CS 234 mini-textbook

Naomi Nishimura

July 31, 2019

The course has been designed so that all necessary information will be provided either in lecture (verbally, written on the board, or presented on slides) or as extra resources posted on the course website (including this mini-textbook). For other sources of information, please see Appendix A.

Although Table 1 provides a rough correlation between lecture modules and the content of this mini-textbook, you should not assume that reading the mini-textbook is a substitute for attending lecture. Certain material, such as self-checks, will be presented in lecture only.

Background	Math overview (Appendix B); computer basics overview (Appendix C)
Module 1	User and provider roles (Section 1.2); planning and coding stages (Section 1.2); models (Section 1.5); modularity and data hiding (Section 1.1); abstract data types (Section 1.2); implementations (Section 1.4); recipes (Section 1.6)
Module 2	Case study (lecture only); choosing and modifying ADTs (Section 1.3); compound data (Section 1.3); static data (Section 7.2); ADT Multiset (Table 6.2); ADT operations (Section 6.1) and groups of ADTs (Section 6.2); Group A ADTs (Section 6.3); pseudocode (Chapter 2); worst-case, average-case, and best-case analysis (Sections 4.1 and 4.2); order notation (Section 4.3); analysis of pseudocode (Sections 4.4, 4.5, and 4.6)
Module 3	Memory (Chapter 3, Appendix C), basic data structures (Sections 7.1 and 7.3); code interfaces in Python (Section 6.8); Python modules for data structures (Appendix D); ADT Set (Table 6.3); types of data (Section 7.2); bit vector (Section 7.3)
Module 4	Case study (lecture only); Group B ADTs (Section 6.4); ADT Stack (Table 6.4); ADT Queue (Table 6.5); linked data structures (Section 7.4); ADT Indexed Sequence (Table 6.6); ADT Ranking (Table 6.7); ADT Grid (Table 6.8)
Module 5	Case study (lecture only); tree terminology (Section 5.1); Group C ADTs (Section 6.5); ADT Binary Tree (Table 6.9), implementations (Section 7.5); ADT Ordered Tree (Table 6.10), implementation (Section 7.5); ADT Unordered Tree (Table 6.11); tree traversals (Section 5.2)
Module 6	Case studies (lecture only); graph terminology (Section 5.3); Group C ADTs (Section 6.5); ADT Undirected Graph (Tables 6.12 and 6.13); data structures for graphs (Section 7.6); ADT Directed Graph (Tables 6.14 and 6.15); breadth-first search and depth-first search (Section 5.4)
Module 7	Case study (lecture only); Group D ADTs (Section 6.6); ADT Dictionary (Table 6.16); searching (Section 8.2.1); sorting (Section 8.3.1); binary search tree (Section 7.7.1); AVL tree (Section 7.7.2); (2,3) tree (Section 7.7.3)
Module 8	Case study (lecture only); Group D ADTs (Section 6.6); ADT Priority Queue (Table 6.17); heap (Section 7.8); heapsort (Section 8.3.2)
Module 9	Average-case analysis (Section B.6); interpolation search (Section 8.2.2); search in a static array (Section 8.2.3); self-organizing heuristics (Section 8.2.4); hashing (Section 7.7.4); bucket sort (Section 8.3.3); radix sort (Section 8.3.4)
Module 10	Skip list (Section 7.7.5), graph problems (Section 5.5); ADT Disjoint Set (Table 6.18); ADT Range (Table 6.19); quadtree (Section 7.9.1); k-d tree (Section 7.9.2); strings and tries (Section 7.10.1); other criteria and analysis (Section 7.11); memory hierarchy and B-tree (Section 7.10.2)

Table 1: Readings by lecture module

Contents

List of Figures

List of Tables

Chapter 1

Course goals

The goal of the course is to enable you to solve problems that involve non-trivial ways of handling data. Just as in first-year computer science courses you were taught first to use built-in functions and operations and then to design and implement your own, now that you have used built-in ways of storing and manipulating data (such as using Python lists and Python dictionaries), here you will be taught how to design and implement your own.

1.1 Modularity and data hiding

Our approach makes use of two of the most important concepts found throughout computer science, namely **modularity** (the division of a program into independent, reusable pieces) and **data hiding** (the hiding of local data from other parts of the program). In first-year computer science, modular thinking was emphasized by the use of helper functions, which made possible the decomposition of a program into multiple tasks.

When we consider modularity on a larger scale, the division of a program into pieces may enable different programmers or groups of programmers to develop those pieces independently. Just like the user of a helper function needs to know only the input, output, and side effects of the helper function (but not how it is implemented), groups focusing on different pieces of a project need to agree only on the **interfaces** between the various pieces. Here, data hiding encompasses the idea of each group knowing only what is needed for the creation of their own piece.

1.2 Roles, stages, and abstract data types

The type of interface used in the course is an **abstract data type**, or **ADT**, which consists of a specification of a type or types of data items as well as a set of operations on those items. Each operation is defined as a set of **preconditions** (requirements that must be satisfied for an operation to be guaranteed to work as specified) and **postconditions** (guarantees of outcomes when the operation is executed, such as what is produced and what the side effects are, provided that the preconditions are met). The ADT can be viewed as a contract between the **user** of the ADT and the **provider** of the ADT. As long as the user of an operation ensures that the

preconditions are satisfied, the provider guarantees that the implementation of the operation will satisfy the postconditions.

Each of the user and the provider have access only to the information that they need to know. The user does not need to know how the operations are implemented, only that they will work as specified. The provider does not need to know how the operations will be used, only how they are specified. In this way, the work can be divided between the user and the provider, each of whom can work independently. Just like the use of helper functions allows the separation of certain tasks from the main program or function, the use of an ADT allows the separation of tasks that manipulate data from other functions. Helper functions allow code reuse (the same helper function can be used in many different programs) and localized updates (one way of coding a helper function can be replaced by another without changing any other part of the code); ADTs provide the same advantages.

It is not coincidental that the term "contract" arises both in the design recipe for a program and in the description of an ADT. Choosing the type of data to use and the type of ADT to employ can be viewed as additional steps in the design recipe, that is, part of the planning stage in coming up with a solution to a problem. To make even clearer the distinction between planning and coding, we will refer explicitly to the **planning** and **coding** stages for both users and providers of ADTs.

The course will cover the roles of both users and providers, and for each, both the planning and the coding stages. For example, in the planning stage, a user will need to identify what ADT is best suited to the problem in question, either making use of a well-known ADT, adapting an ADT, or coming up with a new one. Once the user and the provider have agreed on the ADT(s) to use, the user can identify how often each of the ADT operations will be required. Operations for ADTs discussed in lecture are summarized in Chapter 6.

Meanwhile, without knowing how an ADT is to be used, the provider can investigate various implementations of the ADT, for each one choosing how the data is to be stored and how each of the operations is to be implemented. For each choice, the provider can determine the costs of each operation, and then provide the user with the information. As the user will know both the number of times each operation is used and the cost of each operation for each implementation, it is then possible to choose the best implementation for that particular situation.

1.3 Choosing, modifying, or creating an ADT

The course introduces a series of case studies; for each case study, the user determines the types of data and operations required, and then chooses, modifies, or creates an ADT that can store the data and support the operations. Although the primary focus of the lectures is the creation of common ADTs, by the end of the course the student will have mastered enough common ADTs to be able to handle a case study by choosing or modifying a known ADT. The purpose of the course is not limited to focus on a specific set of ADTs, but rather to cover both common ADTs and the types of modifications that one might make for situations in which none of the common ADTs directly apply.

In choosing which ADT to use or to modify, it is important to keep in mind the tradeoff between generality and efficiency. Depending on how the ADT might be used in the future, it might be desirable to form an ADT that stores general data (see Section 7.2 for more information

on types of data) and supports a large range of operations. However, for a situation in which one knows more about the specifics of the data and there is need for only a small number of operations, it might be better to use a more limited ADT. Typically, the more that an ADT can do, the more expensive each operation is to support.

When there is a common ADT that is similar to what is needed, it may make sense to modify the ADT. The two major types of modifications we will consider are when we need operations not included in the original ADT and when there are operations included in the original ADT that we do not need to use; we will refer to these types of changes as **augmenting** an ADT and **restricting** an ADT, respectively.

When there are ADT operations that are needed but not part of the original ADT, we will consider two possible ways of implementing the operations. In the simpler case, it may be possible to construct the desired operation out of a sequence of operations already supported by the ADT. We can view such a construction as one that can be achieved by the user of the ADT, without any knowledge of how it is implemented and without any changes to the ADT. Even in those cases where such a construction is possible, the provider can determine whether one can create a more efficient algorithm by using or modifying details of the implementation. In such a case, we will create a new ADT by augmenting the original ADT to include the new operation. One of the most common types of augmentation is when the ADT is used exclusively for **compound data** (a data item composed of two or more **fields**, possibly of different types, such as a number and a string); in such cases, there may be separate operations for accessing or updating the value for each field individually.

Less obvious is modification when we have a situation in which one or more of the original ADT operations is never used. Although it is clearly possible to choose an ADT and use only a subset of the operations, for the purposes of efficiency, it might be better to define a more restrictive ADT. In general, the cost of implementing each operation depends not only on the operation itself but on other operations supported by the ADT. The more limited the set of operations supported, the better the chance of saving time and space in the implementation. As discussed in Section 7.2, one of the most common types of restrictions is by removing any operations that change the set of data items being stored.

1.4 Implementing an ADT

The planning stage of the provider results in an **implementation** of an ADT, consisting of a way of organizing and storing the data and a set of algorithms that implement the ADT operations. An implementation can use other ADTs as well as **data structures** (ways of organizing data in the computer's memory). In the example illustrated in Figure 1.1, the provider of ADT A makes use of both an ADT (B) and a data structure (C) to implement A. The provider of ADT A does not need to know the details of how ADT B is implemented, only how the ADT operations for A are implemented using the ADT operations for B. Thus, the provider of one ADT (A) may play the role of user of another ADT (B). Another common use of an ADT within another ADT is when using compound data; in this case, an ADT can be introduced for the type of data, allowing access and updating of any of the fields individually.

We can view the levels of abstraction as occurring in a sequence of layers, with the top, most abstract, layer representing the problem being solved and the interface between each pair

of consecutive layers represented by an ADT. There can be any number of layers of ADTs, as long as each bottom layer specifies a data structure and a set of algorithms that implement the operations of a bottommost ADT. (In Figure 1.1, the bottom layers are the data structures C and D, where D implements the bottommost ADT B.) When specifying a data structure, we are not concerned with where exactly in memory the data items are stored, but rather whether they are to be accessed directly (as in the case of **contiguous memory**) or whether one or more links need to be followed (as in the case of **linked memory**). (These terms are discussed in greater detail in Chapter 3.)

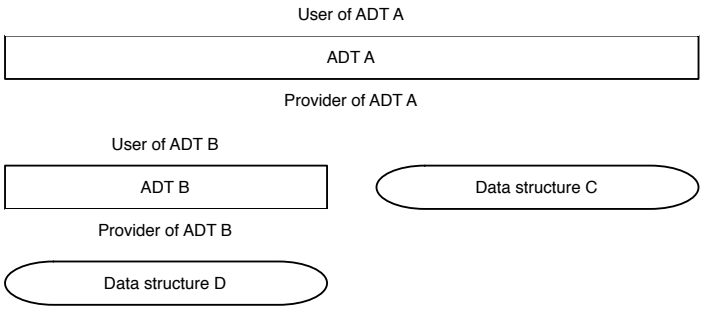

Figure 1.1: Implementation of an ADT using another ADT

Although various sources use inconsistent definitions of the term **data structure**, in this course we use the term to refer only to the way data is organized in memory, without specifying how the ADT operations are implemented. In fact, the same data structure can be used not only to implement different ADTs, but also to provide different implementations of the same ADT. Thus, for a single ADT, a given data structure may be used in multiple, differing implementations, each of which may use different algorithms to implement the ADT operations.

1.5 Models and analysis

In order to determine which implementation of an ADT is most suitable, one could code each implementation, time the execution of operations on representative data, and then compare the results. Such an approach is not only time-consuming and labour-intensive, as it requires the resources to code a potentially large number of options, but the usefulness of the results depends on how close the data, programming language, and machine are to those which are going to be used.

A more time-efficient and generally-applicable approach is to find a way of comparing options without coding and without making any assumptions about the data, programming language, or machine. In this approach, analysis makes use of a **model of computation**, a mathematical definition that captures the essence of computation, with costs based on a **memory model**, which in turn captures the essence of memory use.

We will use a model of computation that represents the computation using a single processor, in which simple tasks (such as simple mathematical and Boolean operations and comparisons)

can be viewed as all taking roughly the same amount of time. For those who have not studied the topic before, an introduction to the basics of computers is provided in Appendix C; this material provides the background underlying the memory models used in the course (Chapter 3) as well as the model of computation. Instead of explicitly defining the model, we will make use of **pseudocode** to both express and analyze our algorithms (Chapter 2).

1.6 Recipes for users and providers

The following "recipes" summarize the steps undertaken by the user and provider in the planning and coding stages:

Recipe for user/plan (solving a problem):

1. Determine types of data and operations.
2. For each type, choose/modify/create an ADT.
3. Develop a pseudocode algorithm using ADT operations.
4. Calculate the algorithm's cost with respect to costs of operations.
5. Using information from the provider, choose the best option.

Recipe for provider/plan (choosing among implementations):

1. Create pseudocode of various options for data structures and algorithms to implement the ADT and its operations.
2. Analyze the cost of each operation for each implementation.
3. Provide options for packages of operation costs.

Recipe for user/code (code solution using ADT operations):

1. Agree on the code interface.
2. Code a solution to the problem using the ADT.

Recipe for provider/code (code implementation of ADT operations):

1. Agree on the code interface.
2. Code the chosen data structure and algorithms implementing the ADT.

1.7 Outline

To establish the assumptions underlying our assessment of various options, in Chapter 2 we discuss how we model computation (pseudocode), and in Chapter 3 we discuss how we model the use of memory (memory models). Then, Chapter 4 covers methods for analysis (worst case, best case, and average case) as well as the use of order notation to express space and time costs.

In order to be able to discuss ADTs and data structures (Chapters 6 and 7), we give background on trees and graphs in Chapter 5. Details on algorithms that implement operations are covered in Chapter 8.

Chapter 2

Pseudocode

In lectures, algorithms will often be expressed in pseudocode, a mixture of code and English. While understanding pseudocode is usually not difficult, writing it can be a challenge.

One example of pseudocode, used in this course, is presented in Section 2.2. Appendix E contains examples of pseudocode found in various textbooks.

2.1 Guidelines for writing pseudocode

Why use pseudocode at all? Pseudocode strikes a sometimes precarious balance between the understandability and informality of English and the precision of code. If we write an algorithm in English, the description may be at so high a level that it is difficult to analyze the algorithm and to transform it into code. If instead we write the algorithm in code, we have invested a lot of time in determining the details of an algorithm we may not choose to code (as we typically wish to analyze algorithms *before* deciding which one to code). The goal in writing pseudocode, then, is to provide a high-level description of an algorithm which facilitates analysis and eventual coding (should it be deemed to be a "good" algorithm) but at the same time suppresses many of the details that vanish with asymptotic notation (discussed in great detail in Section 4.3). Finding the right level in the tradeoff between readability and precision can be tricky. If you have questions about the pseudocode you are writing on an assignment, please ask one of the course personnel to look it over and give you feedback (preferably before you hand it in so you can change it if necessary).

