    def __str__(self): # first attempt
        output =  "["
        for item in self.theData:
            output += str(item) +","
        output = output[:-1] # remove the last comma
        return output + "]"
    """
    def __str__(self): # second attempt
        output =  "["
        #only include indexes from 0 to size-1
        for item in self.theData[:self.size]:
            output += str(item) +","
        output = output[:-1] # remove the last comma
        return output + "]"
    # obviated by dunder method
    def get(self, index):
        if index < 0 or index >= self.size:
            raise IndexError("Index " + str(index) + " is out of range.")
        return self.theData[index]

    # obviated by dunder method
    def set(self, index, item):
        if index < 0 or index >= self.size:
            raise IndexError("Index " + str(index) + " is out of range.")
        oldItem = self.theData[index]
        self.theData[index] = item
        return oldItem


    def __getitem__(self, index):
        if index < 0:
            index =  index % self.size  # yes!
            # If you're confused, test modulo on
            # negative numbers in python.
        if index >= self.size:
            raise IndexError("Index " + str(index) + " is out of range.")
        return self.theData[index]

    def __setitem__(self, index, item):
        if index < 0:
            index =  index % self.size
        if index >= self.size:
```

```
            raise IndexError("Index " + str(index) + " is out of range.")
        oldItem = self.theData[index]
        self.theData[index] = item
        return oldItem

l =  MyArrayList()
for i in range(12):
    l.append(i)
l.remove(2)
print(l)
```

Chapter 6

Linked Lists

Linked lists , also referred to as reference based lists , are the second type of lists typically seen in applications . To be clear a linked list is a list. That means it could be used anywhere an array list can. So Why do we have two objects that are functionally equivalent , two collections that hold things in order, using indexes? The answer is will see, is because each list is good at the thing the other list is less efficient at.

Array based lists use contiguous blocks of memory, allocated all at once and when then capacity of the list is filled up. Utilizing an array makes these types of lists extremely efficient at retrieving an item from a specific index, but adding items anywhere but the end of the list incurs a $O(n)$ runtime.

Linked Lists can do all the things an Array List can, but the underlying structure is completely different. Each item in the list is stored in an Object called a *Node*. Nodes are created as items are added to list, rather than in advance. This means that are not contiguous, but Rather they are scattered throughout the computer's memory . So how in the world do we keep track of where we've stored all these items ? The solution resembles the scavenger hunt through the computer's memory. Each node Not only the memory location of the item that is being stored, but the memory location of the next node in the list . An example of this code can be found below[1]:

```
// a snippet of the Node Class
// This will live inside the LinkedList class
private static class Node<E> {
        E item;
        Node<E> next;

        public Node(E item) {
                this.item =  item;
        }
}
```

[1]Why is this class private in Java `private`? An inner class (or private class) is a class that lives within another class. We use this for two reasons: Our nodes only exist to build the linked list, so they don't need to have their own class. The Second reason is What about `static class`? This means that we can create nodes without having to make a Linked List first!

Upon first glance, this code may be very confusing. Each node class contains a reference to a node inside of it. This may give the impression that nodes situated one inside another, like one of those Russian nesting matryoshka dolls. However, keep in mind what the node is actually storing is not other objects, but instead memory locations of where to find them. This means that our linked list is more akin to a scavenger hunt where each objective in the hunt contains the instructions on how to find the next objective.

In other words, the item Is the data that is being stored (well actually the memory location, don't forget that) , and next refers to the memory location of the next index in the list. Crash course is an excellent video demonstrating this which you can find here:

6.1 Connecting Nodes into a list.

we keep track of only the first and last item in the list, referred to as the head and the tail .

I will be presenting the directions to building a fully functional singly-linked list and doubly-linked list. These directions will differ from the mechanics of how your programming language of choice implements them, but have the same time complexity for their operations. My implementation is constructed with the goal of making the code easy to understand and the decisions that need to be for adding and removing reflect each other. Finally, my code aims to minimize the number of null-pointer exceptions and their ilk a programmer would make.

The full implementations can be found at the end of the Chapter.

6.2 Building a Singly LinkedList

We open up our linked list with a class declaration. If our language uses generics, we specify it there. I'll be choosing not to inherit from the built-in list so we can focus solely on our own code and no external distractions.

In Java, our code begins like this.

