Data Structures

Andrew Rosen

Contents

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?

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. The first writeup of the textbook will be done in Java, but I will try to add as much Python into the book as well.

1.3 To The Instructor

1.4 To The Student

Chapter 2

The Array

2.1 Array Operations

2.2 Finding Values in an Array

Chapter 3

Analyzing Algorithms

3.0.1 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, when we measure cost, we need to do abstractly. When we measure the amount of time that an algorithm takes

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.*

Space

Energy

Other costs - Bandwidth

3.1 Big O Notation

3.1.1 Space Complexity

3.2 The Formal Mathematics of Big O Notation

3.3 Other Notations

Chapter 4

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.

4.1 What is a list?

When you get right down to it, lists are defined by order.

```
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;
}
```

4.2 ArrayLists

An array list, as you might have guessed, are lists built using *arrays*.[1] They work by growing or shrinking the array[2] 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 4.2.2.

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 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.

4.2.1 Generics

4.2.2 Building an ArrayList

4.2.3 More Restrictive or Permissive Generics

4.3 Linked Lists

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[3]:

```
// a snippet of the
private static class Node<E> {
        E item;
```

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

[2] 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

[3] Why is this class private in Java `private`? An inner class (or private class) is a class that lives within another cl What about `static class`? This means that we can create nodes without having to make a Linked List first!

```
        Node<E> next;

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

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.

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.

4.3.1 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 looks like this.

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

In Python

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

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.

```java
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.

Instance Variables and Constructor

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

Adding

We start writing our add method by defining a

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.

Getting a Node at a Specific Index

4.4 Analysis

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

4.5 Source Code

```
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(index,index,item)
    return True

def add(self, index: int, item: E) -> None:
    if(index < 0 or index > self.size):
        raise Exception("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:
            self.head = None
            self.tail = None
        elif index == 0:
            toReturn = self.head.item
            self.head = self.head.next

        self.size -= 1
        return toReturn



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

Chapter 5

Stacks

5.1 Building a Stack

5.2 Mazes - Stacks and Backtracking

5.3 Discrete Finite Automata

Chapter 6

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 obi

6.1 Linked Based Implementation

6.2 Array Based Implementation

We could use

Chapter 7

Recursion

7.1 Recursive Mathematics

7.2 Recursive Problem Solving

7.2.1 Recursive Backtracking

7.2.2 Recursive Combinations

7.3 Recursion and Puzzles

7.4 Recursion and Art

7.5 Recursion and Nature

Chapter 8

Trees

8.1 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.

8.2 Heaps

8.2.1 Priority Queues

8.3 Trees and Heaps in Java

Chapter 9

Sorting

9.1 Quadratic-Time Algorithms

9.1.1 Bubble Sort

9.1.2 Selection Sort

9.1.3 Insertion Sort

9.2 Log-Linear Sorting Algorithms

9.2.1 Tree Sort

9.2.2 Heap Sort

9.2.3 Quick Sort

9.2.4 Merge Sort

9.3 Unique Sorting Algorithms

9.3.1 Shell Sort

9.3.2 Radix Sort

9.4 State of the Art Sorting Algorithms

9.4.1 Tim Sort

9.4.2 Quick Sort

9.4.3 Distributing and Parallelization

Chapter 10

Sets and Maps

10.1 Sets

10.2 Maps

10.3 Hash Tables

10.3.1 Creating a Hash Function

10.4 Map Reduce

Chapter 11

Graphs

11.1 Introduction and History

11.2 Qualities of a Graph

11.2.1 Undirected Edges

11.2.2 Directed Edges

11.2.3 Weighted Edges

11.3 Directed Acyclic Graphs

11.4 Building a Graph

11.4.1 Adjacency List

11.4.2 Adjacency Matrix

11.5 Graph Algorithms

11.5.1 Searching and Traversing

Breadth First Search

Depth First Search

11.5.2 Shortest Path

11.5.3 Topological Sorting

11.5.4 Minimum Spanning Trees

11.6 Graphs, Humans, and Networks

11.6.1 The Small World

The Milgram Experiment

The Less-Known Milgram Experiment

11.6.2 Scale Free Graphs

11.7 Graphs in Art and Nature - Voronoi Tessellation

11.8 Distributed Hash Tables

Figure 11.1: The wings of a dragonfly. Credit: Joi Ito (CC BY 2.0)

Chapter 12

Other Data Structures

12.1 Skip Lists