Just as a proof is written with a type of reader in mind (hence proofs in undergraduate textbooks tend to have more details than those in journal papers), algorithms written for different audiences may be written at different levels of detail. In assignments and exams for the course, you need to demonstrate your knowledge without obscuring the big picture with unneeded detail. Here are a few general guidelines for checking your pseudocode:

1. Mimic good code and good English. Using aspects of both systems means adhering to the style rules of both to some degree. It is still important that variable names be mnemonic, comments be included where useful, and English phrases be comprehensible (full sentences are usually not necessary).

2. Ignore unnecessary details. If you are worrying about the placement of colons, you are using too much detail. It is a good idea to use some convention to group statements (begin/end, brackets, or whatever else is clear), but you shouldn't obsess about syntax.

3. Don't belabour the obvious. In many cases, the type of a variable is clear from context; unless it is critical that it is specified to be an integer, for example, it is often unnecessary to make it explicit.

4. Take advantage of programming shorthands. Using branching or looping structures is more concise than writing out the equivalent in English; general constructs that are not peculiar to a small number of languages are good candidates for use in pseudocode. Using parameters when defining functions is concise, clear, and accurate, and hence should be included in your pseudocode.

5. Consider the context. If you are writing an algorithm for mergesort, the statement "Use mergesort to sort the values" is hiding too much detail; if we have already studied mergesort in class and later use it as a subroutine in another algorithm, the statement would be appropriate to use.

6. Don't lose sight of the underlying model. It should be possible to "see through" your pseudocode to the model below; if not (that is, you are not able to analyze the algorithm easily), it is written at too high a level.

7. Check for balance. If the pseudocode is hard for a person to read or difficult to translate into working code (or worse yet, both!), then something is wrong with the level of detail you have chosen to use.

2.2 Pseudocode used in the course

The pseudocode used in the course will adhere to the following conventions:

- Variable names are capitalized, and function names are written in all capital letters. Where helpful for readability, such as to separate words, an underscore (_) is used.

- To make it distinguishable from code, which is presented in `typewriter font`, pseudocode is presented using *italics*. (The one exception is function names, which may be written with or without italics.)

- Simple Python list operations, such as *[]* for an empty list or accessing an item in a position, will be included, but no powerful operations, such as `sorted` and `reverse`.

- The Python list operation slice can be used, with the indices having the same meaning as in Python: *[a:b]* consists of items a up to but not including b, where if a is omitted the slice starts at the first item in the list and if b is omitted the last item in the slice is the last item in the list. The operation can also be used for strings.

- The Python method `len` will be written as LENGTH for consistency with other names.

- Each function definition contains a preamble consisting of the name of the function and parameters, the input, the output (if any), and side effects (if any).
- Function applications are given as the name of the function with all parameters appearing afterwards in parentheses, separated by commas. For consistency, instead of using dot notation for methods, the object appears as one of the inputs. For example, to determine the length of list L we would write LENGTH*(L)*.
- Reserved words (such as ***for*** and ***if***) are shown in boldface.
- Assignment statements are shown using $\leftarrow$, like in Example 3 (see Appendix E for pseudocode examples). Other common conventions are the use of = (Example 4) and := (Example 2).
- Types of variables are not listed explicitly, unlike in Examples 2 (which uses `integer`) and 4 (which uses `int`). Ideally, the code will be written in such a way that the type is clear from context, or from the listing of inputs and outputs.
- Indentation is used to group statements, like in Python.
- The word ***return*** is used to indicate that a value is returned.
- Branching mimics Python; thus, ***if*** and ***else*** are used, but unlike in Examples 1–3, not ***then***.
- In ***for*** loops, a loop that goes from the value *Start* to the value *Finish* will be written ***for*** *Count* ***from*** *Start* ***to*** Finish, where *Count* can be replaced by a variable of another name. The comparable Python code would be `for count in range(start, finish+1)`.
- For a ***for*** loop over a list or other collection, ***for each*** is used.
- For the ease of analysis, almost always a line will contain a finite number of simple tasks (cost $\Theta(1)$) or a function application.

As you can see from the example of binary search below, the pseudocode is quite close to Python.

```
BIN_SEARCH(L, Item)
INPUT:     A list L with items in nondecreasing order
OUTPUT:    True or False depending on whether Item is in L
if LENGTH(L) ≤ 1
    if LENGTH(L) == 1 and L[0] == Item
        return True
    else
        return False
else
    Mid = FLOOR(LENGTH(L) / 2)
    if L[Mid] == Item
```

```
    return True
else if L[Mid] > Item
    return BIN_SEARCH(L[:Mid], Item)
else
    return BIN_SEARCH(L[Mid+1:], Item)
```

Chapter 3

Memory models

Just like a model of computation is used to give a simplified view of the steps taken during computation, a memory model is a simplified representation of how computer memory is organized and accessed. In the planning stage, the provider of an ADT will use a memory model to specify how the data is to be stored. Our representations are intended not to be exact and accurate depictions of what happens in a computer, but to provide sufficient information for us to be able to compare different ways of structuring data.

In your previous study of computer science, you may have been exposed to a variety of different memory models, each used to illustrate a particular concept. For example, when students are first taught about variables, the memory allocated to a variable is sometimes represented as a box labeled by an **identifier** (the name of the variable). When a variable is created, the box is drawn and the label attached, and when the variable is initialized, the initial value is placed in the box (Figure 3.1). Thereafter, if the value of the variable is changed, the value inside the box is changed.

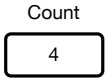

Figure 3.1: A simple memory model

The model mentioned above is consistent with the way Python manages identifiers in **namespaces** (mappings between identifiers and addresses). As you've experienced in your programming in Python, there are various namespaces used, including namespaces that are local to functions or modules, as well as a global namespace. Using `from math import *` puts all of the functions in the math module into the global namespace. Typically we instead use `import math` so that the application of a function in the module requires that the name of the function be appended with `math.` to distinguish it from any new function of the same name.

Often a different memory model is introduced to support the concept of **aliasing** of mutable data such as Python lists. Instead of storing a value in a box labeled by an identifier, now an identifier labels an arrow that points to a box storing data. When the same data can be accessed using two different identifiers, there are two arrows, one for each identifier, pointing to the same data. Similar ideas are used to represent built-in data types; for example, a list may appear as a sequence of boxes, each accessed using the index of the particular item (Figure 3.2).

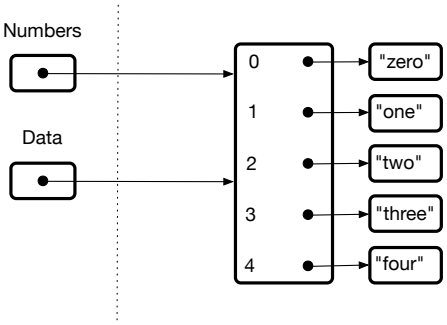

Figure 3.2: A memory model depicting a list

Here, too, the model is consistent with Python (though not necessarily all other languages). Just as in Python, an immutable data type, such as a tuple, can contain within it a mutable data type, such as a list.

For this course, we will make use of memory models that allow us to understand the relative costs of choosing different ways of arranging data. Rather than be concerned about where in memory our data is stored, we will be concerned with how easy it is to find our data, access it, and change it. In much of our analysis, we will focus on two main ways to allocate memory for a group of related data items, all of the same type. (If you have not already done so, you may wish to read Appendix C now.)

In **contiguous** memory, all items in the group are stored in one chunk. A chunk has the advantage that if you can find the location of the first cell in the chunk, you can easily find the cells in the rest of the chunk. The memory for one item can be seen as a **subchunk** (or **slot**) of the chunk for the group, where the address of the first cell in each subchunk can be easily calculated from the address of the entire chunk.

In contrast, in **linked** memory, multiple chunks of memory are used, typically one chunk for each item in a group. The name comes from the links that are used by a program to navigate through the data: each chunk will store not only one or more pieces of data, but also one or more addresses of other chunks. The term **pointer** is used for a type of data that represents an address; in illustrations, typically pointers are depicted as arrows (and empty, or **null** pointers, are depicted by diagonal lines), allowing us to represent addresses as locations in an image rather than as numbers. In addition, we can assume that there is a way to associate an identifier with an address. In our example of aliasing, for two variables to be associated with the same data item, the two variables are associated with the same address (that is, the address of the first cell in the chunk in which the data item is stored).

We will not be concerned with the role of the operating system, other than to expect that when we write a program that uses memory, the operating system will handle the job of finding a free chunk of memory of the correct size, and that when we are no longer using that memory, the operating system will reclaim the memory for later reuse.

Due to the nature of memory handling, if no processing is done to initialize newly-allocated memory, reading memory before writing to it may result in the reading of a value that had been stored in the memory when it was being used by a previous process. To avoid this problem, at times we will opt to initialize a chunk of memory to empty. For each data structure that we use

in the course, we will specify whether or not we opt for this type of initialization.

As discussed in Appendix C, the costs of accessing memory in various levels of the hierarchy may differ. In this course, we will consider two different models. For most of the course, we will ignore such differences, making use of a one-level model. At times we will expand our focus to a two-level model in which data is moved between main memory and external memory, and where data structures are designed based on the size of a page. More complex models are also possible, but will not be considered in this course.

Chapter 4

Analysis

Throughout the course, we will be considering various implementations of ADTs, where each implementation consists of the choice of how to store the data (using one or more data structures and/or ADTs) and the choice of algorithms for each of the ADT operations. By assigning time and space costs to various aspects in our model of computation (Chapter 2) and our memory model (Chapter 3), we will be able to analyze various options without having to code them all.

In our analysis, we have not specified particular software or hardware. We will categorize costs of options into groups, where differences between groups are so wide that small changes resulting from particular choices of programming languages or machines will not alter the ways options are categorized.

In the remainder of this chapter, we explore how we can form groups by representing costs in a succinct way.

4.1 Operations with costs that vary

Whether we are determining the amount of space used by a data structure or the amount of time used by an algorithm, the resources required will often vary depending on the particulars of the data being stored. For example, if an operation entails examining all possible locations in a data structure where data might be stored, then the cost of the operation will depend on the size of the data structure. If instead the operation entails examining all of the data items being stored, ignoring empty locations, then the cost of the operation will depend on the number of data items. At times, the cost of an operation may depend not only on the size of the data structure or the number of data items, but also specific values and locations of data items.

In order to be able to compare multiple options, for each option we need to find a succinct way of representing the required resources, rather than including detailed information about each possible situation that might occur. At the same time, we wish to include enough detail that we have an overall picture of the behaviour of a particular algorithm. Accordingly, we will discuss the running time and space use in terms of quantities such as the size of an input (often represented as n), the size of a data structure, and the number of data items stored in a data structure at the time of an operation. In most of the course, we will assume that the number of bits used to store a single item of simple data (such as a number, string, Boolean, or pointer) is a constant; as a consequence, the number of data items is used as a way of measuring the size

of an input.

4.2 Representing cost as a function

For the time being, we represent each cost as a function of a single variable n; we will expand our analysis to additional variables in Section 4.6. In order to represent the running time of an algorithm as a function f, we need to determine a value $f(n)$ for each possible input size n. For convenience, we can express the function as if it were defined on all nonnegative values of n, even though the only values of n that represent input sizes are nonnegative integers.

If the running time of an algorithm is the same on all inputs of a fixed size n, then $f(n)$ is defined to be that running time. If instead the running time is not identical for all inputs of a fixed size n, we can still define a function f to represent running times by choosing a value $f(n)$ that represents the range of running times of inputs of size n. In doing so, we will end up throwing away some of the details of the differences in costs. Our goal is to choose a representative that captures enough of the behaviour to help us select among options.

There are different ways of choosing representatives, each of which results in a different type of running time. We obtain **best-case running time** if we choose as $f(n)$ the smallest possible running time for any input of size n; analogously, if we choose as $f(n)$ the largest possible running time for any input of size n, we obtain **worst-case running time**. Each of these choices ignores all values larger (respectively, smaller) than the one chosen.

Another way of determining $f(n)$ is to take into account how likely each input of size n is to occur. More precisely, a **probability distribution** is an assignment of **probabilities** to each possible input, where the sum of all the probabilities equals 1. (Further discussion of probability can be found in Appendix B.6.) The **average-case running time** with respect to a particular probability distribution is found by setting $f(n)$ to equal $\sum_{i \in I} Pr[i] \cdot t(i)$ where I is the set of inputs of size n, $Pr[i]$ is the probability of input i, and $t(i)$ is the running time on input i.

Unfortunately, the average-case running time is only useful when the probability distribution accurately reflects the type of data on which the algorithm is to be run. As it is rare that such distributions are easy to obtain, in many cases the average case is not used.

The most common type of running time used in analysis is worst-case running time. Although it might seem unduly pessimistic, it has the advantage that it provides a guarantee: the actual running time will never be worse than the worst-case running time. When we have more information about the input, we will at times discuss average-case running time. The main use of best-case running time will be on assignments and exams, to test your understanding of the concepts.

Remember that when we refer running time, we are referring to a function of an input size n. In particular, this means that we cannot make assumptions about the value of n, such as assuming that $n = 1$ for a best-case analysis.

Note: Please do not use the term *runtime* interchangeably with **running time**. The former is more accurately used to describe when a program is being run, such as in the phrase "runtime stack."

4.3 Categorizing functions into groups

Although our motivation for categorizing functions into groups is to be able to compare worst-case running times of algorithms, it is important to keep in mind that the discussion in this section applies to any function at all. It could express worst-case, best-case, or average-case running time, or it could be a function that has nothing to do with running times at all.

We will be categorizing functions using **order notation**, also known as **asymptotic notation**. (We will use the terms interchangeably.) The use of the word "asymptotic" reflects the fact that in this notation, we are grouping functions into sets, based on how the functions behave in the long term, as the value of n increases. Just like numbers might be grouped by "order of magnitude," here we group functions into categories that are significantly different from each other, such as "constant time," "linear time," and "exponential time" (all as a function of n).

In this course we will use three types of notation, O, Ω, and Θ, each of which defines a set with respect to a simple function on a single variable. (Discussion of order notation on simple functions of multiple variables can be found in Section 4.6.) We consider a function $f(n)$ on a single variable to be simple if it consists of a single term (that is, it is not of the form $f(n) = g(n) + h(n)$) and is not the product of a function and a constant (that is, it is not of the form $f(n) = cg(n)$ for any constant c). We will see shortly how such simple functions are sufficient to describe all the groups of interest. Note: We will use names such as $f(n)$, $g(n)$, and $h(n)$ for functions playing various roles in expressions. Whenever a function appears in order notation, it will be a simple function as described above. Typical simple functions we will see are 1 (for constant-time costs), $\log n$, n, $n \log n$, n^2, n^3, and 2^n).

The set $O(f(n))$ contains all functions that can be bounded above by a constant multiplied by $f(n)$ for all values of n that are sufficiently large. More formally, a function $g(n)$ is in $O(f(n))$ if $g(n) \leq c \cdot f(n)$ for all values of $n \geq n_0$, where $c > 0$ and $n_0 \geq 1$ are both constants. Analogously, the set $\Omega(f(n))$ contains all functions that can be bounded below by a constant multiplied by $f(n)$ for all values of n that are sufficiently large. More formally, a function $g(n)$ is in $\Omega(f(n))$ if $g(n) \geq c \cdot f(n)$ for all values of $n \geq n_0$, where $c > 0$ and $n_0 \geq 1$ are both constants. Notice that in both definitions, we are not required to use specific values for c and n_0; all we need to do is to ensure that some such values exist.

There are a few immediate observations we can make as a consequence of these definitions. If we have two simple functions $g(n)$ and $h(n)$ such that $g(n) < h(n)$, then it is easy to see that if $f(n) \in O(g(n))$, then $f(n) \in O(h(n))$ (that is, since $f(n) \leq c \cdot g(n) \leq c \cdot h(n)$ for all $n \geq n_0$), and that if $f(n) \in \Omega(h(n))$, then $f(n) \in \Omega(g(n))$.