```
public class LinkedList<E> { }
```

In Python

```
class LinkedList(object):
pass
```

6.2.1 The Node

We want the Node class to be a private/internal class, so that the Node we write for a singly linked list and doubly linked list won't get mixed up in our coding environments. This also applies for other data structures that will be using nodes.

```
public class LinkedList<E> {

        private static class Node<E>{
                E item;
```

```
                        Node<E> next;

                        public Node(E item){
                                this.item = item;
                        }
                }
        }
```

```
class LinkedList(object):
        class Node(object):
                def __init__(self, item) -> None:
                self.item = item
                self.next = None

        pass
```

In the Node private/internal/inner class (and only there), the `this` or `self` refers to the **node** rather than the linked list.

6.2.2 Instance Variables and Constructor

Our linked list Linkedlist only needs a few Instance variables in order to Function. We need to keep track of the size; Without it we would have no idea what the valid indices are in the list. We need to keep track of the head so we know where to start our scavenger hunt for any particular index or item we're looking for. Finally we'll keep track of the tail . While keeping track of the tail isn't strictly necessary , keeping track of it means that will be able to add an item to the end of the linked list very efficiently (`O(1)`).

The only job of the constructor is to initialize everything to either zero or null.

Finally, it's probably a good idea to go ahead and write getter method for the size of the list.

```
public class LinkedList<E> {
        private Node<E> head;
        private Node<E> tail;
        private int size;


        public int size(){
                return this.size;
        }
}
```

6.2.3 Adding

Our Linked list has two add methods, just like the array list. The first only takes in an item and adds that item to the end of the linked list . It will do this by calling our second method which takes in an index and an item and inserts that item at that index.[2]

Let's take a look at our first add[3] method:

```
public boolean add(E item){
        this.add(this.size, item);
        return true;
}
```

```
def add(self, item):
        self.add(self.size, item)
        return True
```

Simple enough! But what about that second add method? When we do any kind of operation on a linked list, we need to think about how instance variables in a linked list will be altered. Fortunately, we only have three instance variables: `size`, `head`, and `tail`. When adding to a linked list, the size will always be altered as long as the index is valid. Our list's `head` will only be altered when we add an item to the beginning of the list and our `tail` will only be altered when we add to the end of the list. If the list is empty , then the node for that added item becomes both the head and the tail.

We can simplify our job by breaking the add method into five separate cases:

1. The index that we want to add to is out of bounds.
2. We are adding an item to a list that is completely empty. This is going to change the head and tail the list from nolta something.
3. We are adding an item to index 0, which is going to change the head of the list.
4. We are going to add an item to the end of the list, which means that we are going to change what the tail is.
5. We are adding to some other index in the list , which means that we don't have to bother changing the head or the tail.

Let's start with the first case.

Checking the index is in or out of bounds

Since we passed the check above , we should take a moment before we add an item to address things that need to happen no matter what for Every add condition . Specifically, we need to have a node to hold the item we are adding , and we want to go ahead and increment the size of the list At the end of the method so we don't forget about it.

I will be calling the node that holds the item we are inserting into the list `adding`, As calling it node would be extremely confusing, since we are dealing with so many nodes and other variables like next that are also four letters long.

Here's what our changes look like.

[2]If this sounds familiar, it's because this is precisely what the add method in the arraylist does. Shocking, right?

[3]As with the arraylist , the add method returns a boolean to signify that we were successfully able to add it to the list . This will always be true, but we do this because Java expects this for collections, as explained in arraylists

```
    public void add(int index, E item) {
            // Scenario 1: index is out of bound
            if(index < 0 || index > size ) {  //O(1)
                    throw new IndexOutOfBoundsException(index
↪   + " is out of bounds");
            }

            Node<E> adding = new Node<E>(item);
            /* the rest of our code*/
            size++;
```

Adding to an Empty List

Now let's consider Adding to an empty list. An empty list means the size is 0. If that's the case, we are going to make Adding the new head of the list, As well as the new tail. Just like if you are the only person in line at checkout you are both the first person and the last person in line , this node will also be the first node and the last node in the list , which is why it Will be both the head and tail of the list (at least until we add another item).

