Preface

The design and analysis of ef·cient data structures has long been recognized as a vital subject in computing and is part of the core curriculum of computer scienceand computer engineering under in Python provides an introduction to data structures and algorithms, including their design, analysis, and implementation. This book is designed for use in a beginning-level data structures course, To promote the development of robust and reusable software, we have tried to take a consistent object-oriented viewpoint throughout this text. One of the mainideas of the object-oriented appr jects as instances of an abstract data type (ADT), which includes a repertoire of methods for performing operations on data objects of this type. We then empha- size that there may be several different implementation strategies for a particularADT, and explore the relative pr Python implementations for almost all data structures and algorithms discussed, and we introduce important object-oriented design patterns as means to organize those implementations into reusable components.

Desired outcomes for readers of our book include that:

·They have knowledge of the most common abstractions for data collections (e.g., stacks, queues, lists, trees, maps).

·They understand algorithmic strategies for producing ef·cient realizations ofcommon data structures.

·They can analyze algorithmic performance, both theoretically and experi-mentally, and recognize common trade

·They can wisely use existing data structures and algorithms found in modern programming language libraries.

·They have experience working with concrete implementations for most foun- dational data structures and algorithms.

·They can apply data structures and algorithms to solve complex problems.

In support of the last goal, we present many example applications of data structuresthroughout the book, includir in structured formats such as HTML, simple cryptography, text frequency analy- sis, automated geometric layout, Huffman coding, DNA sequence alignment, andsearch engine indexing.

Our study of data structures thus far has focused primarily upon the ef·ciency of computations, as measured by the number of primitive operations that are executed on a central processing unit (CPU). In practice, the performance of a com-puter system is also greatly impa more subtle issues involving the use of a computer·s memory system.

We ·rst discuss ways in which memory is allocated and deallocated during the execution of a computer program, and the impact that this has on the program·sperformance. Second, we discus in today·s computer systems. Although we often abstract a computer·s memoryas consisting of a single pool of i memory). We consider the use of classic data structures in the algorithms used to manage memory, and how the use of memory hierarchies impacts the choice of data structures and algorithms for classic problems such as searching and sorting.

15.1 Memory Management

In order to implement any data structure on an actual computer, we need to use computer memory. Computer memory is organized into a sequence of words , each of which typically consists of 4, 8, or 16 bytes (depending on the computer). These memory words are numbered from 0 to N·1, where Nis the number of memory words available to the computer. The number associated with each memoryword is known as its memory ad viewed as basically one giant array of memory words. For example, in Figure 5.1 of Section 5.2, we portrayed a section of the computer·s memory as follows:

2160 2145214621472148214921502151215221532154215521562157 2158 2144 2159

In order to run programs and store information, the computer·s memory must bemanaged so as to determine what data is stored in what memory cells. In this section, we discuss the basics of memory management, most notably describing the way in which memory is allocated to store new objects, the way in which portionsof memory are deallocated and which the Python interpreter uses memory in completing its tasks.

14.2 Data Structures for Graphs

In this section, we introduce four data structures for representing a graph. In each representation, we maintain a collection to store the vertices of a graph. However, the four representations differ greatly in the way they organize the edges.

·In an edge list , we maintain an unordered list of all edges. This minimally suf·ces, but there is no ef·cient way to locate a particular edge (u,v),o rt h e set of all edges incident to a vertex v.

·In an adjacency list , we maintain, for each vertex, a separate list containing those edges that are incident to the vertex. The complete set of edges can be determined by taking the union of the smaller sets, while the organizationallows us to more ef·ciently ·nd all e

·Anadjacency map is very similar to an adjacency list, but the secondary container of all edges incident to a vertex is organized as a map, rather thanas a list, with the adjacent vertex se

·Anadjacency matrix provides worst-case O(1)access to a speci·c edge (u,v)by maintaining an n×nmatrix, for a graph with nvertices. Each entry is dedicated to storing a reference to the edge (u,v)for a particular pair of vertices uand v; if no such edge exists, the entry will be None .

A summary of the performance of these structures is given in Table 14.1. We give further explanation of the structures in the remainder of this section.

Operation
Edge List
Adj. List
Adj. Map
Adj. Matrix
vertex
count()
O(1)
O(1)
O(1)
O(1)
edge
count()
O(1)
O(1)
O(1)
O(1)
vertices()
O(n)
O(n)
O(n)
O(n)
edges()
O(m)
O(m)
O(m)
O(m)
get
edge(u,v)
O(m)
O(min (du,dv))