Omitting other types of notation, such as o and ω, that we will not be using in this course, the last remaining notation we will introduce is $\Theta(f(n))$. A function $g(n)$ is in $\Theta(f(n))$ if it is in both $O(f(n))$ and $\Omega(f(n))$. We observe from the above discussion that although a function can be in $O(g(n))$ as well as $O(h(n))$ (or in $\Omega(g(n))$ as well as $\Omega(h(n))$) for two different functions $g(n)$ and $h(n)$, it can be in $\Theta(f(n))$ for only one simple function $f(n)$. Consequently, when we are trying to give as much information as possible about a function, we will use Θ notation. If we are using O notation, we will try to express a function as in $O(f(n))$ for as small a function $f(n)$ as possible (since the larger the $f(n)$, the larger the set, and the less we are saying about our function), and similarly, we will try to express a function as in $\Omega(f(n))$ for as large a function $f(n)$ as possible.

To see why we can express all the groups that we need using simple functions, we first observe

that since we can use any constant c, clearly any multiple of $f(n)$ is in $O(f(n))$. Moreover, when we have a function $f(n)$ that consists of several terms, we can show that $f(n) \in \Theta(g(n))$, where $g(n)$ is the simple function that appears in the dominant term. For example, if $f(n) = 3n^2+10n+6$, we can show that $f(n) \in \Theta(g(n))$ by demonstrating that $f(n)$ is bounded below by $3n^2$ for all n (and hence is in $\Omega(n^2)$) and bounded above by $19n^2 = 3n^2+10n^2+6n^2 \geq 3n^2+10n+6 = f(n)$ for all $n \geq 1$ (and hence is in $O(n^2)$).

Please be very careful not to confuse the idea of a lower bound on a function (as represented by Ω) with the choice of smallest values as representatives for best-case running time, nor to confuse the idea of an upper bound on a function (represented by O) with the choice of largest values as representatives for worst-case running time. Remember that both Ω and O can be used to describe any function at all, whether worst-case, best-case, or average-case running time, or a function that does not represent any kind of running time.

4.4 Using order notation to analyze pseudocode

If you read about order notation in various textbooks, you are likely to encounter exercises that ask you to express a complicated function in order notation. In this course, the way we will use order notation is to remove complications before expressing functions. In particular, since we will typically be using order notation to analyze pseudocode, we will simplify functions into order notation as we go instead of analyzing the cost in detail in the first place.

Our typical use of order notation will be in the analysis of pseudocode, where we assign order notation to the cost of each line or block of lines. If an algorithm can be decomposed into a sequence of blocks of lines, then we can analyze the entire algorithm by analyzing each block. Moreover, since the cost of the sum of several terms is dominated by the cost of the highest term, we can ignore all but the dominant term, as discussed in Section 4.3. Similarly, we can express the cost of any constant number of constant-time operations as in $\Theta(1)$.

For branching, we will make use of the fact that we are determining worst-case costs, using the most expensive (worst case) branch in our calculations. For looping, the total cost will be the sum of the costs of all the loop bodies. When each loop body has the same cost, we can simply multiply the number of iterations by the cost of a loop body. More complex analyses are discussed in lecture.

4.5 Constant-time operations

Throughout the course, we will be analyzing pseudocode based on assumptions about the costs of various operations. In this section, various operations with constant-time costs are listed.

As a default, we will handle numbers and Booleans as entities of a constant size, and view operations on them as taking constant time. Such operations include common mathematical operations (such as $+$, $-$, $\backslash$, and $*$), Boolean operations (such as `and`, `or`, and `not`), and comparisons (such as $<$, $>$, $<=$, $>=$, $==$, and $!=$).

Similarly, because their costs do not depend on the size of inputs, we assume constant-time worst-case costs for such steps as assigning a value to a variable, using a variable, and using ***return*** to return a value.

Strings will be handled in different ways in different situations. For data structures that manipulate strings, the length of a string will be viewed as a part of an input. If instead a function produces a string such as *"Not found"*, producing the string will be seen as a constant-time operation.

If you are unsure about the cost of an operation that you are using, please consult course personnel.

4.6 Using multiple variables

We can also define order notation with respect to multiple variables, for situations in which values depend on various types of information, such as both the number of edges (m) and the number of vertices (n) in a graph. For general graphs, we do not know whether m or n is bigger, and hence we cannot simplify our representation to use a single variable. Consequently, a simple function on two variables can contain more than one term, provided that none of the terms dominates any of the others. Thus, a simple function could not contain both mn and mn^2, since mn^2 dominates mn, but it could contain both m^2n and mn^2, as either one might be the dominant term.

Simplifying a function is more complicated when we have additional information about the variables being used. For example, if we are calculating a function based on both the size of a data structure (n) and the number of data items stored (k), we know that $n \geq k$. In this case, a simple function cannot contain both n^2 and k^2 as terms, as the former dominates the latter. However, a simple function could have the terms n^2 and k^3, as we do not know which one is dominant.

Chapter 5

Trees and graphs

We will make use of more complex structures, such as trees and graphs, to form more complex ways of relating information in ADTs. Although trees can be viewed as special types of graphs, we introduce slightly different terminology for trees and graphs.

5.1 Tree terminology

In the trees we consider in this course, each tree consists of a set of **nodes**, one of which is a special node called the **root**. Although there exist unrooted trees, we will rarely, if ever, consider them in this course. In addition, when viewed as a graph, we could use the term **vertex** instead of **node**. We may opt to use the term **tree node** when it is necessary to make a distinction between a node in a tree and a node in a linked data structure (Section 7.3).

A connection between two nodes is an **edge**; each edge is between a **parent** and a **child**. If we draw a tree by putting the root at the top of the page and drawing each edge more-or-less vertically, then the parent is the node at the top of the edge (closer to the root) and the child is the node at the bottom of the edge (farther from the root).

As a node can have more than one child, we use the term **children** as the plural of **child**. The set of nodes that have the same parent as u are called the **siblings** of u. Working downwards in a tree, for any node u we call its children, the children of its children, and so on, the **descendants** of u. The tree formed from a node u and all its descendants (including the edges that join them) is the **subtree rooted at** u.

Although a node can have any number of children, it can have at most a single parent, the only node without a parent being the root of the tree. If we trace back from a node u to its parent, the parent of its parent, and so on, we will eventually reach the root of the tree. The nodes visited along the way form the **ancestors** of u. Although differing definitions are used, in this course we will assume that a node is neither its own descendant nor its own ancestor.

Two other useful pieces of terminology make a distinction between nodes that have no children (**leaves**) and nodes that have at least one child (**internal nodes**).

We will distinguish between different types of trees, depending on extra information that is provided. In the definitions given so far, we have not defined an order on the children of a node; each tree we have defined is an **unordered tree**. If instead we specify an order for the children of each node, we have an **ordered tree**.

By adding further restrictions, we can define a **binary tree** in which each node has at most two children, where each child is either a **left child** or a **right child**. When a node has two children, it has both a left child and a right child; if it has one child, it is either a left child or a right child. The subtree rooted at the left child of u (if it exists) is the **left subtree** of u and the subtree rooted at the right child of u (if it exists) is the **right subtree** of u.

A key property of trees is that there is exactly one sequence of edges connecting any pair of nodes; we can describe this sequence in terms of a **path** between nodes v_0 and v_k, which is a sequence of nodes $\{v_0, \ldots, v_k\}$ such that $\{v_i, v_{i+1}\}$ is an edge for all $0 \leq i < k$. A path is **simple** if each node appears at most once in the sequence. We define the **length** of a path to be the number of edges in the path. The **depth** of a node is the number of edges in the path from the node to the root; the depth of the root is thus 0. We refer to all the nodes of a particular depth as being on the same **level**. To determine the **height** of a node u, we instead consider the maximum number of edges in any path from the node to leaf in the subtree rooted at u. Because a subtree rooted at a leaf contains only the leaf itself, each leaf has height 0. The **height of a tree** is the height of the root of the tree.

There are special terms used to describe special types of binary trees. In a **perfect** binary tree, each node has either zero or two children and all leaves have the same depth. As a consequence, for a perfect binary tree of height h, each level $\ell < h$ contains 2^ℓ internal nodes and level h contains 2^h leaves.

In a **complete** binary tree of height h, each level $\ell < h$ contains 2^ℓ nodes and level h contains at most 2^h nodes, situated as far to the left as possible. More formally, if we consider the nodes at level $h-1$ in order from left to right as $u_0, u_1, \ldots, u_{2^{h-1}-1}$, then for some u_j, $0 < j$, each u_i has two children for $i < j$, each u_k is a leaf for $j < k$, and u_j has either two children or just a left child. By this definition, every perfect tree is a complete tree (where $j = 2^{h-1} - 1$ and u_j has two children), but not every complete tree is a perfect tree. In particular, in a complete tree, there can be leaves on levels $h-1$ and h whereas in a perfect tree there are only leaves on level h.

5.2 Tree traversals

In order to search for data stored in a tree, we make use of different ways of traversing all the nodes in a tree. The traversals are described here in general terms that allow the existence of multiple possible orderings of a given type; in practice a particular implementation will result in one specific ordering. For example, the order in which the children of a vertex are stored, even in an unordered tree, may have an impact on how an algorithm orders the nodes.

The first three types of traversal orders are applicable to any tree; for an unrooted tree, any node can be used to serve the role of the root in the descriptions that follow. In a **postorder traversal**, each node appears after its children in the traversal, and in a **preorder traversal**, each node appears before its children. In a **level order traversal**, nodes appear in increasing order of depth. As noted above, a particular implementation may result in a particular way of breaking ties among children in a postorder or preorder traversal and among nodes at the same level in a level order traversal.

An additional type of traversal order is applicable only to binary trees; in an **inorder traversal**, any node will appear in the ordering after all the nodes in its left subtree and before

all the nodes in its right subtree.

5.3 Graph terminology

A **graph** G consists of a set $V(G)$ of **vertices** and a set of $E(G)$ of **edges**. In this course, we will consider two types of graphs that differ in whether or not edges are directed from one vertex to another; in the directed case, we'll often use the term **arc** instead of **edge**. Thus, in an **undirected graph**, each edge is an unordered pair of vertices $\{u, v\}$ and in a **directed graph**, each arc is an ordered pair of vertices (u, v); u and v are the **endpoints** of the edge or arc. Unless stated otherwise, we will assume that the graphs are **simple**, which means the two endpoints must be distinct vertices, there can be at most one edge connecting a pair of vertices in an undirected graph, and there can be at most one arc connecting a given vertex to another given vertex in a directed graph (there can be at most one arc (u, v) and at most one arc (v, u)).

For either undirected or directed graphs, we may wish to store extra information with a vertex, edge, or arc. Such information can consist of various labels or colours, or numbers, such as **weights**.

5.3.1 Terminology for undirected graphs

Various terms are used to describe relationships among vertices and edges. In an undirected graph, two vertices u and v are **adjacent** if there exists an edge $\{u, v\}$. For relationships between edges or between edges and vertices in an undirected graph, we use the term **incident**: an edge and each of its endpoints are incident, and two edges are incident if they share an endpoint.

We frequently wish to determine the surroundings of a particular vertex. The set of vertices adjacent to u is the set of its **neighbours**; collectively, the set of neighbours is the **neighbourhood** of u. The size of the neighbourhood of u (or equivalently, the number of incident edges) is the **degree** of u. In a simple graph, the degree of a vertex can be at most one less than the number of vertices in the graph. A vertex of degree zero is an **isolated vertex**.

More generally, we wish to traverse a sequence of edges in a graph. A **path** between vertices v_0 and v_k is a sequence of vertices $\{v_0, \ldots, v_k\}$ such that $\{v_i, v_{i+1}\} \in E(G)$ for all $0 \leq i < k$; a path is **simple** if no vertex is repeated. We form a cycle by joining the first and last vertex in a path; more formally, a **cycle** is a sequence of vertices $\{v_0, \ldots, v_k\}$ such that $\{v_i, v_{i+1}\} \in E(G)$ for all $0 \leq i < k$ and $\{v_k, v_0\} \in E(G)$. A graph without a cycle is **acyclic**.

In certain cases, there is a path between any pair of vertices in the graph; such a graph is **connected**.

5.3.2 Terminology for directed graphs

For an arc (u, v), the vertex u is the **tail** of the arc and the vertex v is the **head** of the arc; an arc is typically drawn as an arrow directed from u to v. In a directed graph, vertices connected by an arc are categorized as the **in-neighbours** of u (vertices v for which the arc (v, u) exists) and **out-neighbours** of u (vertices v for which the arc (u, v) exists). The numbers of in-neighbours and out-neighbours are known, respectively, as the **indegree** and **outdegree** of u.

A **directed path** is a sequence of vertices $\{v_0, \ldots, v_k\}$ such that $(v_i, v_{i+1}) \in E(G)$ for all $0 \leq i < k$.

5.4 Breadth-first search and depth-first search

The algorithms **breadth-first search** and **depth-first search** are employed not only to determine the set of vertices reachable from a given input vertex, but also as templates for various algorithms that process such vertices. Here we consider only the basic forms of the two algorithms. Each can be modified to solve various problems, typically by adding extra steps associated with each vertex or edge either when it is first encountered or right before the last time it is visited.

As the names imply, the two algorithms use different strategies for determining which node to list (or process) first. In **breadth-first search**, vertices are processed in order of increasing distance from the input vertex, where the distance can be seen as the shortest path from the input vertex to the vertex in question. Thus, after the input vertex, first all neighbours of the input vertex are listed, then all neighbours of neighbours of the input vertex that are not also neighbours of the input vertex, and so on. In contrast, vertices are ordered by **depth-first search** by seeking those farthest from the input vertex first, where a farthest vertex can be found by following a path until it is not possible to go further without revisiting an already-visited vertex. Details of both algorithms follow in the next sections.

For each algorithm, we make use of a colouring of the vertices to indicate progress by the algorithm. Initially, all vertices are white, indicating that they have not yet been visited. When a vertex has been visited (but not necessarily listed or fully processed, as in the case of depth-first search), it is coloured gray. Finally, when all processing on the vertex has been completed, it is coloured black. The colours are used to help organize each search.

In the same way that there were multiple orderings for tree traversals, as discussed in Section 5.2, our descriptions of the two algorithms are given in a general form that allows multiple possible orderings. The exact order of vertices, for either of the two algorithms, will depend on a particular implementation (and, in particular, the order in which the neighbours of a vertex are processed).