```
            // Scenario 2: adding to an initially empty list
            if(size == 0) {
                    head = adding;
                    tail = adding;
            }
```

Adding an item to the beginning of the list

Adding an item to the beginning of the list means that the node containing it becomes the new head of the list. We do this bye attaching Adding to the list, Then informing the list adding is the new head .We do this by setting adding's .next Two point to the current head of the list, then setting The list had to be the node we added.

```
            // Scenario 3: adding a new head
            else if(index == 0) {//(1)
                    adding.next = head;
                    head = adding;
            }
```

Here, we introduce one of the most important rules we need to follow when working with a linked list : when we are adding an item to the linked list attached the list first , then update the rest of the list to accommodate the new reality.

Failing to do this can have catastrophic results. Consider below Where we set Adding as new head first

```
            // Mistakes were made
            else if(index == 0) {
                    head = adding;  // oops
                    adding.next = head;
            }
```

Note that the number of operations we do here Is always the same no matter how big the list is! This means that adding to the head is a constant time operation.

Adding an item to the end of the list

```
// Scenario 4: adding a new tail
else if(index == size ){
        tail.next = adding;
        tail = adding;
}
```

Sidebar: Getting a Node at a Specific Index

```
private Node<E> getNode(int index){  //O(n)
        Node<E> current = head;
        for (int i = 0; i < index; i++) {
                current = current.next;
        }
        return current;
}
```

Inserting an item into a specific index

```
// Scenario 5: everything else
else {
        Node<E> before =  getNode(index -1);
↪   //O(n)
        adding.next = before.next;
        before.next = adding;
}
```

The end result

```
public void add(int index, E item) {
        // Scenario 1: index is out of bound
        if(index < 0 || index > size ) {  //O(1)
                throw new
↪   IndexOutOfBoundsException("Not a valid index :(");
        }

        Node<E> adding = new Node<E>(item);

        // Scenario 2: adding to an initially
↪   empty list
        if(size == 0) {
                head = adding;
                tail = adding;
        }
        // Scenario 3: adding a new head
        else if(index == 0) {  //    O(1)
```

```
                adding.next = head;
                head = adding;
        }
        // Scenario 4: adding a new tail
        else if(index == size ){
                tail.next = adding;
                tail = adding;
        }
        // Scenario 5: everything else
        else {
                Node<E> before =  getNode(index
↪  -1);  //O(n)
                adding.next = before.next;
                before.next = adding;
        }


        size++;
}
```

6.3 Get and Set

Before we got onto our remove method, let's take a look at `get` and `set` very briefly.

6.3.1 Get

Just like with an ArrayList, the get method returns the item and the specified index. However, since we can't go directly to a specific index like we can with an array or ArrayList, we need to iterate thru the `.next` links until we get to the appropriate node. Fortunately, we can just use our `getNode` function that we created when we were writing `add`.

```
public E get(int index) {
        if(index < 0 || index >= size ) {
                throw new
↪  IndexOutOfBoundsException(index + " is out of bounds");
        }
        return getNode(index).item;
}
```

6.3.2 Set

Set operates very similar to get. Remember, set also returns the item that is already at the specified index, essentially replacing it.

```
public E set(int index, E item) {
        if(index < 0 || index >= size ) {  //O(1)
                throw new
↪  IndexOutOfBoundsException(index + " is out of bounds");
```

```
                }
                Node<E> node = getNode(index);
                E toReturn = node.item;
                node.item = item;

                return toReturn;
        }
```

6.4 Remove

6.5 Analysis

Array lists and linked lists are both extremely powerful objects that fulfill the same purpose, but in radically different ways.

6.5.1 Some Algorithms Play Better

6.6 Potential Project/Practice/Labs

6.7 Source Code