In a classic story, the famous mathematician Archimedes was asked to deter-
mine if a golden crown commissioned by the king was indeed pure gold, and not
part silver, as an informant had claimed. Archimedes discovered a way to performthis analysis while stepping int
·Eureka, eureka!· for he had discovered an analysis tool (displacement), which,
when combined with a simple scale, could determine if the king·s new crown wasgood or not. That is, Archimede
discovery was unfortunate for the goldsmith, however, for when Archimedes did
his analysis, the crown displaced more water than an equal-weight lump of puregold, indicating that the crown w
In this book, we are interested in the design of ·good· data structures and algo-
rithms. Simply put, a data structure is a systematic way of organizing and access-
ing data, and an algorithm is a step-by-step procedure for performing some task in
a ·nite amount of time. These concepts are central to computing, but to be able toclassify some data structures a
The primary analysis tool we will use in this book involves characterizing the
running times of algorithms and data structure operations, with space usage also
being of interest. Running time is a natural measure of ·goodness,· since time is a
precious resource·computer solutions should run as fast as possible. In general,the running time of an algorithm
ning time is affected by the hardware environment (e.g., the processor, clock rate,
memory, disk) and software environment (e.g., the operating system, programminglanguage) in which the algorit
is done in a program compiled into native machine code instead of an interpreted
implementation. We begin this chapter by discussing tools for performing exper-imental studies, yet also limitatio
Focusing on running time as a primary measure of goodness requires that we be
able to use a few mathematical tools. In spite of the possible variations that comefrom different environmental fa
and develop a mathematical way of analyzing algorithms.

13.1 Abundance of Digitized Text

Despite the wealth of multimedia information, text processing remains one of the dominant functions of computers. Computer are used to edit, store, and displaydocuments, and to transport docu tems are used to archive a wide range of textual information, and new data is being generated at a rapidly increasing pace. A large corpus can readily surpass a petabyteof data (which is equivalen

·Snapshots of the World Wide Web, as Internet document formats HTML andXML are primarily text formats, with
·All documents stored locally on a user·s computer
·Email archives
·Customer reviews
·Compilations of status updates on social networking sites such as Facebook
·Feeds from microblogging sites such as Twitter and Tumblr

These collections include written text from hundreds of international languages. Furthermore, there are large data sets (such as DNA) that can be viewed computationally as ·strings· even though they are not language.

In this chapter we explore some of the fundamental algorithms that can be used to ef·ciently analyze and process large textual data sets. In addition to having interesting applications, text-processing algorithms also highlight some important algorithmic design patterns.

We begin by examining the problem of searching for a pattern as a substring of a larger piece of text, for example, when searching for a word in a document. The pattern-matching problem gives rise to the brute-force method , which is often inef·cient but has wide applicability.

Next, we introduce an algorithmic technique known as dynamic programming , which can be applied in certain settings to solve a problem in polynomial time that appears at ·rst to require exponential time to solve. We demonstrate the applicationon this technique to the probl be similar but not perfectly aligned. This problem arises when making suggestions for a misspelled word, or when trying to match related genetic samples.

Because of the massive size of textual data sets, the issue of compression is important, both in minimizing the number of bits that need to be communicated through a network and to reduce the long-term storage requirements for archives. For text compression, we can apply the greedy method , which often allows us to approximate solutions to hard problems, and for some problems (such as in textcompression) actually gives rise

Finally, we examine several special-purpose data structures that can be used to better organize textual data in order to support more ef·cient run-time queries.

15.2 Memory Hierarchies and Caching

With the increased use of computing in society, software applications must man-
age extremely large data sets. Such applications include the processing of online·nancial transactions, the organ
ses of customers· purchasing histories and preferences. The amount of data can be
so large that the overall performance of algorithms and data structures sometimesdepends more on the time to a

15.2.1 Memory Systems

In order to accommodate large data sets, computers have a hierarchy of differ-
ent kinds of memories, which vary in terms of their size and distance from theCPU. Closest to the CPU are the ir
cess to such locations is very fast, but there are relatively few such locations. At
the second level in the hierarchy are one or more memory caches . This memory
is considerably larger than the register set of a CPU, but accessing it takes longer.At the third level in the hierarc
main memory orcore memory . The internal memory is considerably larger than
the cache memory, but also requires more time to access. Another level in the hi-erarchy is the external memory
drives, and/or tapes. This memory is very large, but it is also very slow. Data storedthrough an external network
with even greater storage capacity, but even slower access. Thus, the memory hi-
erarchy for computers can be viewed as consisting of ·ve or more levels, each ofwhich is larger and slower than

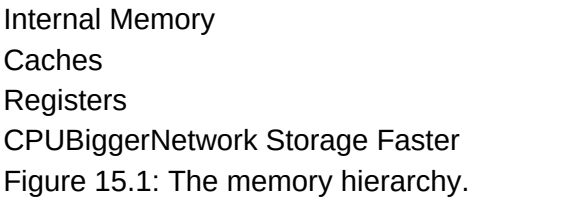

Figure 15.1: The memory hierarchy.

Contents and Organization

The chapters for this book are organized to provide a pedagogical path that starts with the basics of Python programming and object-oriented design. We then add foundational techniques like algorithm analysis and recursion. In the main portionof the book, we present fundam

1.Python Primer
2.Object-Oriented Programming
3.Algorithm Analysis
4.Recursion
5.Array-Based Sequences
6.Stacks, Queues, and Deques
7.Linked Lists
8.Trees
9.Priority Queues
10.Maps, Hash Tables, and Skip Lists
11.Search Trees
12.Sorting and Selection
13.Text Processing
14.Graph Algorithms
15.Memory Management and B-Trees
A.Character Strings in Python
B.Useful Mathematical Facts

A more detailed table of contents follows this preface, beginning on page xi.