5.4.1 Breadth-first search

In breadth-first search, a queue (initially containing the input vertex) is used to manage the vertices being processed. At each iteration, the first vertex in the queue is removed, its neighbours are found and added to the queue, and then the vertex itself is added to the ordering. In the pseudocode algorithm (Figure 5.1), the ADT Undirected Graph has been augmented with operations Vertex_Colour*(G, One)* (which produces the colour of vertex *One*) and Set_Vertex_Colour*(G, One, New)* (which sets the colour of vertex *One* to *New*), and it is assumed that all vertices are initialized to white.

```
BFS(G, Start)
INPUT:      A graph G and a vertex Start in G; all vertices are white
EFFECTS:    Visits all vertices reachable from Start; all such vertices are black
1   Q ← Create_Queue()
2   Enqueue(Q, Start)
3   Set_Vertex_Colour(Q, Start, gray)
4   Optional steps placed here
5   while not Is_Empty_Queue(Q)
6       Current ← Dequeue(Q)
7       for Nbr in Neighbours(G, Current)
8           if Vertex_Colour(G, Nbr) == white
9               Enqueue(Q, Nbr)
10              Set_Vertex_Colour(G, Nbr, gray)
11      Set_Vertex_Colour(G, Current, black)
12      Optional steps placed here
```

Figure 5.1: Breadth-first search pseudocode template

5.4.2 Depth-first search

Just like the vertices in breadth-first search can be seen as being stored in a queue for processing, those in depth-first search can be seen as being stored in a stack. Typically this does not entail using an explicit stack, but instead relies on the implicit stack inherent in recursive algorithms. As in the algorithm for breadth-first search, in Figure 5.2 we augment the ADT Undirected Graph with operations Vertex_Colour*(G, One)* and Set_Vertex_Colour*(G, One, New)*, and assume that all vertices are initialized to white.

```
DFS(G, Start)
INPUT:      A graph G and a vertex Start in G; all vertices are white
EFFECTS:    Visits all vertices reachable from Start; all such vertices are black
1   Set_Vertex_Colour(G, Start, gray)
2   Optional steps placed here
3   for each Nbr in Neighbours(G, Start)
4       if Vertex_Colour(G, Nbr) == white
5           DFS(G, Nbr)
6   Set_Vertex_Colour(G, Start, black)
7   Optional steps placed here
```

Figure 5.2: Depth-first search pseudocode template

5.5 Problems on graphs

In the course, various problems on graphs are discussed, though not necessarily in detail, and not necessarily with solutions provided. As discussed in class, they are introduced in part to motivate the use of multiple ADTs together to solve a problem, and in part to motivate taking a course on algorithm design, such as CS 231.

Chapter 6

ADTs

6.1 Introduction to ADTs

Throughout the course, various ADTs will be introduced and discussed. Perhaps even more than the ADTs themselves, it is crucial to understand the following underlying principles:

- ADTs provide a way of separating manipulation of data from other steps in an algorithm.
- Identifying when an ADT can be used entails determining data types and operations.
- When selecting among ADTs, using the ADT with the most restrictive operations is likely to result in one requiring the fewest resources.
- Understanding common ADTs allows one to make slight modifications for special situations.

Since the goal of the course is to enable students to both select and modify known ADTs, many of the ADTs that are defined in this mini-textbook provide a starting point for a more in-depth discussion of various options. In lectures, assignments, and exams, variants on ADTs presented here may be considered.

In considering ADT operations, we consider the following rough categories:

Create A create operation may or may not include initialization.

Query Such operations do not modify the ADT, but instead return information such as whether or not the ADT is empty.

Search A search operation may determine whether or not a data item with a specified value is stored in the ADT, or may return the value of a data item with a particular specification, such as its position within the ADT.

Add Adding a value to an ADT may or may not require that the data item is not present, and may or may not specify the position in which it should be added.

Modify A data item may be modified based on its value and its position (absolute or relative).

Delete Deleting a value may or may not require that the data item is present, and may or may not specify the value or position (absolute or relative).

Table 6.1 outlines conventions that we will use in our ADT defintions.

Name	What it does	Options
CREATE*()*	produces a new empty ADT	may specify capacity and/or initialize
IS_EMPTY*(A)*	produces *True* if A is empty, else *False*	
IS_IN*(A, Data)*	produces *True* if data item *Data* is in A, else *False*	
LOOK_UP*(A, Info)*	produces a data item corresponding to *Info*	*Info* may include positional information or full or partial information on a value
ADD*(A, Data)*	adds *Data* to A, perhaps replacing another data item	may specify positional information
DELETE*(A, Data)*	deletes *Data* from A	may have the precondition that *Data* is in A or may make no change if *Data* is not present

Table 6.1: Conventions and options for ADT operations

Finally, we observe that certain operations will produce either a data item or, if it is not present, the value *False*. In order to prevent ambiguity, we assume throughout that *False* is not a data item. It would be easy to modify an implementation to allow *False* to be a data item by choosing a different way of indicating the absence of a data item.

6.2 Groups of ADTs

The course is organized so that ADTs are presented roughly in order of increasing complexity, based on the type of data items stored and the types of operations supported. To make the progression clear, ADTs have been placed into groups. These group names are not used in other sources, and are designed not as material to memorize but rather as a framework to help make the relationships among the ADTs more coherent.

One way in which groups are characterized is by how data items can be accessed. In the simplest ADTs, in group A, the only information that is used is the value of the data item itself. In the remaining groups, each data item can be viewed as having a **position**. Although many of the positions can be seen as describing a spatial relationship among data items, it is important to remember that there is no obligation that data items stored in adjacent positions in the ADT are also stored in adjacent positions in the data structure that implements the ADT. In order to emphasize this distinction, we will consider various implementations that violate our intuition.

The ADTs in group B allow the access of data items by position rather than by value. In some of the ADTs in the group, a position of a data item is determined by the operation that adds the data item; in others, the position of a data item may depend on operations on other

data items. In each ADT in the group, the items can be viewed as being arranged by a linear ordering in one dimension, with (again) the caveat that the actual implementation may store them in a different order entirely.

More complex relationships among data items are captured by the ADTs in group C, where the positions of data items can be related in non-linear ways. In trees, data items are stored as nodes that are connected by edges. Even more generality is possible in graphs, in which vertices and edges may both be data items and in which the constraints on connections are fewer than those for trees.

The ADTs in group D can be viewed as customizations of earlier ADTs in which data items are now (key, element) pairs, where the element in turn can be customized to store multiple values. The ADTs differ in options for the keys, where the constraints on the keys (e.g. to be distinct or to be orderable) give rise to ADT operations and implementations that leverage that information.

Finally, we use group E for other ADTs mentioned in the course, some only briefly.

6.3 Group A

In this group of ADTs, we first consider data items of any type. For each ADT in this group, no positional information is provided. In the ADT Multiset (Table 6.2), more than one copy of a value can be stored, whereas in the ADT Set (Table 6.3), each value can appear at most once. The pseudocode interfaces presented here are the final ones presented in class; intermediate interfaces may be discussed in the process of developing the ADTs.

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty multiset	
IS_IN*(M, Data)*	*M* is a multiset, *Data* is a data item	*True* if present, else *False*	
ADD*(M, Data)*	*M* is a multiset, *Data* is a data item		adds a copy of *Data* to *M*
DELETE*(M, Data)*	*M* is a multiset, *Data* is a data item in *M*		deletes one copy of *Data* from *M*

Table 6.2: Pseudocode interface for ADT Multiset

6.4 Group B

The ADTs in group B differ from those in group A due to the specification of positional information associated with each data item. For each of the ADTs in the group, the positions can be viewed as forming a one-dimensional, linear arrangement. Depending on the ADT, the position of a data item can depend on one or more operations, and access may be limited to data items in specific positions.

In particular, in the ADT Stack, additions and deletions occur at the top of the stack. We will use the conventions that appear in Table 6.4; slight variants appear in other sources, such

as the Necaise textbook using the term peek instead of TOP, and POP being defined such that either an error occurs if the stack is empty or nothing happens if the stack is empty. In contrast, in the ADT Queue (Table 6.5), addition occurs at the back of the queue whereas deletion occurs at the front of the queue.

In the ADT Indexed Sequence (Table 6.6), the position of a data item is associated with an index. Note that for this ADT, the positions of data items depend on the sequence of operations executed, not the values of the data items. In particular, data need not be orderable. There is at most one value per position. In the ADT Ranking (Table 6.7), each position is a rank, where at any time there is exactly one item of each rank from 0 up to the maximum rank. The ADT Grid (Table 6.8) associates two pieces of positional information with each data item.

For all the ADTs, it is crucial to remember that the positions not need to be related in any particular way to how data items are stored in an implementation.

6.5 Group C

In group C we consider more complex relationships among data items, structured as trees or graphs. In a tree, each position is a node; in the operations, each node is identified by its unique ID. We consider ADTs for the Binary Tree (Table 6.9), the Ordered Tree (Table 6.10), and the Unordered Tree (Table 6.11). The ADT Unordered Tree differs from the ADT Ordered Tree in several ways: ADD_LEAF does not require the specification of a sibling and CHILDREN returns nodes in a specific order.

In an undirected graph (Tables 6.12 and 6.13), both vertices and edges can be seen as positions (and in a directed graph, Tables 6.14 and 6.15, both vertices and arcs); here we opt to restrict our attention to simple graphs (graphs in which there can be at most one edge joining a pair of vertices) and hence identify each edge using the IDs of its endpoints. As is true for all of the ADTs discussed in the course, there are other, equally valid, choices that could have been made instead. We will assume that there is an ADT Pair that allows us extract the endpoints in each pair.

In simple situations, it suffices to consider just a single value for a node or vertex ADT. For our implementations, we may use an ID in place of an ADT, or we may assume that IDs of n nodes or vertices are the integers 0 through $n - 1$.

6.6 Group D

Group D differs from the earlier groups in that data items are now (key, element) pairs. Different ADTs in the group put different constraints on the keys. For the ADT Dictionary (Table 6.16), the data items are (key, element) pairs, where keys are distinct but not necessarily orderable, and elements are any data. The term dictionary is inspired by, but not identical to, a dictionary for a language, where keys are words and elements are etymology, pronunciation, and definitions. It is similar, but not identical, to the Python data type dictionary, as well as similar to the Necaise textbook's ADT Map (which is restricted to the case where keys are orderable).

In contrast, for the ADT Priority Queue (Table 6.17), the data items are (key, element) pairs, where keys are orderable but not necessarily distinct, and elements are any data. Instead of looking up items by their keys, the ADT supports access to an item with the minimum key value.

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty set	
IS_IN*(S, Data)*	*S* is a set, *Data* is a data item	*True* if present, else *False*	
ADD*(S, Data)*	*S* is a set, *Data* is a data item not in *S*		adds *Data* to *S*
DELETE*(S, Data)*	*S* is a set, *Data* is a data item in *S*		deletes *Data* from *S*

Table 6.3: Pseudocode interface for ADT Set

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty stack	
IS_EMPTY*(S)*	*S* is a stack	*True* if empty, else *False*	
TOP*(S)*	*S* is a stack that is not empty	data item that is on top	
PUSH*(S, Data)*	*S* is a stack *Data* is a data item		adds *Data* to top of *S*
POP*(S)*	*S* is a stack that is not empty	data item that was on top	deletes data item from top of *S*

Table 6.4: Pseudocode interface for ADT Stack

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty queue	
IS_EMPTY*(Q)*	*Q* is a queue	*True* if empty, else *False*	
FIRST*(Q)*	*Q* is a queue that is not empty	data item that is first	
ENQUEUE*(Q, Data)*	*Q* is a queue *Data* is a data item		adds *Data* to back of *Q*
DEQUEUE*(Q)*	*Q* is a queue that is not empty	data item that was first	deletes data item from front of *Q*

Table 6.5: Pseudocode interface for ADT Queue

Name	Preconditions	Produces	Mutates
CREATE*(Cap)*	*Cap* is a positive integer	a new empty indexed sequence of capacity *Cap*, with all entries initialized to empty	
CAP*(I)*	*I* is an indexed sequence	the capacity of *I*	
IS_EMPTY*(I)*	*I* is an indexed sequence	*True* if empty, else *False*	
LOOK_UP*(I, Index)*	*I* is an indexed sequence, *Index* is a nonnegative integer less than the capacity of *I*	data item at index *Index*, if any, else *False*	
ADD*(I, Index, Data)*	*I* is an indexed sequence, *Index* is a nonnegative integer less than the capacity of *I*, *Data* is a data item		stores data item *Data* at index *Index*, replacing any previous data item at index *Index*
DELETE*(I, Index)*	*I* is an indexed sequence, *Index* is a nonnegative integer less than the capacity of *I*	data item that was at index *Index*, if any, else *False*	deletes data item at index *Index*, if any

Table 6.6: Pseudocode interface for ADT Indexed Sequence

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty ranking	
IS_EMPTY*(R)*	*R* is a ranking	*True* if empty, else *False*	
MAX_RANKING*(R)*	*R* is a ranking	maximum rank used (-1 if empty)	
LOOK_UP*(R, Rank)*	*R* is a ranking, *Rank* is less than or equal to the maximum rank of *R*	data item that has rank *Rank*	
ADD*(R, Rank, Data)*	*R* is a ranking, *Data* is a data item, *Rank* is at most one more than the maximum rank of *R*		adds *Data* with rank *Rank* and other all items with ranks *Rank* or greater have ranks increased by one
DELETE*(R, Rank)*	*R* is a ranking, *Rank* is less than or equal to the maximum rank of *R*	data item that had rank *Rank*	deletes item with rank *Rank* and all items with ranks *Rank* $+1$ or greater have ranks decreased by one

Table 6.7: Pseudocode interface for ADT Ranking

Name	Preconditions	Produces	Mutates
CREATE*(Num_Rows, Num_Cols)*	*Num_Rows* and *Num_Cols* are positive integers	a new grid with *Num_Rows* rows and *Num_Cols* columns, with all entries initialized to empty	
ROWS*(G)*	*G* is an grid	number of rows	
COLUMNS*(G)*	*G* is an grid	number of columns	
IS_EMPTY*(G)*		*True* if empty, else *False*	
LOOK_UP*(G, Row, Col)*	*G* is an grid, *Row* and *Col* are nonnegative integers, $0 \leq Row < Num\_Rows$, $0 \leq Col < Num\_Cols$	data item in row *Row* and column *Col*, if any, else *False*	
ADD*(G, Row, Col, Data)*	*G* is an grid, *Row* and *Col* are nonnegative integers, $0 \leq Row < Num\_Rows$, $0 \leq Col < Num\_Cols$, *Data* is any data item		stores data item *Data* in row *Row* and column *Col*, replacing any previous data item in row *Row* and column *Col*
DELETE*(G, Row, Col)*	*G* is an grid, *Row* and *Col* are nonnegative integers, $0 \leq Row < Num\_Rows$, $0 \leq Col < Num\_Cols$	data item that was in row *Row* and column *Col*, if any, else *False*	deletes data item in row *Row* and column *Col*, if any

Table 6.8: Pseudocode interface for ADT Grid

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty binary tree	
IS_EMPTY*(B)*	B is a binary tree	*True* if empty, else *False*	
ROOT*(B)*	B is a binary tree that is not empty	root of B	
VALUE*(B, Node)*	B is a binary tree, *Node* is a node in B	value stored in *Node*	
PARENT*(B, Node)*	B is a binary tree, *Node* is a node in B	parent of *Node* if any, else *False*	
LEFT_CHILD*(B, Node)*	B is a binary tree, *Node* is a node in B	left child of *Node* if any, else *False*	
RIGHT_CHILD*(B, Node)*	B is a binary tree, *Node* is a node in B	right child of *Node* if any, else *False*	
SET_VALUE*(B, Node, Data)*	B is a binary tree, *Node* is a node in B, *Data* is a data item		sets value stored in *Node* to *Data*
ADD_LEAF*(B, Par, Side, Data)*	B is a binary tree, *Par* and *Side* are both empty or *Par* is a node in B and *Side* is either *Left* or *Right*, *Data* is a data item	new node	creates a new node storing *Data* to replace/add the root if *Par* is empty and replace/add the *Side* subtree of *Par* otherwise
DELETE_LEAF*(B, Node)*	B is a binary tree, *Node* is a leaf in B		deletes *Node*