```python
from typing import Generic, TypeVar


E =  TypeVar('E')

class LinkedList(Generic[E]):

    class Node(Generic[E]):
        def __init__(self, item: E) -> None:
            self.item = item
            self.next = None

    def __init__(self) -> None:
        self.head = None
        self.tail = None
        self.size = 0

    def __len__(self) -> int:
        return self.size

    def getNode(self, index: int) -> Node:
        current = self.head
        for i in range(index):
            current = current.next
        return current

    def add(self, item: E) -> bool:
```

```
        self.add(self.size,item)
        return True

    def add(self, index: int, item: E) -> None:
        if(index < 0 or index > self.size):
            raise IndexError("Invalid add at index " + str(index)
             ↪  +" with item" + str(item) +".")

        adding = self.Node(item)
        if(self.size == 0):
            self.head = adding
            self.tail = adding
        elif(index == 0):
            adding.next = self.head
            self.head = adding
        elif(index == self.size):
            self.tail.next = adding
            self.tail = adding
        else:
            before = self.getNode(index - 1)
            adding.next = before.next
            before.next = adding

        self.size += 1

    def remove(self, index: int) -> E:
        if(index < 0 or index >= self.size):
            raise Exception("Invalid remove at index " +
             ↪  str(index) +".")

        toReturn = None
        if self.size == 1:
            toReturn =  self.head.item
            self.head = None
            self.tail = None
        elif index == 0:
            toReturn = self.head.item
            self.head = self.head.next
        elif index == self.size -1:
            toReturn = self.tail.item
            self.tail = self.getNode(index - 1)
            self.tail.next =  None
        else:
            before =  self.getNode(index - 1)
            toReturn = before.next.item
            before.next = before.next.next
        self.size -= 1
        return toReturn