Prerequisites

We assume that the reader is at least vaguely familiar with a high-level program-ming language, such as C, C++

·Variables and expressions.
·Decision structures (such as if-statements and switch-statements).
·Iteration structures (for loops and while loops).
·Functions (whether stand-alone or object-oriented methods).

For readers who are familiar with these concepts, but not with how they are ex-pressed in Python, we provide a p give a comprehensive treatment of Python.

It is worth noting that top-level commands with the module source code are executed when the module is ·rst imported, almost as if the module were its own script. There is a special construct for embedding commands within the modulethat will be executed if the modul
if
name
==
__main__
:
Using our hypothetical utility.py module as an example, such commands will be executed if the interpreter is started with a command python utility.py,b u t not when the utility module is imported into another context. This approach is often used to embed what are known as unit tests within the module; we will discuss unit testing further in Section 2.2.4.

1.11.1 Existing Modules

Table 1.7 provides a summary of a few available modules that are relevant to a study of data structures. We have already discussed the math module brie·y. In the remainder of this section, we highlight another module that is particularly important for some of the data structures and algorithms that we will study later in this book.

Existing Modules

Module Name	Description
array	Provides compact array storage for primitive types.
collections	De·nes additional data structures and abstract base classes involving collections of objects.
copy	De·nes general functions for m aking copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	De·nes common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of in teraction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

Table 1.7: Some existing Python modules relevant to data structures and algorithms.

Pseudo-Random Number Generation

Python·s random module provides the ability to generate pseudo-random numbers, that is, numbers that are statistically random (but not necessarily truly random). Apseudo-random number generator uses a deterministic formula to generate the

Class Data Members
A class-level data member is often used when there is some value, such as a con-
stant, that is to be shared by all instances of a class. In such a case, it would
be unnecessarily wasteful to have each instance store that value in its instance
namespace. As an example, we revisit the PredatoryCreditCard introduced in Sec-
tion 2.4.1. That class assesses a $5 fee if an attempted charge is denied becauseof the credit limit. Our choice o
the literal value in our code. Often, the amount of such a fee is determined by the
bank·s policy and does not vary for each customer. In that case, we could de·neand use a class data member as
classPredatoryCreditCard(CreditCard):
OVERLIMIT
FEE = 5 #t h i si sac l a s s - l e v e lm e m b e r
defcharge(self,p r i c e) :
success = super().charge(price)
if not success:
self.
balance += PredatoryCreditCard.OVERLIMIT
FEE
return success
The data member, OVERLIMIT
FEE, is entered into the PredatoryCreditCard
class namespace because that assignment takes place within the immediate scope
of the class de·nition, and without any qualifying identi·er.
Nested Classes
It is also possible to nest one class de·nition within the scope of another class.
This is a useful construct, which we will exploit several times in this book in theimplementation of data structures
classA: #t h eo u t e rc l a s s
classB: # the nested class
...
In this case, class Bis the nested class. The identi·er Bis entered into the name-
space of class Aassociated with the newly de·ned class. We note that this technique
is unrelated to the concept of inheritance, as class Bdoes not inherit from class A.
Nesting one class in the scope of another makes clear that the nested class
exists for support of the outer class. Furthermore, it can help reduce potential name
con·icts, because it allows for a similarly named class to exist in another context.For example, we will later introd
de·ne a nested node class to store the individual components of the list. We will
also introduce a data structure known as a treethat depends upon its own nested

5.2.2 Compact Arrays in Python

In the introduction to this section, we emphasized that strings are represented using an array of characters (not an array of references). We will refer to this more direct representation as a compact a the primary data (characters, in the case of strings).

0ASM P L E

345 12

Compact arrays have several advantages over referential structures in terms of computing performance. Most signi·cantly, the overall memory usage will be much lower for a compact structu explicit storage of the sequence of memory references (in addition to the primary data). That is, a referential structure will typically use 64-bits for the memory address stored in the array, on top o stored independently as a one-character string, there would be signi·cantly more bytes used.

As another case study, suppose we wish to store a sequence of one million, 64-bit integers. In theory, we might hope to use only 64 million bits. However, we estimate that a Python list will use four to ·ve times as much memory . Each element of the list will result in a 64-bit memory address being stored in the primary array, and an int instance being stored elsewhere in memory. Python allows you to query the actual number of bytes being used for the primary storage of any object. This is done using the getsizeof func a typical int object requires 14 bytes of memory (well beyond the 4 bytes needed for representing the actual 64-bit number). In all, the list will be using 18 bytes per entry, rather than the 4 bytes th

Another important advantage to a compact structure for high-performance computing is that the primary data are stored consecutively in memory. Note well that this is not the case for a referential structure. That is, even though a list maintains careful ordering of the sequence of memory addresses, where those elements reside in memory is not determine memory near other data that might be used in the same computations.

Despite the apparent inef·ciencies of referential structures, we will generally be content with the convenience of Python·s lists and tuples in this book. The only place in which we consider alternatives will be in Chapter 15, which focuses on the impact of memory usage on d several means for creating compact arrays of various types.