Table 6.9: Pseudocode interface for ADT Binary Tree

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty ordered tree	
IS_EMPTY*(O)*	*O* is an ordered tree	*True* if empty, else *False*	
ROOT*(O)*	*O* is an ordered tree that is not empty	root of *O*	
VALUE*(O, Node)*	*O* is an ordered tree, *Node* is a node in *O*	value stored in *Node*	
PARENT*(O, Node)*	*O* is an ordered tree, *Node* is a node in *O*	parent of *Node* if any, else *False*	
CHILDREN*(O, Node)*	*O* is an ordered tree, *Node* is a node in *O*	all children of *Node* (Group B ADT)	
ONE_CHILD *(O, Node, Index)*	*O* is an ordered tree, *Node* is a node in *O*, *Index* is a nonnegative integer at most one less than the number of children of *Node*	child *Index* of *Node*	
SET_VALUE *(O, Node, Data)*	*O* is an ordered tree, *Node* is a node in *O*, *Data* is a data item		sets value stored in *Node* to *Data*
ADD_LEAF *(O, Par, Sib, Data)*	*O* is an ordered tree, *Par* is a node in *O* or empty, *Sib* is a child of node *Par* or empty, *Data* is a data item	new node	creates a new node storing *Data* to replace/add the root if *Par* is empty, become the first child of *Par* if *Sib* is empty, and otherwise become the next sibling of *Sib*
DELETE_LEAF *(O, Node)*	*O* is an ordered tree, *Node* is a leaf in *O*		deletes *Node*

Table 6.10: Pseudocode interface for ADT Ordered Tree

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty unordered tree	
IS_EMPTY*(U)*	*U* is an unordered tree	*True* if empty, else *False*	
ROOT*(U)*	*U* is an unordered tree that is not empty	root of *U*	
VALUE*(U, Node)*	*U* is an unordered tree, *Node* is a node in *U*	value stored in *Node*	
PARENT*(U, Node)*	*U* is an unordered tree, *Node* is a node in *U*	parent of *Node* if any, else *False*	
CHILDREN*(U, Node)*	*U* is an unordered tree, *Node* is a node in *U*	all children of *Node* (Group B ADT)	
SET_VALUE *(U, Node, Data)*	*U* is an unordered tree, *Node* is a node in *U*, *Data* is a data item		sets value stored in *Node* to *Data*
ADD_LEAF *(U, Par, Data)*	*U* is an unordered tree, *Par* is a node in *U* or empty, *Data* is a data item	new node	creates a new node storing *Data* to replace/add the root if *Par* is empty, and otherwise become a child of *Par*
DELETE_LEAF *(U, Node)*	*U* is an unordered tree, *Node* is a leaf in *U*		deletes *Node*

Table 6.11: Pseudocode interface for ADT Unordered Tree

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty undirected graph	
VERTICES*(G)*	G is an undirected graph	IDs of vertices (Group B ADT)	
EDGES*(G)*	G is an undirected graph	unordered pairs of IDs for endpoints of edges (Group B ADT)	
VERTEX_VALUE*(G, One)*	G is an undirected graph, *One* is the ID of a vertex in G	value of vertex with ID *One*	
EDGE_VALUE*(G, One, Two)*	G is an undirected graph, *One* and *Two* are IDs of vertices in G that are endpoints of an edge	value of edge with endpoints with IDs *One* and *Two*	
NEIGHBOURS*(G, One)*	G is an undirected graph, *One* is the ID of a vertex in G	IDs of all neighbours of vertex with ID *One* (Group B ADT)	
ARE_ADJACENT *(G, One, Two)*	G is an undirected graph, *One* and *Two* are IDs of vertices in G	*True* if there is an edge with endpoints vertices with IDs *One* and *Two*, else *False*	

Table 6.12: Pseudocode interface for ADT Undirected Graph, part 1

Name	Preconditions	Produces	Mutates
SET_VERTEX_VALUE *(G, One, Data)*	G is an undirected graph, *One* is the ID of a vertex in G, *Data* is a data item		sets value of vertex with ID *One* to *Data*
SET_EDGE_VALUE *(G, One, Two, Data)*	G is an undirected graph, *One* and *Two* are IDs of vertices in G that are endpoints of an edge, *Data* is a data item		sets value of edge with endpoints with IDs *One* and *Two* to *Data*
ADD_VERTEX *(G, One)*	G is an undirected graph without a vertex with ID *One*		adds a new vertex with ID *One*
ADD_EDGE *(G, One, Two)*	G is an undirected graph, *One* and *Two* are IDs of vertices in G that are not endpoints of an edge		adds a new edge with endpoints with IDs *One* and *Two*
DELETE_VERTEX *(G, One)*	G is an undirected graph, *One* is the ID of a vertex in G		deletes vertex with ID *One* and all edges incident on *One*
DELETE_EDGE *(G, One, Two)*	G is an undirected graph, *One* and *Two* are IDs of vertices in G that are endpoints of an edge		deletes edge with endpoints vertices with IDs *One* and *Two*

Table 6.13: Pseudocode interface for ADT Undirected Graph, part 2

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty directed graph	
VERTICES*(G)*	G is a directed graph	IDs of vertices (Group B ADT)	
ARCS*(G)*	G is a directed graph	ordered pairs of IDs for tails and heads of arcs (Group B ADT)	
VERTEX_VALUE*(G, One)*	G is a directed graph, *One* is the ID of a vertex in G	value of vertex with ID *One*	
ARC_VALUE*(G, One, Two)*	G is a directed graph, *One* and *Two* are IDs of vertices of G that are tail and head of an arc	value of arc with tail *One* and head *Two*	
IN_NEIGHBOURS*(G, One)*	G is a directed graph, *One* is the ID of a vertex in G	IDs of all in-neighbours of *One* (Group B ADT)	
OUT_NEIGHBOURS*(G, One)*	G is a directed graph, *One* is the ID of a vertex in G	IDs of all out-neighbours of *One* (Group B ADT)	
IS_ARC*(G, One, Two)*	G is a directed graph, *One* and *Two* are IDs of vertices in G	*True* if there is an arc from tail *One* to head *Two*, else *False*	

Table 6.14: Pseudocode interface for ADT Directed Graph, part 1

Name	Preconditions	Produces	Mutates
SET_VERTEX_VALUE *(G, One, Data)*	*G* is a directed graph, *One* is the ID of a a vertex in *G*, *Data* is a data item		sets value of vertex with ID *One* to *Data*
SET_ARC_VALUE *(G, One, Two, Data)*	*G* is a directed graph, *One* and *Two* are IDs of vertices in *G* that are tail and head of an arc, *Data* is a data item		sets value of arc with tail *One* and head *Two* to *Data*
ADD_VERTEX*(G, One)*	*G* is a directed graph without a vertex with ID *One*		adds a new vertex with ID *One*
ADD_ARC*(G, One, Two)*	*G* is a directed graph, *One* and *Two* are IDs of vertices in *G* with no arc with tail *One* and head *Two*		adds a new arc from head *One* to tail *Two*
DELETE_VERTEX*(G, One)*	*G* is a directed graph, *One* is the ID of a vertex in *G*		deletes vertex with ID *One* and all arcs with *One* as tail or head
DELETE_ARC*(G, One, Two)*	*G* is a directed graph, *One* and *Two* are IDs of vertices in *G* that are tail and head of an arc		deletes arc with tail *One* and head *Two*

Table 6.15: Pseudocode interface for ADT Directed Graph, part 2

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty dictionary	
IS_EMPTY*(D)*	*D* is a dictionary	*True* if empty, else *False*	
LOOK_UP*(D, Key)*	*D* is a dictionary *Key* is a key	element stored with *Key* if any, else *False*	
ADD*(D, Key, Element)*	*D* is a dictionary, *Key* is a key, *Element* is an element		stores pair *(Key, Element)*, replacing pair *(Key, Other)* if any exists
DELETE*(D, Key)*	*D* is a dictionary, *Key* is a key of a pair in *D*		deletes pair *(Key, Element)*

Table 6.16: Pseudocode interface for ADT Dictionary

6.7 Group E

This section contains ADTs discussed briefly at the end of the course, namely ADT Disjoint Set (Table 6.18) and ADT Range (Table 6.19).

6.8 Code interfaces for ADTs

Code interfaces can be found on the course website for selected ADTs used in lectures or assignments. Examples given may be variants from those presented in lecture.

In the translation of ADT operations from the pseudocode interface to the code interface, the following ideas will be used:

- When an ADT is implemented as a object with the same name as the ADT, the pseudocode operation *Create* will be implemented using the method `__init__`. Accordingly, to use the method for the *Create* operation, use the name of the object, such as `Multiset()`.
- The operation Is_In will be implemented as `in`. For example, for the function application Is_In*(L, Item)*, we will use `item in L`.
- For most other operations, the name will appear in lower case using dot notation, with names of variables being put in lower case as well. For example, the pseudocode operation Add*(S, Data)* will be implemented as the method `add` and applies as `s.add(data)`.

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty priority queue	
IS_EMPTY*(P)*	*P* is a priority queue	*True* if empty, else *False*	
LOOK_UP_MIN*(P)*	*P* is a priority queue that is not empty	pair such that key has minimum value	
ADD*(P, Key, Element)*	*P* is a priority queue, *Key* is a key, *Element* is an element		adds pair *(Key, Element)*
DELETE_MIN*(P)*	*P* is a priority queue that is not empty	pair such that key has minimum value	deletes pair

Table 6.17: Pseudocode interface for ADT Priority Queue

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty set of disjoint sets	
FIND_REP*(D, Data)*	*D* is a disjoint set, *Data* is a data item in *D*	data item that serves as representative of set containing *Data*	
ADD_SET*(D, Data)*	*D* is a disjoint set, *Data* is a data item not in *D*		creates a new set containing only *Data*
UNION*(D, One, Two)*	*D* is a disjoint set, *One* and *Two* are data items in *D*		combines the sets containing *One* and *Two*

Table 6.18: Pseudocode interface for ADT Disjoint Set

Name	Preconditions	Produces	Mutates
CREATE*()*		a new empty range	
RANGE*(R, Low, High)*	*R* is a range, *Low* and *High* are values	all data items in *R* with values in the range from *Low* to *High*	
ADD*(R, Data)*	*R* is a range, *Data* is a data item not in *R*		adds *Data* to *R*
DELETE*(R, Data)*	*R* is a range, *Data* is a data item in *R*		deletes *Data* from *R*

Table 6.19: Pseudocode interface for ADT Range

Chapter 7

Data types and structures

7.1 Introducing data structures

The data structures considered in the course range from multipurpose data structures that can be used for various ADTs (e.g. the array) to more specialized data structures that are optimized for a particular ADT (e.g. the adjacency matrix). For a single data structure, there can be different algorithms used to implement the same operation. For example, linear probing and double hashing can use the same data structure. The design of a data structure can be optimized for special situations, where there is additional information known about operations to be executed or the type of data to be stored, as discussed in Section 7.2.

In each case we will specify whether the data structure makes use of contiguous memory, linked memory, or both. Data structures are sometimes used in combination; in addition, at times we make use of auxilliary variables storing such information as indices and pointers. Most of the data structures we consider will be **internal**, meaning that we assume that all of the information is stored in main memory. We will also briefly discuss implications of using a memory model in which data is stored in external memory, resulting in **external** data structures. Further discussion of memory models can be found in Chapter 3, and further discussion of memory in Appendix C.

We introduce basic data structures in Section 7.3, extending the ideas to variants in Section 7.4. Data structures for specific ADTs follow in Sections 7.5 (trees), 7.6 (graphs), 7.7 (dictionaries), and 7.8 (priority queues). The chapter ends with a brief discussion of other data structures considered towards the end of the course (Sections 7.9 and 7.10) as well as new criteria for data structures and a new way of analyzing operations (Section 7.11).

7.2 Types of data

Although in many cases we will be able to store any type of data in our data structures, at times we will take advantage of special properties of the data. When we are storing **general data**, the only operation that we can execute on data items is comparison for equality. If instead we have **orderable data**, we can compare data items either for equality or for ordering, with operations such as $<$, $>$, $\leq$, and $\geq$. The most general type of data we consider is **digital data**, which supports additional types of operations as computations, such as decomposing a data item into

smaller values (such as a string into characters or a number into digits), or applying arithmetic operations to a number. As discussed in Section 7.3, we can also handle compound data, where we may focus on individual fields and their properties.

When the data that is being stored does not change, the ADT can be restricted to remove the addition and deletion operations; alternatively, the original ADT is retained with the understanding that the data is unchanging, or **static**. Here we will make a distinction between **static data** (no additions, deletions, or modifications occur) and a **static data structure** (no rearrangements of the data items are made). Be careful when encountering the term in other sources, as the term **static data structure** is used for both definitions, as well as to describe a data structure of fixed size but potentially changing data.

7.3 Basic data structures

Here we introduce two basic data structures, one using contiguous memory and the other using linked memory. An **array** (not equivalent to the Python array) is a data structure used to store a sequence of data items or **elements** in contiguous memory (Figure 7.1). If an array has enough space for k elements, each of which is allocated b bits, then the total number of bits in the chunk allocated to the array is kb bits. Each subchunk can be viewed as a **slot** in the array which either stores a data item or is empty. The **size** of an array is the number of slots it contains. Each slot is assigned a non-negative integer **index**, starting at 0, that indicates its position in the array. In the example of k slots of b bits each, the address of the slot with index i can be found by adding bi to the address of the first bit in the array. The ability to read or modify any of the slots in time that is independent of the size of the array is called **random access**.

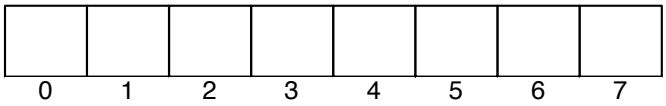

Figure 7.1: An array data structure

One special type of array is a **bit vector**, in which each slot consists of a single bit. A typical use of a bit vector would be to mark each integer i from 0 to the size of the array minus 1 as being either present (setting the bit at index i to 1) or absent (setting the bit at index i to 0).

A sequence of data items can be stored in linked memory using a **linked list** (Figure 7.2). In a linked list, there is a chunk of memory, or **node**, allocated to each element of the list as well as a pointer to the node corresponding to the first data item in the sequence. Each node is big enough to store not only a data item but also a pointer to the chunk containing the next data item in the sequence. For clarity, at times we will use the term **linked node** to distinguish the term from the term node used in the description of a tree (Section 5.1).

Natural extensions of contiguous and linked implementations allow enough space for a compound data item (defined in Section 7.2) in each array slot or linked node. Access to individual

fields can be achieved by viewing each data item as an ADT (including operations for modification and access to individual values) or by viewing each array slot or linked node as containing specified space for each field.

7.4 Variants on arrays and data structures

The data structures in this section include modifications that allow more flexible navigation in or modification to a sequence of data items. In the circular array and circular linked list, the data items can be viewed as connected in a circular fashion. In the doubly-linked list, the idea of pointers forwards and backwards in the sequence is introduced. Finally, the doubly-linked circular list makes use of both circular connections and pointers forwards and backwards in each linked node.

In a **circular array** (Figure 7.3), it is possible to store a sequence of data items starting at any slot; such a structure works well for additions and removals at the beginning and end of the sequence. A circular array consists of an array as well as two variables *Head* and *Tail*, where *Head* stores the index of the slot containing the first data item in the sequence and *Tail* stores the index of the slot containing the last data item in the sequence. When *Head* is smaller than *Tail*, the remaining data items are placed in the slots between *Head* and *Tail*. If instead, as in Figure 7.3, *Tail* is smaller than *Head*, then the sequence of items can be found in the slots from *Head* to the last slot in the array, followed by the slots from the first slot in the array through *Tail*. In the example in the figure, the data structure stores the sequence a, b, c, d, e, f.

In a **circular linked list**, the idea of joining the last data item to the first item is used to make it easy to find both the first and the last items. As shown in Figure 7.4, there is a pointer to the tail of the list (from which the head of the list can easily be found).

The **doubly-linked list** (Figure 7.5) and **doubly-linked circular list** (Figure 7.6) are both linked data structures that make use of nodes containing not only a data item and a pointer to the next node in the data structure, but also a pointer to the previous node in the data structure. Either can have a *Head* pointer, a *Tail* pointer, or both, though since the head and tail can be easily found from each other in the doubly-linked circular list, the advantage of including both pointers is small.