    def get(self, index: int) -> E:
```

```
        return self.getNode(index).item


l = LinkedList()
print(len(l))
```

Chapter 7

Stacks

Our next data structure is the Stack. The stack may seem unnecessary as a data structure after we introduce its features. After all, can't a list do all the things that a stack can do and more?

Working with the limited operations of a allows us to approach problems with a different mindset.

7.1 Stack Operations

7.2 Building a Stack

7.3 Built-in Stacks

7.4 Solving Problems with A Stack

7.5 Mazes - Stacks and Backtracking

7.6 Discrete Finite Automata

Chapter 8

Queues

A Queue (pronounced by saying the first letter and ignoring all the others) is a data structure which emulates the real word functionality of standing in a line (or queue, for those from Commonwealth nations). In a Queue, items are processed in the order they are inserted into the Queue. So if Alice enters the Queue, followed by Bob, followed by Carla, Alice would be the first to leave the Queue, then Bob, and then Carla.

The use cases for Queues are fairly obvious

8.1 Linked Based Implementation

8.2 Array Based Implementation

We could use

Part III

Recursion

Chapter 9

Recursion

9.1 Introduction

9.2 Recursive Mathematics

9.2.1 Fibonacci

As it turns out, while this technically works...it's pretty terrible. In short, using recursion, I managed to accidentally[1] write an $O(2^n)$, or exponential time, algorithm. This is very bad. This means increasing n by one *doubles* the runtime of out algorithm!

9.3 Redoing Things With Recursion

Many of the things we are about to see should not be actually used and serve only as examples, like our `printThis` function

9.3.1 Printing Recursively

9.3.2 Recursive Linear Search

9.4 Binary Search

9.4.1 Runtime Analysis

How to not be scared of logarithms

You may have learned that logarithms are the inverse operation to exponentiation.

This is an utterly useless definition when programming.

A more way of thinking about logarithms is "how many times can I recursively split something?" For example, $\log bx$ asks "how many times can I recursively split my x items into b seperate piles?"

[1] All right, I did this totally on purpose.

A more concrete example: $\log_2 16 = 4$, not because $2^4 = 16$, but because a pile of 16 items can be split in half into two piles of 8, each pile of 8 can be split in half into two piles of 4, the 4's can be split into 2's, the 2's into 1's — four splits total:

<picture>

Back to it.

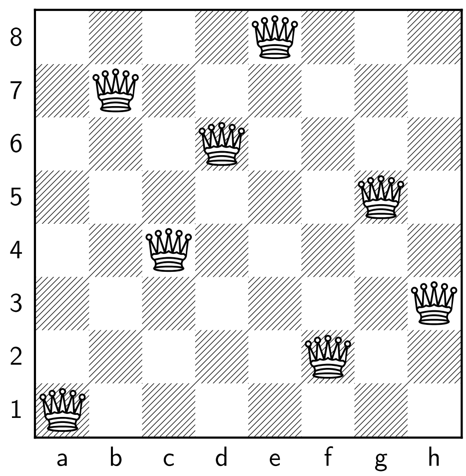
9.5 Recursive Backtracking

Recursion really comes in handy when we are trying to solve complex puzzles. One of the most famous examples of

The Recursive Backtracking Algorithm

9.5.1 Mazes Again

9.5.2 The Eight Queens Puzzle



Brute Force Solution

A brute force algorithm means we will be checking every single possible state to find a solution. In this case, a brute force solution for the Eight Queens Puzzle would every possible placement of eight queens on a chessboard, such as these two:

<Chess notation here>

There are a total of $\binom{64}{8} = 4426165368$ possible ways to place 8 queens on a chessboard with 64 spaces.

Recursive Solution Outline

A Place Holder For Validity

Performing the Recursion

Checking just One condition

Checking all the Conditions

9.5.3 Additional Problems left to the Reader

Knight's Tour

Sudoku

9.6 Recursive Combinations

9.7 Recursion and Puzzles

9.8 Recursion and Art

9.9 Recursion and Nature

Chapter 10

Trees

Our next major data structure is trees. Specifically, we will be looking at binary search trees.

Trees are an excellent data structure for storing things since they implement all the operations we care about for collections in logarhythmic time[1]

However, trees are not without limitations. Trees will only work with data that can be stored hierarchically or in an order.

10.1 The Parts of a Tree

The first thing we need to do when introducing trees is define a vocabulary.

Much like the linked list, a tree is made of nodes. However, unlike a linked list , nodes in a tree are not arranged in a line, Instead, they are arranged in a heirachy.

Each node sit above multiple other nodes , with the nodes below it being referred to as their children or child nodes. The node connecting all these children is called the parent.

<A picture of one node, Represented by a circle with four arrows coming out below it. Each arrow points to yet another node. The Node with the arrows coming out of it is the parent, and the nodes below it are the children >

This relationship can be extended Ad infinitum as we can see with the picture below

<Picture with nodes labeled>

However anything above grandchild and grandparent just becomes tedious , so we tend to Generalize this relationship to ancestors and descendants. A key point here is to remember that while we are borrowing terms from the family tree , nodes will only have one parent . Each node can have multiple children, however .

We refer to the links connect each of the nodes as branches or links or edges. This tends to be a matter of personal preference.

[1] Specifically , Trees implement everything in average case log rhythmic time and worst case linear time , but if we do a bit of extra work and make it a self balancing binary tree (which will seem much later in this chapter) we can make this tree worst case log arhythmic for all operations

Finally , we have one special node that sits above all the other nodes . This note is the root and it is analogous to the head of a linked list . All of our operations will start at the root of the node[2].

Remember , programmers are stereotypically outdoors of averse, So they May have forgotten what a real tree looks like. Thus, we'll see that the root of the tree is at the top of the tree and our leaves are at the bottom[4]

10.1.1 Where the Recursion comes in

There is a reason we learned recursion before we introduce trees. Trees are the exemplar recursive data structure

Each tree has a root and That route has children . If we view each of those children as the root of their own subtree , this can make our algorithms for adding removing and searching extremely easy to write.

<picture Of tree, the recursive subtrees are dash circled.>

<Picture of the left subtree, with it's trees circled>

10.2 Binary Search Trees

A diagram of a binary search tree. It is made up of nodes, represented by circles, and edges (also called links or branches), represented by arrows.

[5]

10.3 Building a Binary Search Tree

10.3.1 The Code Outline

We use the `Comparable` class in Java to require that all objects stored in the tree has a **total ordering**[6].In practice, this means that anything `Comparable` can be sorted.

Python, of course, doesn't need these restrictions.

