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

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 ray is , but not necessarily how to use it in Java or Python[1].

2.1 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.

2.2 Python and Arrays

Python doesn't really do arrays like other languages. `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]

[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]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 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. You can bui The python package `numpy` contains

Why would we want to use a

2.2.1 Cool Ways to Build a List in Python

Con

2.3 How an Array Works

2.3.1 Operations

Retrieving a item stored at an index

Setting the value of an index

2.3.2 Array Internals and the Memory Formula

2.4 Common Array Algorithms

2.4.1 Finding Values in an Array

Finding the Minimum

Finding the Average

2.4.2 Limitations

Most frequent characters

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 Examples with Arrays

3.3 The Formal Mathematics of Big O Notation

3.4 Other Notations

Chapter 4

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.

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 4.4.

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.

4.3 Generics

4.4 Building an ArrayList

4.4.1 More Restrictive or Permissive Generics

4.5 Analysis

[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

Chapter 5

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:

5.1 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.

5.2 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
```

5.2.1 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.

5.2.2 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;
        }
}
```

5.2.3 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++;
}
```

5.3 Get and Set

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

5.3.1 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;
}
```

5.3.2 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;
}
```

5.4 Remove

5.5 Analysis

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

5.6 Potential Project/Practice/Labs

5.7 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 6

Stacks

6.1 Building a Stack

6.2 Mazes - Stacks and Backtracking

6.3 Discrete Finite Automata

Chapter 7

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

7.1 Linked Based Implementation

7.2 Array Based Implementation

We could use

Chapter 8

Recursion

8.1 Introduction

8.2 Recursive Mathematics

8.2.1 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!

8.3 Redoing Things With Recursion

Many of the things we are about to see should not be attempted in production and serve only as examples, like our printThis function

8.4 Recursive Problem Solving

8.4.1 Recursive Backtracking

8.4.2 Recursive Combinations

8.5 Recursion and Puzzles

8.6 Recursion and Art

8.7 Recursion and Nature

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

Chapter 9

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.

9.1 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]

9.1.1 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>

9.2 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.

9.3 Building a Binary Search Tree

9.3.1 The Code Outline

We use the `Comparable` class in Java to require that all objects stored in the tree has a **total ordering**[5].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)

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

[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]The formal definition is as follows

```
        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();
                }
        }


}
```

9.3.2 Add

9.3.3 Contains

9.3.4 Delete

Chapter 10

Heaps

10.0.1 Priority Queues

Chapter 11

Sorting

11.1 Quadratic-Time Algorithms

11.1.1 Bubble Sort

11.1.2 Selection Sort

11.1.3 Insertion Sort

11.2 Log-Linear Sorting Algorithms

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

11.2.1 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.

11.2.2 Heap Sort

11.2.3 Quick Sort

11.2.4 Merge Sort

11.3 Unique Sorting Algorithms

11.3.1 Shell Sort

11.3.2 Radix Sort

11.4 State of the Art Sorting Algorithms

11.4.1 Tim Sort

11.4.2 Quick Sort

11.4.3 Distributing and Parallelization

Chapter 12

Sets and Maps

12.1 Sets

12.2 Maps

12.3 Hash Tables

12.3.1 Creating a Hash Function

12.4 Map Reduce

Chapter 13

Graphs

13.1 Introduction and History

13.2 Qualities of a Graph

13.2.1 Undirected Edges

13.2.2 Directed Edges

13.2.3 Weighted Edges

13.3 Directed Acyclic Graphs

13.4 Building a Graph

13.4.1 Adjacency List

13.4.2 Adjacency Matrix

13.5 Graph Algorithms

13.5.1 Searching and Traversing

Breadth First Search

Depth First Search

13.5.2 Shortest Path

13.5.3 Topological Sorting

13.5.4 Minimum Spanning Trees

13.6 Graphs, Humans, and Networks

13.6.1 The Small World

The Milgram Experiment

The Less-Known Milgram Experiment

13.6.2 Scale Free Graphs

13.7 Graphs in Art and Nature - Voronoi Tessellation

13.8 Distributed Hash Tables

Figure 13.1: The wings of a dragonfly. Credit: Joi Ito (CC BY 2.0)

Chapter 14

Other Data Structures

14.1 Skip Lists