7.5 Data structures for trees

In the course we will consider data structures for ordered trees (including binary trees) as well as unordered trees; we will also consider modifications and enhancements of these ADTs to support the use of various types of trees as implementations of other ADTs, most notably the ADT Dictionary.

In linked implementations of trees, each node in the tree is implemented as a node in a linked structure. (To prevent confusion, in the remainder of this section the term "tree node" will be used for a node in the tree and the term "linked node" will be used for a node in a data structure.) Depending on the application, a linked node may contain various types of data (such as a value, colour, and/or weight) and various links (such as links to the parent and children). In such implementations, the search for a tree node consists of starting at the

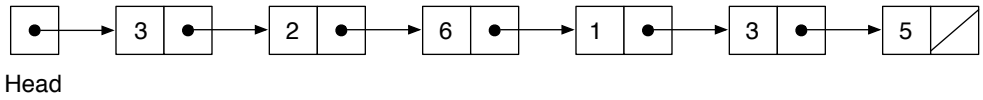

Figure 7.2: A linked list data structure

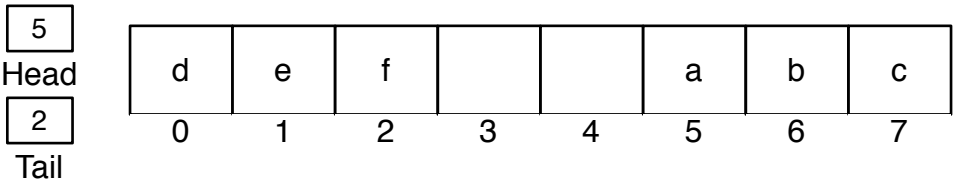

Figure 7.3: A circular array data structure

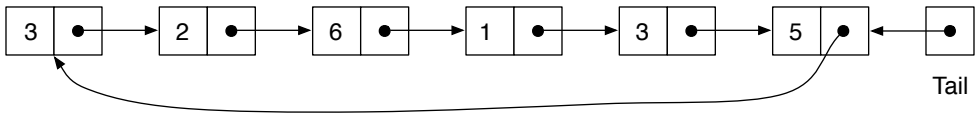

Figure 7.4: A circular linked list

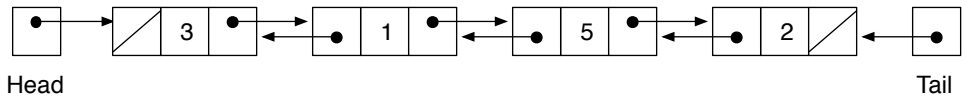

Figure 7.5: A doubly-linked list data structure

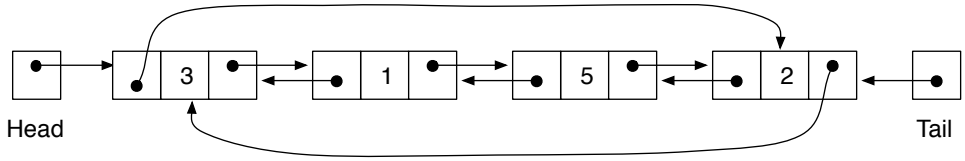

Figure 7.6: A doubly-linked circular list data structure

linked node corresponding to the root and traversing links until the target tree node is found. Modifying a tree, such as by adding or deleting a tree node, can be achieved by changing various pointers. In more sophisticated structures, additional operations may be supported, entailing the modification of various parts of a tree. One such example is a *threaded tree*, in which there is a link from the linked node for a tree node u to the linked node for u's successor in a traversal order, such as an inorder traversal for a binary tree. (Information on tree traversals can be found in Section 5.2.)

If a tree node can have any number of children, using a linked implementation provides a challenge, since storing one pointer for each child may result in a linked node of size linear in the number of tree nodes. In an implementation studied in class, we handle this situation by using a linked structure in which again each linked node represents a tree node *Node*, but in this case contains pointers to the linked nodes representing the following tree nodes:

- the parent of *Node*, if *Node* is the first child of its parent, or otherwise the previous sibling of *Node*;
- the first child of *Node*, if any; and
- the next sibling of *Node*, if any.

For an unordered tree, the order of siblings is arbitrary.

In contiguous implementations of trees, the biggest challenge is to ensure that it is possible to navigate in the tree by figuring the relationships among tree nodes stored at different slots in an array. When there are few constraints on how a tree might be structured, any implementation that correlates a fixed slot with a particular position in the tree is likely to waste a lot of space, as slots need to be allocated for nonexistent tree nodes. An example given in class makes use of a level-order traversal of a tree.

We observe that the contiguous implementation is a viable option when the tree is either a perfect or a complete binary tree (defined in Section 5.1), as in a heap (discussed in Section 7.8).

7.6 Data structures for graphs

Due to the amount of information that can be stored in a graph, we present simplified versions of each data structure and then mention how they can be modified to accommodate other features. We may initially restrict vertices to integers in the range from 0 to the number of vertices minus one, and we may initially restrict edges to be pairs of such integers. In modifications, we make changes for directed graphs, weighted graphs, and a richer set of information for vertices and edges, potentially viewing each vertex and each edge as its own ADT.

7.6.1 Edge list

In its simplest form, an **edge list** is a data structure that stores all the edges in the graph in no particular order. For the purposes of this course, we assume that the edge list is implemented as a linked list, where each linked node contains the endpoints of the edge being stored. For a directed graph, we could differentiate between the head and the tail of an arc, and for a

weighted graph, we could store edge weights (or other information about edges) along with the endpoints. Here vertices need not be limited to the integers in the range from 0 to the number of vertices minus one.

More challenging is the storing of information about vertices, which may be required if there is a vertex without any neighbours (and hence not represented by any edge). We may choose to add one or more auxiliary data structures to store various types of data, either as individual structures for each type of information, or as a single structure containing all information.

7.6.2 Adjacency matrix

In an **adjacency matrix**, there is a slot for each ordered pair of vertices, where a 1 can be used to designate the existence of an edge (or arc) and a 0 otherwise. Typically a slot is accessed using two integer indices, corresponding to the two endpoints of the edge or arc.

For a directed graph, the order of the vertices in the pair corresponds to which vertex is the head and which the tail of the arc. Additional information about edges, such as weights, can be stored instead of 1's and 0's. For additional information about vertices, additional data structures can be used.

7.6.3 Adjacency list

An **adjacency list** is typically implemented as an array of pointers to linked lists, where there is one slot in the array (and hence one linked list) for each vertex. The linked list associated with a particular vertex corresponds to the neighbours of that vertex or, alternatively, to the edges incident on that vertex.

For a directed graph, each slot in the array can contain two pointers, one to a linked list of in-neighbours and the other to a linked list of out-neighbours. Additional information about vertices can also be stored in the slots of the array, and additional information about edges can be stored in linked nodes in the linked lists. Alternatively, array entries or linked nodes can provide pointers to vertex or edge ADTs, respectively.

7.7 Data structures for dictionaries

For economy of space, in the data structures described below, many descriptions are presented in terms of the position of a key, without mention of the element. You can assume that each data structure stores (key, element) pairs, using one of the techniques for compound data discussed in the last paragraph of Section 7.3.

7.7.1 Binary search trees

Many data structures are built on the concept of a **binary search tree**, which is a binary tree that satisfies the **binary search tree property**: for every node *Node* in the tree, the key *Key* stored in *Node* is greater than the values of keys stored in the left subtree of *Node* and less than the values of keys stored in the right subtree of *Node*. At times a more general definition is given, for situations in which keys are not distinct; in that case, we can replace "greater

than" by "greater than or equal to" and "less than" by "less than or equal to" in the property above. As a consequence of the binary search tree property, one can use an inorder traversal (Section 5.2) to extract the values stored in the nodes in nondecreasing order.

The binary search tree property plays a crucial role in the search operation, which relies on the fact that comparing a sought value to the value in a node identifies whether the value can be found in the node itself (if it is equal), in the left subtree (if it is smaller), or in the right subtree (if it is larger). In this sense, a binary search tree mimics the behaviour of binary search (Section 8.2.1), where we can view the current node being examined as playing the role of the middle element and the left and right subtrees as the resulting smaller sequences of values to be searched. Eventually, in the case of successful search, the sought value is found. An unsuccessful search will terminate by the discovery of a node that either contains a key larger than the sought key but no left child or a key smaller than the sought key but no right child.

As both ADD and DELETE rely on a search (for a place to add a node or a node to delete, respectively), we can describe implementations of the operations in terms of the above algorithm. The ADD operation can be viewed as the search for a location to add the new data item followed by a modification to the binary search tree. More specifically, the operation starts with an unsuccessful search for the value to add. The node at which the search terminates can be made into the parent of a new node containing the added value (a left child if the sought value is smaller than the key and a right child if the sought value is larger than the key). Consequently, the node containing a new value is a leaf.

For the operation DELETE, once the node containing the value has been found, the goal is to remove the node in such a way that what remains is still a binary search tree. In the easiest case, if the data item is in a leaf, the leaf can easily be removed. If instead the data item is not in a leaf, but instead is in a node *Node* with a single child *Child*, then *Child* can be made into a child of the parent of *Node* (the left child, if *Node* is the left child of its parent, and the right child, if *Node* is the right child of its parent). Because all the values in the subtree rooted at *Child* are smaller (respectively, larger) than the value in the parent of *Node* when *Node* is a left child (respectively, right child) of its parent, the resulting tree is still a binary tree and still satisfies the binary search tree property.

For the most complicated scenario, we consider the deletion of a node *Node* that has two children. In this case, we determine the node *Succ* that stores the inorder successor of *Node*. Because an inorder traversal generates the values in the tree in order, we know that there cannot exist a value that is greater than the value stored in *Node* and also smaller than the value stored in *Succ*. We also know that since the value stored in *Succ* is greater than the value stored in *Node*, *Succ* must be a node in the right subtree of *Node*. Putting these two facts together leads to the conclusion that *Succ* cannot have a left child, as otherwise we would obtain a contradiction (the left child contains a value greater than the value stored in *Node*, as it is in the right subtree of *Node*, and smaller than the value stored in *Succ*, as it is in the left subtree of *Succ*).

We now observe that we can swap the values in *Node* and *Succ* and then delete *Succ*. Because *Succ* does not have a left child, it can be deleted using one of the two easier cases described above (where the node being deleted either is a leaf or has only a single child).

To determine the worst-case running times of the operations, we observe that the cost of searching depends on the number of nodes traversed, which in the worst case is one greater than the height of the tree. For the operation ADD, the only additional cost is the constant-time

cost of modification of the tree to insert the new value. For the operation DELETE, there may be both a constant-time cost for modification as well as the cost of finding an inorder successor, which is in $O(h)$ for h the height of the tree. In the following sections, we consider variants on binary search trees that ensure bounds on the height, and hence on the cost of searching.

7.7.2 AVL trees

Because the cost of searching for a node is proportional to its depth in the tree, various data structures have been designed to limit the overall height of the tree (and hence the depth of each node). We define a node in a binary tree to be **balanced** if the heights of its left and right subtrees differ by at most 1 and **imbalanced** otherwise; for the purposes of calculations, an absent subtree has height -1. A binary tree satisfies the **height balance property** if every node in the tree is balanced. An **AVL tree** is a binary search tree that satisfies the height balance property.

As a consequence of the property, any AVL tree with n nodes has height in $\Theta(\log n)$; the details of the proof are outside the scope of the course. In order to maintain the height balance property, at times the addition or deletion of a node will require a restructuring of the tree. We need to ensure that with each restructuring, the tree that is formed is an AVL tree; we need to check that it is a binary search tree and that it satisfies the height balance property. We implement an AVL as a binary search tree in which each node stores its height. We observe that by doing so, we are now further required to ensure that each restructuring results in correct heights to be stored in nodes.

The operation used to rebalance a tree is called a **rotation** at a node, where the rotation at a node entails replacing the subtree rooted at the node by another subtree that contains the same values but has a different root. Each rotation involves the changing of a constant number of pointers in a linked implementation of a binary tree. After an addition or a deletion (as in a binary search tree), we define the **pivot node** to be the imbalanced node of greatest depth in the resulting tree; the pivot node will be found on the path from the position of the added or deleted node to the root. Determining the pivot node can be achieved by tracing the path from the position of the added or deleted node to the root, updating heights and checking the balance of each node as it is processed. How the tree is rotated depends on the relative heights of the subtrees rooted at the children and grandchildren of the pivot node.

Depending on whether the operation was an addition or a deletion, a rotation may or may not result in all the nodes in the tree becoming balanced. In particular, we need to examine not only the balance of all the nodes in the new subtree, but also how the height of the subtree before the operation compares with the height of the subtree after the operation. In our arguments, we observe that we need only consider the balance of nodes on the path from the pivot node to the root, as for any other node, there are no changes in balance.

We first consider all the possible cases that might result in a node u becoming a pivot node. In the explanation, we use the following notation for any node v and subtree T: $L(v)$ is the left child of v in the tree before rotation; $R(v)$ is the right child of v in the tree before rotation, T_v is the subtree rooted at node v before rotation; and height(T) is the height of tree T. By the definition of a pivot node, we know that $L(u)$ and $R(u)$ are both balanced, but that u is imbalanced. Moreover, we know that u was balanced before the addition or deletion; since the

addition or deletion of a single node can only change the height of a subtree by 1, the difference in heights between the subtrees of u (that is, $T_{L(u)}$ and $T_{R(u)}$) must be 2.

As a consequence, one of the following cases must apply:

1. $\text{height}(T_{L(u)}) = \text{height}(T_{R(u)}) + 2$, and $\text{height}(T_{L(L(u))}) > \text{height}(T_{R(L(u))})$.
2. $\text{height}(T_{L(u)}) = \text{height}(T_{R(u)}) + 2$, and $\text{height}(T_{L(L(u))}) < \text{height}(T_{R(L(u))})$.
3. $\text{height}(T_{L(u)}) = \text{height}(T_{R(u)}) + 2$, and $\text{height}(T_{L(L(u))}) = \text{height}(T_{R(L(u))})$.
4. $\text{height}(T_{R(u)}) = \text{height}(T_{L(u)}) + 2$, and $\text{height}(T_{R(R(u))}) > \text{height}(T_{L(R(u))})$.
5. $\text{height}(T_{R(u)}) = \text{height}(T_{L(u)}) + 2$, and $\text{height}(T_{R(R(u))}) < \text{height}(T_{L(R(u))})$.
6. $\text{height}(T_{R(u)}) = \text{height}(T_{L(u)}) + 2$, and $\text{height}(T_{R(R(u))}) = \text{height}(T_{L(R(u))})$.

As we will discuss further as we consider each case, not all of the cases can occur for both additions and deletions.

Case 1

For convenience, we use h to denote $\text{height}(T_{R(u)})$, and hence $\text{height}(T_{L(u)}) = h + 2$. We now consider the subtrees of $L(u)$, and since the height of a tree is one greater than the maximum height of its subtrees, $\text{height}(T_{L(L(u))}) = h+1$. Finally, since $L(u)$ is balanced, $\text{height}(T_{R(L(u))}) = h$.