```
public class BinaryTree<E extends Comparable<E>> {
}
```

Much like our Linked List, we don't need much in the way of instance variables. We'll create a `root` to keep track of the starting place for our tree and size to keep track of how many items we have stored.

Finally, we will also create our inner `Node` class for the Tree. It needs to hold the item and the locations of the left and right children. We'll also go ahead and add a The constructor and a method for printing out the item in the node (`toString` in Java and `__str__` in Python)

[2]Remember , programmers are stereotypically outdoors of averse, So they May have forgotten what a real tree looks like. Thus, we'll see that the root of the tree is at the top of the tree and our leaves are at the bottom[3]

[4]Or maybe it's some weird hydroponic zero-G kind of thing.

[5]An aside about array based implementations.

[6]The formal definition is as follows

```
public class BinaryTree<E extends Comparable<E>> {
        private Node<E> root;
        private int size;

        public BinaryTree() {
                this.root = null;

        }

        /* Other code will go here.*/

        private static class Node<E extends
↪   Comparable<E>> {
                private E item;
                private Node<E> left;  // left child
                private Node<E> right; // right child
                public Node(E item) {
                        this.item = item;
                }
                public String toString() {
                        return item.toString();
                }
        }


}
```

10.3.2 Add

10.3.3 Contains

10.3.4 Delete

Chapter 11

Heaps

11.1 Priority Queues

11.2 Removing From other locations

Chapter 12

Sorting

Now that we have a handle on sorting =,

12.1 Quadratic-Time Algorithms

12.1.1 Bubble Sort

12.1.2 Selection Sort

12.1.3 Insertion Sort

12.2 Log-Linear Sorting Algorithms

The most commonly used sorting algorithms take $O(n \lg(n))$ time. This is the hard limit on runtime

12.2.1 Tree Sort

The tree sort is the simplest algorithm to we will cover. Performing Tree sort is a matter of three simple steps

1. Create a tree.
2. Load the items you want to sort into the tree.
3. Perform an inorder traversal of the tree.

The performance of this algorithm depends completely on the type of tree we create for this algorithm. Using a self-balancing binary search tree, adding n items to the tree takes $O(n \lg(n))$ and an in order traversal takes $O(n)$ steps, for a grand total of $O(n)$ runtime. Using a binary search tree that does not self balance means that there is a worst case scenario of $O(n^2)$ for adding all the n items.

Using a tree also means we use extra space since all the data has to be moved into a tree, using $O(n)$ space.

12.2.2 Heap Sort

You might expect that heapsort deserves the same treatment as treesort. After all, a heap has the same structure as a tree and both are constructed to perform operations in $logn$ time.

12.2.3 Heapify

12.2.4 Quick Sort

12.2.5 Merge Sort

12.3 Unique Sorting Algorithms

12.3.1 Shell Sort

12.3.2 Radix Sort

12.4 State of the Art Sorting Algorithms

12.4.1 Tim Sort

12.4.2 Quick Sort

12.5 But What if We Add More Computers: Parallelization and Distributed Algorithms

GENERATED BY chapgpt

Parallel sorting algorithms are designed to be executed on a single computer with multiple processors or cores, while distributed sorting algorithms are designed to be executed on a network of computers working together. Both types of algorithms can be used to significantly improve the performance of sorting for large data sets, especially when the data does not fit in the memory of a single computer.

There are many different parallel and distributed sorting algorithms, each with its own characteristics and trade-offs. Some common techniques used in these algorithms include:

Data partitioning: Splitting the data into smaller chunks that can be sorted independently and then merged back together. Load balancing: Ensuring that the work is distributed evenly among the available processors or computers. Communication: Allowing the processors or computers to communicate and exchange data during the sorting process.

Some examples of parallel and distributed sorting algorithms include:

Parallel merge sort: A parallel version of the merge sort algorithm that divides the data into smaller chunks and sorts them in parallel, then merges the sorted chunks back together. MapReduce: A programming model for distributed computing that is often used for sorting large data sets in a distributed environment, such as on a cluster of computers. Bitonic sort: A parallel sorting algorithm that uses a recursive divide-and-conquer approach to sort the data using a network of processors.