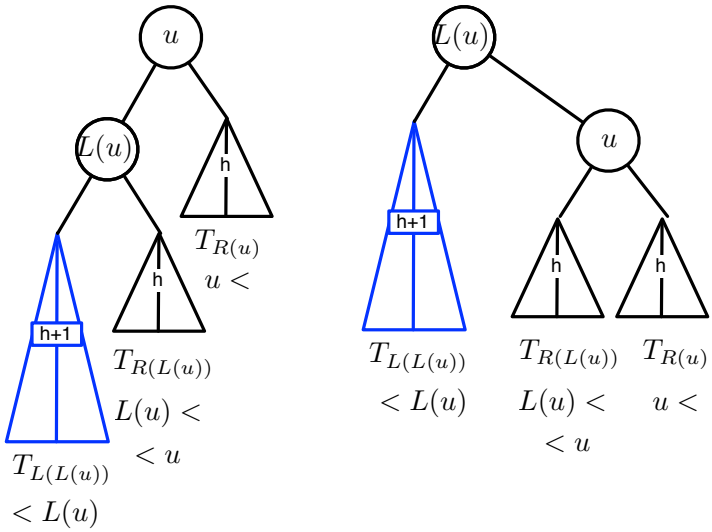

Figure 7.7: AVL case 1

We now rearrange the subtree rooted at u by making $L(u)$ the root of the tree, u the right child of $L(u)$, and $R(L(u))$ the left child of u, while retaining $R(u)$ as the right child of u and $L(L(u))$ as the left child of $L(u)$ (Figure 7.7). We make no changes to any of $T_{L(L(u))}$, $T_{R(L(u))}$, or $T_{R(u)}$, so they still contain balanced nodes and retain their original heights.

We now need to show that the resulting tree is a binary search tree and that all the nodes are balanced; to do so, we only need to consider the nodes u and $L(u)$, as no other nodes have had changes to the contents of their subtrees. Due to their positions in the original tree, we know that for each value x in $T_{L(L(u))}$, x is less than the value in $L(u)$. Similarly, we know that for each value x in $T_{R(L(u))}$, the value of x is between the values in $L(u)$ and u, and for each value x in $T_{R(u)}$, the value of x is greater than the value in u. Clearly, the binary search tree property holds at both u and $L(u)$.

To ensure that the all nodes are now balanced, we observe that u now has two subtrees of height h (and hence is balanced) and as a consequence $L(u)$ now has two subtrees of height $h+1$ (and hence is also balanced). Finally, we consider the parent of the new subtree. If the modification was an addition, then the height of subtree before addition was $h+2$ (as both subtrees of $L(u)$ would have had height h before the addition) and the height of the subtree after rotation would have been restored to $h+2$. If instead the modification was a deletion, then it would have resulted from the deletion of a node in $T_{R(u)}$ reducing its height from $h+1$ to h, and hence the height of the subtree before deletion was $h+3$ and the height after rotation is $h+2$.

Case 2

As in case 1, for convenience, we use h to denote height$(T_{R(u)})$, and hence height$(T_{L(u)}) = h+2$. In this case, $T_{R(L(u))}$ is the higher of the two subtrees of $L(u)$, and hence height$(T_{R(L(u))}) = h+1$. Using the same reasoning, the higher (if any) of the two subtrees of $R(L(u))$ will have height h, with the other subtree having height either h or $h-1$ (because $R(L(u))$ is balanced). Finally, since $L(u)$ is balanced and height$(T_{L(L(u))}) <$ height$(T_{R(L(u))}) = h+1$; we can conclude that height$(T_{L(L(u))}) = h$.

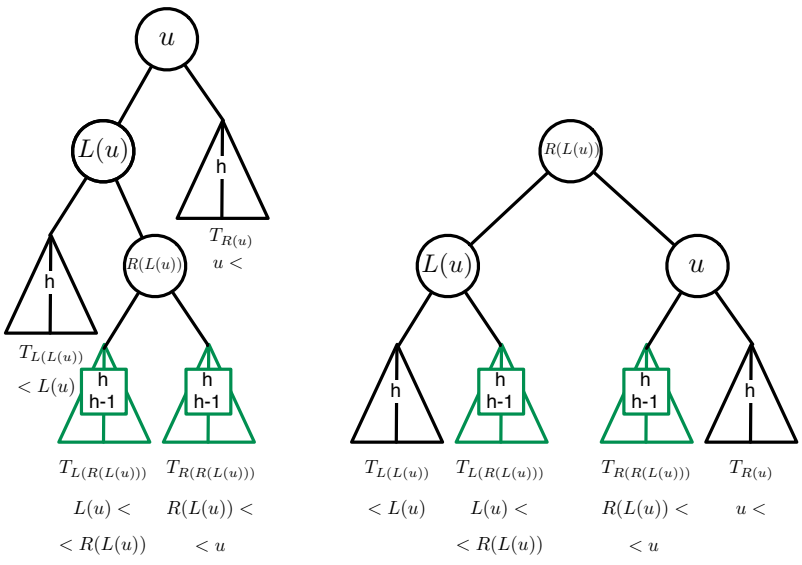

Figure 7.8: AVL case 2

In this case, we make the following rearrangements: $R(L(u))$ is made into the root of the

tree with left child $L(u)$ and right child u; the subtrees of $L(u)$ are $T_{L(L(u))}$ and $T_{L(R(L(u)))}$; and the subtrees of u are $T_{R(R(L(u)))}$ and $T_{R(u)}$, as seen in Figure 7.8.

Using reasoning similar to that in Case 1, we can show that the resulting tree is a binary search tree and that all the nodes are balanced, this time by focusing on u, $L(u)$, and $R(L(u))$. The ranges of values in each subtree are illustrated in Figure 7.8, and can be shown to satisfy the binary search tree property. We can show that all nodes in the new subtree are balanced due to each of $L(u)$ and u having one subtree of height h and one of height either $h-1$ or h.

Finally, we consider the balance of the parent of the new subtree. Because u was balanced before the modification, if the modification was an addition, the height of the subtree before addition was $h+2$. It is not hard to see that the height of the subtree after rotation is also $h+2$. If instead the modification was a deletion, since deletion cannot increase height, the subtree would have had height $h+3$ before modification and height $h+2$ after rotation.

Case 3

In this case, we again use h to denote height$(T_{R(u)})$, and hence height$(T_{L(u)}) = h+2$. Since the heights of the subtrees of $L(u)$ are equal, height$(T_{L(L(u))}) =$ height$(T_{R(L(u))}) = h+1$. We observe that this case cannot occur due to a single addition, since u would be imbalanced even if only one of the two subtrees of $L(u)$ had height $h+1$. Instead, there must have been a deletion from $T_{R(u)}$, reducing its height from $h+1$ to h.

For this case, the rotation and reasoning about the binary search tree property are the same as for Case 1. The only difference is in the resulting height of the tree, and hence in the balance of the parent of the subtree. In this case, the height of the subtree before deletion and the height of the subtree after rotation are both $h+3$.

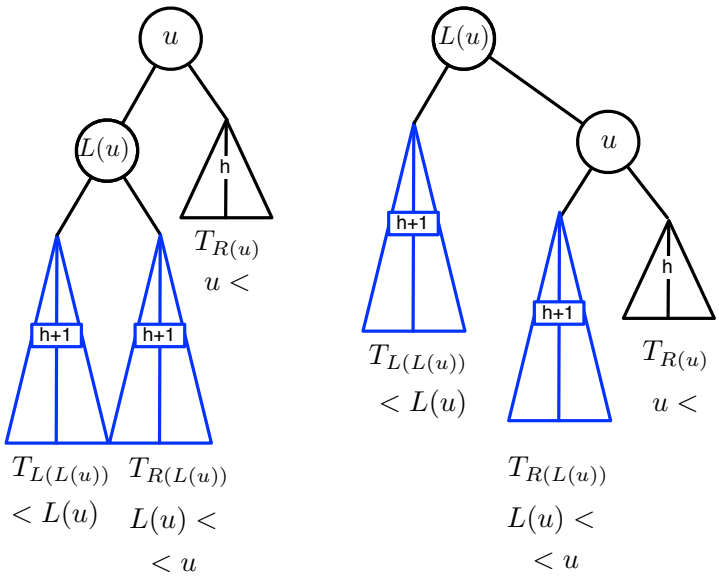

Figure 7.9: AVL case 3

Cases 4 through 6

Case 4 is symmetric to Case 1, as illustrated in Figure 7.10 below; the same arguments apply by replacing each L by R and each R by L. In the same way, Case 5, as illustrated in Figure 7.11, is symmetric to Case 2, and Case 6, as illustrated in Figure 7.12, is symmetric to Case 3.

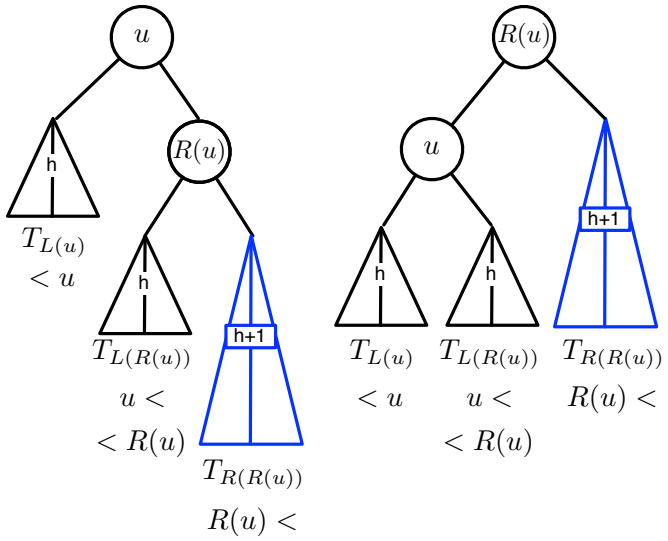

Figure 7.10: AVL case 4

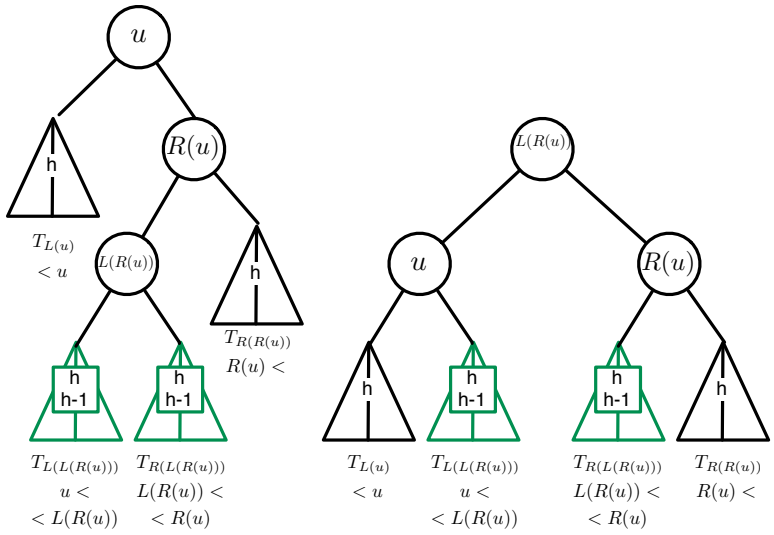

Figure 7.11: AVL case 5

To complete our discussion of addition and deletion in AVL trees, we first observe that each rotation can be executed in constant time. For addition, only a single rotation is needed, as the tree resulting from the rotation will have height equal to the height of the tree prior to the modification. However, for a deletion, the replacement tree might result in a new pivot node

being identified and consequently a subsequent rotation, as in Cases 1, 2, 4, and 5. The total number of rotations will be at most the number of nodes on the path from the deleted node to the root of the tree, or $O(\log n)$ due to the bound on the height of an AVL tree.

7.7.3 (2,3) trees

Another idea for reducing the height of a search tree comes from the observation that if instead of storing one data item in a node u, partitioning the values in T_u into those in the left subtree (values smaller than the key in u) and those in the right subtree (values greater than the key in u), if we store c data items in a node u, we can partition the values into $c+1$ subtrees (values smaller than the first key in u, values between the first and second keys in u, and so on, with the last partition being values greater than the cth key in u). In the extreme case of $c = n-1$, we could form a tree of height 1, though such a structure has two drawbacks: the cost of choosing among the n subtrees would no longer be constant, and implementing additions or deletions could be costly.

In the next data structure, we make use of nodes storing multiple data values (and hence multiple keys) in order to provide flexibility that can be exploited in maintaining a bound on the height while supporting additions and deletions. We define a $(2,3)$ **tree** to have two types of internal nodes: each has either one data item and two children or two data items (with the first having a key smaller than the second) and three children. Leaves are at the same depth, and each one contains either one or two data items. Search is possible as the following properties are maintained:

- for a node with one data item, keys of data items in the left subtree are smaller than the key in the node and keys of data items in the right subtree are larger than the key in the node, and
- for a node with two data items, keys of data items in the left subtree are smaller than the first key in the node, keys of data items in the middle subtree are larger than the first key in the node and smaller than the second key in the node, and keys of data items in the right subtree are larger than the second key in the node.

The operation Look_Up is then a minor modification of a search in a binary search tree, where instead of checking a single key at a node, two keys may be checked. More formally, the search starts at the root of the tree as the current node. If the current node is an internal node with one key, the sought key is compared to the stored key; the search terminates if the keys are equal, and otherwise either the left child or the right child of the current node becomes the current node, depending on whether the sought key is less than or greater than the key stored in the current node. If instead the current node is an internal node with two keys, the sought key is first compared to the first stored key, with the search terminating if they are equal and the left child becoming the current node if the sought key is smaller than the first stored key. Otherwise, the sought key is compared to the second stored key, with the search terminating if the keys are equal, the middle child becoming the current node if the sought key is smaller than the second stored key, and the right child becoming the current node otherwise. For a leaf node, the sought key is compared to each stored key, resulting in the key either being found or being determined to be absent from the tree.

We can implement ADD in a manner similar to that in binary search trees, using the fact that each unsuccessful search terminates in a leaf. The new data item can be added to the leaf, if there is space for an additional value to be stored. Otherwise, we add the new data item, resulting in **overflow** (a node having too many keys) and then execute one or more **split operations** to modify the tree so that it is again a (2, 3) tree.

Executing the split operation on a leaf ℓ entails replacing ℓ, which now contains three data items, by two leaves, one for the data item with the smallest key and one for the data item with the largest key. The data item m with the middle key now needs to be incorporated into a node at the level above the leaf. If the parent p of the leaf has a single data item and two children, then p can be modified to have two data items (including m) and three children (including both of the new leaves). In particular, if ℓ was the left child of its parent, m becomes the first key in p and the new leaves become the left and middle children of p; otherwise, m becomes the second key in p and the new leaves become the middle and right children of p. However, if the parent p of the leaf already had two keys, then we can temporarily add the new key and subtrees to p, resulting in another overflow and hence another split, this time at p.

The execution of a split operation on an internal node is similar to the operation on a leaf, with the additional task of allocating subtrees to the newly formed nodes. Due to the overflow, the node to split stores three data items and has links to four subtrees. We replace the node by two new nodes, one containing the leftmost data item and two leftmost subtrees and the other containing the rightmost data item and the two rightmost subtrees. The middle data item is then passed up to the parent of the node, where it in turn may result in a further overflow and split. The process will terminate either by encountering a node with only a single data item or by splitting the root to make the tree higher.

A similar idea is used in deletion, which as in the algorithm for the binary search tree, starts with a search. In this case, if the data item sought is not in a leaf, its inorder successor s is in a leaf, as otherwise there would be at least one value in the left subtree of s and hence between the sought data item and s, violating the definition of an inorder successor. In the latter case we exchange the node with its inorder successor so that deletion always starts from a leaf. If the leaf containing the data item to delete contains two data items, we can delete one and be guaranteed that the result is a $(2, 3)$ tree. Otherwise, deleting the data item results in **underflow**, a node with too few data items.

If the empty leaf has a sibling with two data items, then it is possible to rearrange the data items in the parent and siblings of the leaf to ensure that each leaf has at least one and at most two data items. Otherwise, we can **fuse** together two leaves to form a single leaf that contains the data items in the leaves as well as the data item in the parent that was used to choose between the leaves. If removal of the data item from the parent again results in underflow, we continue the process as far as needed, potentially reaching the root of the tree.