There are many other parallel and distributed sorting algorithms as well, each with their own specific characteristics and trade-offs. If you are interested in learning more about these algorithms, you may want to consider reading more about parallel and distributed computing, as well as specific techniques such as data partitioning, load balancing, and communication.

Parallel VS Distributed

12.6 Further Reading

12.6.1 Pedagogical Sorting Algorithms

Sleep Sort

Part IV

Hashing

Chapter 13

Sets

Sets programmed implementations of mathematical sets

13.1 Operations

We will use Venn diagrams to graphically demonstrate operates with two sets

13.1.1 Adding an item to a Set

Adding items to a set is fairly straightforward.

As we will see, adding to a set can be either $O(1)$ or $O(\log n)$ time, depending on the implementation

13.1.2 Removing an item to a Set

13.1.3 Union

In Java, this is the `addAll()` method.

13.1.4 Intersection

13.1.5 Set Difference

13.1.6 Subset

13.2 Operation Analysis

Most sets are implemented using a Hash Table.

13.2.1 TreeSet Vs HashSet Vs Linked Hash Set

13.3 Sets and Problem Solving

Sets are super efficient checklists.

13.3.1 Checking for Uniqueness or Finding Duplicates

Chapter 14

Maps

Chapter 15

Hash Tables

15.1 Creating a Hash Function

Chapter 16

Map Reduce

16.1 Map

The `map()` operation[1] is a powerful function that may require us to think differently about the way we have approached programming so far.

The map operation takes in 2 arguments, a collection and a function to apply to every item in the collection

When we are writing functions , we are creating new verbs for our programming language to use . These verbs take in arguments, nouns that we may have declared or defined ourselves. But one thing that we May not have done yet is passing a function as an argument to another function.

This is not an uncommon operation in mathematics Example listed below

The semantics of this in every programming language is different , but the concept is the same

Why introduces here? Because a lot of common operations that can be done with map reduce involve using hash tables

[1]It is mildly confusing that there is a `map` data structure and a `map()` operation, so I will be marking the `map()` operation with a function invocation.

Part V

Relationships

Chapter 17

Graphs

In some ways, Graphs are the most important data structure. Graphs represent and model relationships, and humans are defined by relationships. The archetypical examples of graphs used to be maps and the distances between landmarks or looking for the shortest path.

With the advent of social media, we can talk about graphs with a few examples that might be easier to intuit.

17.1 Introduction and History

17.2 Qualities of a Graph

The physical layout of a graph doesn't actually matter[1]

17.2.1 Vertices

- Vertices must be unique.

17.2.2 Edges

Undirected Edges

Directed Edges

Weighted Edges

17.3 Special Graphs and Graph Properties

17.3.1 Planar Graphs

Graphs that are planar can have their vertices and edges laid out in such a way that no two edges will cross.

[1] Some properies, such as whether a graph is *planar* or *bipartite* effectively care if a graph can be physically laid out in a certain way.

Figure 17.1: The wings of a dragonfly. Credit: Joi Ito (CC BY 2.0)

17.3.2 Bipartite Graphs

17.3.3 Directed Acyclic Graphs

17.4 Building a Graph

17.4.1 Adjacency List

17.4.2 Adjacency Matrix

Matrix multiplication and GPU Abuse

17.5 Graph Libraries

17.5.1 Java - JUNG

17.5.2 Python - networkx

There is only one realistic choiceforusing graphs in Python. The package networkx is extremely powerful, extremely versatile,and actively maintained.

17.6 Graphs, Humans, and Networks

17.6.1 The Small World

The Milgram Experiment

The Less-Known Milgram Experiment

17.6.2 Scale Free Graphs

17.7 Graphs in Art and Nature - Voronoi Tessellation

Chapter 18

Graph Algorithms

18.1 Searching and Traversing

18.1.1 Breadth First Search

18.1.2 Depth First Search

18.2 Shortest Path

18.2.1 Djikstra's Algorthim

Improving The Algorithm

Failure Cases

18.2.2 Bellman-Ford

18.3 Topological Sorting

18.3.1 Khan's Algorithm

18.4 Minimum Spanning Trees

18.4.1 Kruskal's Algorithm

18.4.2 Prim's Algorithm