To determine the costs of addition and deletion, we observe that each split or fuse can be executed in constant time, and that the number of splits or fuses will be at most the height of the tree, which is in $O(\log n)$.

An idea similar to that used in the $(2, 3)$ tree can be used in the **B-tree** (Section 7.10.2), where now the maximum number of data items stored in a node is related to the size of a page, allowing for efficient use of external memory.

7.7.4 Hashing

The basic idea behind **hashing** is that when keys are digital data, a function (called a **hash function**) can be devised that maps each key to an index, called a **bucket**. Because more than one key can be assigned to the same bucket, the performance of various methods of hashing depends on how they handle **collisions**, that is, keys that map to the same bucket. Since most schemes can have terrible worst-case behaviour, hashing schemes are typicaly analyzed using average-case analysis; in some situations, randomization is used as a further tool to help avoid the worst case.

The two basic ways to handle collisions, or **collision resolution** methods, are **separate chaining**, in which all items that map to a particular bucket are stored there (such as in a linked list), and **open addressing**, in which only a single item is stored in each bucket. In open addressing, collisions will inevitably lead to items being redirected to other buckets, with the resulting increase in search time.

Search time is measured in terms of the number of **probes**, or comparisons with other keys; the average number of probes for a search is compared to the **load factor**, which is the total number of keys divided by the number of buckets.

Hash functions

To determine the mapping between keys and buckets, we make use of one or more **hash functions**, where a hash function maps any key value to one of the possible buckets (typically the integers 0 through $B-1$, for B the number of buckets). The goal of each hash function is to distribute keys as evenly as possible without requiring a lot of computation time. Here it is critical that keys be digital data on which such functions can be applied.

In defining hash functions, we view each key as a sequence of bits; in this manner, although the data item might be a string or an image, it can be interpreted as a number, and hence an input to a hash function.

Because the goal is to generate numbers in the range from 0 to $B-1$, two natural techniques to use are the selection of $\lfloor \log B \rceil$ bits to represent numbers in the range, or to use the function mod to use the value $k \bmod B$ in the range. Pros and cons of various options will be discussed in class.

In the remainder of our discussion on hashing, we use f to denote a hash function.

Separate chaining

In separate chaining, each key k is stored in the bucket $f(k)$, which is implemented as a linked list. We can thus implement Add in constant time by adding an item to the beginning of the list. However, the worst-case costs of Look_Up and Delete are both in $\Theta(n)$, as a linear number of keys may end up in the same bucket.

Advantages of separate chaining include the flexiblity in size of buckets and low cost for modifications. Disadvantages include not only the linear worst-case time but also the extra space needed to store pointers in a linked list.

Open addressing

In open addressing, at most one item is stored in each bucket. Associated with each data item is a **probe sequence**, the order in which buckets are to be searched in order to find the item or insert the item. We will consider various ways of choosing probe sequences.

If we consider the special case in which there are no deletions, then we can use the first empty slot encountered in a probe sequence as the location to add a data item (or, if in a search, as an indication that the item is not present). More sophisticated schemes can be used in which items in a probe sequence are stored in sorted order, so that the insertion of one item might entail the reinsertion of another item. For deletion, items can be rearranged or, to aid in search, a marker can be placed in a slot to indicate that an item had been present and was then deleted.

Open addressing does not require extra space for pointers, but there is less flexibility in the use of space than in separate chaining. In particular, the number of items stored is limited to the number of buckets.

One of the most straightforward ways of choosing a probe sequence is to start at the bucket $f(k)$ and, if that is not available, to check the next buckets in order, wrapping around to the first bucket once the last bucket has been checked. Such a probe sequence is known as **linear probing**, as it behaves like linear search. More formally, for B the number of buckets, the first bucket in the sequence is $f(k)$, and for any position p in the sequence, the next is calculated as $(p+1) \bmod B$. Unfortunately, linear probing often leads to **clustering**, the phenomenon in which once a collision occurs, more collisions tend to pile up in the same location.

To try to avoid clustering, instead of using an offset of one between subsequent buckets in each probe sequence, we might opt to use different offsets between different positions in the sequence. In **quadratic probing**, position i in the probe sequence (starting at position 0) is $(f(k) + i^2) \bmod B$. Although such a method does avoid having all the probe sequences cover the same subsequence, it still has the problem that if $f(k_1) = f(k_2)$, the probe sequences for k_1 and k_2 will be identical.

In **double hashing**, a second hash function is used to choose the offsets for a particular key k. In this scheme, position i in the probe sequence for k will be $(f_1(k) + if_2(k)) \bmod B$, where f_1 is the **primary hash function** and f_2 is the secondary hash function. Here we need to be careful to choose values carefully in such a way that each probe sequence visits all buckets; as discussed in class, we need to ensure that the secondary hash value and the size of the hash table are **relatively prime**, as defined in Appendix B.4. One easy way to do so is to choose the size of the hash table to be a prime number.

7.7.5 Skip lists

As the **skip list** data structure is presented only briefly at the end of the course, students are not expected to understand more than is presented in class.

The idea behind the skip list is to combine the best attributes of both linked and contiguous data structures, namely ease of modification and ease of search, using randomization to obtain good behaviour. Each data item is represented by a vertical linked list of h nodes, where h is the **height** of the item; there are also horizontal linked lists formed by all nodes at the same level. In addition, there are linked lists representing negative and positive infinity, with height greater than that for any of the data items. Randomization is used to select the height of a

newly added node.

Searching in a skip list is conducted by starting at the highest level, determining the largest key at that level that is no greater than the item being sought. If the item itself is found, either the search can terminate or, if additional data is stored with the node at the bottom level, the vertical linked list for the data item can be traversed. If the item is not found, then the same process is started at the next level down. Since all items are represented at the bottom level, eventually either the item will be found or the position in the skip list where it should be added will be located.

7.8 Data structures for priority queues

In class we focus on one data structure for priority queues, namely the **heap**, which is a special type of binary tree; the constraints on the heap allow it to be efficiently implemented using the array implementation of a binary tree. The heap is a complete binary tree such that the keys stored in each node satisfy the **heap-order property**: the key in a node is no greater than the key stored in either child of the node, if any. The combination of strict requirements on the shape of the tree and loose requirements on the placement of keys in nodes leads to efficient support for priority queue operations.

The heap-order property ensures that the minimum element is located in the root, allowing for a simple constant-time implementation of Look_Up_Min by returning the data item at index 0 in the array. Consequently, locating the minimum element for Delete_Min is trivial, with the challenge being to ensure that after its removal the remaining data items form a heap. Similarly, finding where to add the next leaf in a complete tree is trivial (it is at the array index one greater than the index storing the last leaf); the difficulty lies in making sure that the complete tree satisfies the heap-order property. For both Delete_Min and Add, we will augment the ADT Binary Tree to support an operation Swap_Node_Values, allowing us to express our algorithms as a sequence of such swaps.

The algorithm used to implement Add begins by placing the new key in the leaf after the last leaf, and then using "bubble up" to fix the keys along the path from the new leaf to the root by a series of swaps. Starting with the new leaf as the current node, we check the keys in the current node and its parent. If the key in the current node is greater than the key in its parent, we stop. Otherwise, we swap the keys in the current node and its parent, making the parent into the current node.

To see why the execution of "bubble up" ensures that the resulting tree satisfies the heap-order property, we first observe that if there is no change to the keys in the children of a node, then since the heap-order property was satisfied at the node before the addition, it would continue to be satisfied after the addition. We now consider the outcome of a swap, using C to denote the key stored at the current node, P to denote the key stored in the parent of the current node, and S to denote the key stored in the sibling of the current node. Since there was no change to S, $P < S$. Due to the previous swaps, now $C < P$. After swapping the keys C and P, the parent now contains C and the children contain P and S. Because $C < P$ and $C < P < S$, the heap-order property is satisfied at the parent.

To determine the cost of Add, we rely on the fact that the height of the tree is in $\Theta(\log n)$. Since a single swap can be executed in constant time (it involves the changing of a constant

number of pointers and data items) and the maximum number of swaps is the height of the tree, the total worst-case cost is in $\Theta(\log n)$.

We use a similar idea to execute DELETE_MIN: we remove the key at the root, move the key in the last leaf to the root to form a complete binary tree, and then swap keys until the heap-order property is restored. In "bubble down" we start with the root as the current node, repeatedly making adjustments until there are no violations to fix. To fix the problem at the current node, we examine the keys stored at the node and both of its children (or, if it has only a left child, the key stored in that child). If the key at the parent is not smaller than the keys stored in the children, then the data item in the child with the smaller key is swapped with the data item in the parent, with that child becoming the new current node. Since the key stored in the parent is the smallest of the three keys, clearly the heap-order property holds at the parent.

Using an argument similar to that for ADD, it can be seen that each swap can be executed in constant time, and hence the worst-case cost is in $\Theta(\log n)$.

For sorting using a heap, please see Section 8.3.2.

7.9 Data structures for multi-dimensional data

7.9.1 Quadtree

A **quadtree** is a data structure that works well for two-dimensional data, such as the pixels in an image. At each level in the tree, the space is divided into four quadrants, with a node representing each quadrant. Multiple levels can be used to place data items at distinct leaves of the tree.

The quadtree can be generalized to three dimensions by breaking up a three-dimensional cube into eight regions, resulting in an **octtree**.

7.9.2 k-d tree

A **k-d tree** makes it possible to tailor a tree to the placement of data items in space, which is useful in situations that result in inefficient quadtrees. To form a k-d tree, the space is repeatedly divided into two pieces, either horizontally or vertically, so that half of the data items in the space appear in each piece. The split is made either horizontally or vertically such that the dimension chosen has the larger range of values.

7.10 Other data structures

7.10.1 Trie

A **trie** (originally pronounced "tree" for "reTRIEval", now also pronounced "try") is a data structure used to store strings. By viewing a string as a sequence of characters, we can use the characters to search for a string in a tree. Each node in the tree may have a child for each possible character. Thus, a string of length ℓ can be found by searching through ℓ nodes, one for each character.

As an optimization, in a **compressed tree** paths through the tree can be compressed by observing that if a node has only a single child, the node itself is not required in the search. In such a data structure, it is possible that a string of length ℓ can be found by searching fewer than ℓ nodes, in particular when other strings are dissimilar, and would not share nodes encountered in the search in a trie.

7.10.2 B-tree

A **B-tree** is similar to a (2,3) tree (Section 7.7.3) in that each node may contain a range of values and have a range of children; in this case, the size of the range is related to the amount of information that can be transferred at once between different levels of memory.

7.11 Other criteria and analysis

Towards the end of the course, as time permits, we will briefly discuss various ways in which we design and assess data structures, according to different demands of various applications. For example, **kinetic data structures** are used to capture the idea of motion and **persistent data structures** store multiple versions simultaneously. The area of **succinct data structures** focuses on creating data structures that use as little space as possible.

When analyzing the worst-case costs of implementations of operations, we typically ignore the relationship among operations. In **amortized analysis**, we calculate the worst-case cost of a sequence of operations. When an expensive operation can only happen after a long sequence of cheap operations, such an analysis gives a more accurate representation of the cost of a sequence of operations, as it can be much less than the sum of the worst-case costs of each individual operation in the sequence.

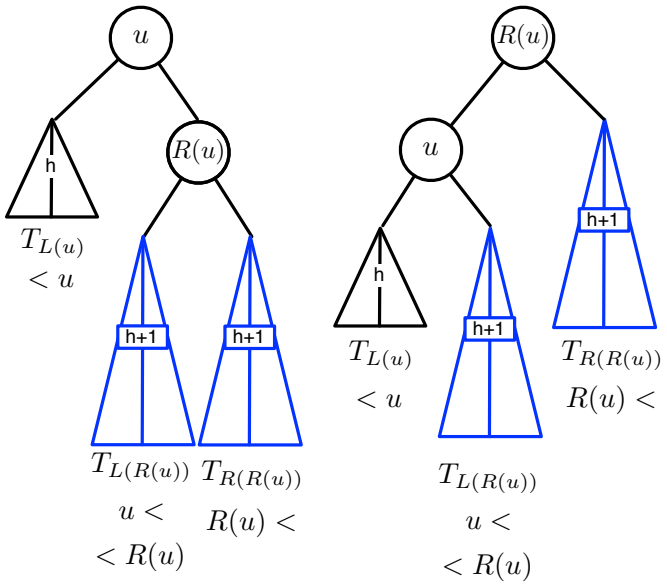

Figure 7.12: AVL case 6

Chapter 8

Algorithms that implement operations

8.1 Types of algorithms and results

When assessing an algorithm, both the algorithm and its analysis will depend on assumptions made about the data and the type of algorithm. When the sequence of steps taken by an algorithm on a particular input is fixed (as is the case in most of the course), the algorithm is **deterministic**. We will also briefly discuss using **randomization**, where the behaviour of an algorithm or structure of a data structure may depend on the outcome of various random choices. In the two remaining sections of the chapter, we focus on searching (Section 8.2) and sorting (Section 8.3).

8.2 Searching

8.2.1 Basic search algorithms

We observe that for many ADTs, multiple operations rely on being able to find a specified data item. Such operations may include searching for a data item, adding a data item (where the search may determine whether the data item is already present or where the data item should be added), and deleting a data item.

The cost of a search algorithm may depend on the type of data and the type of search. For example, when data items are orderable (Section 7.2), the placement of data items and the algorithm used may rely on the order. In addition, at times we will distinguish between the costs of a **successful search** (the search for a data item that is present) and an **unsuccessful search** (the search for a data item that is not present). In designing algorithms, we might focus on improving the cost of one type of search, perhaps at the expense of the other.

The algorithm **linear search** can be employed for any type of data, as it entails checking each data item in turn (typically in a linear order for a sequence-like structure) to determine whether it matches the data item being sought. When used on general data or on orderable data that is not stored in sorted order, it may be necessary to examine all the data items to determine whether or not a specified data item is present. If instead the data is orderable and stored in order, it might be possible to conclude that a data item is absent without examining all data items, since as soon as a data item larger than that sought is detected, it is clear that

the search is unsuccessful.

When data is orderable, it is possible to use **binary search** (covered in greater detail in the notes for first-year computer science courses) to repeatedly check the middle element in the remaining section to be searched. However, such a strategy requires not only that the data is orderable and stored in sorted order, but also that it is possible to access elements without scanning through from the first data item, implying the use of a contiguous rather than a linked data structure. As seen in the pseudocode for binary search presented in Section 2.2, the algorithm proceeds recursively by searching in a subsection of the data by comparing the data item being sought to the data item in position *Mid*, defined to be approximately half-way between the first and last positions in the sequence.

8.2.2 Interpolation search

The idea behind **interpolation search** is that if data items are reasonably evenly spaced, then the outcome of a previous comparison can be used to determine which comparison to make next. In contrast to binary search, in which at each step the sought value is compared to the element in the middle of the remaining sequence of values, here the choice of the position *Mid* is calculated based on where the data item being sought falls in the interval between the first and the last data item in the sequence.

In more detail, for *Low* the position of the first data item and for *High* the position of the last data item, we set *Mid* to be *Low* plus the "weight" multiplied by the size of the interval. The size of the interval is the difference between *High* and *Low*. The "weight" is the difference between the value in position *Low* and the value being sought divided by the total range of values stored in positions *Low* through *High*, where the total range of values is the difference between the values stored in positions *High* and *Low*.

For example, if we are using the ADT Indexed Sequence, then we could define *Mid* as follows, where the sought value is *Sought* and the indexed sequence is *I*:

$$Mid = Low + \lfloor \frac{(Sought - \textsc{Look\_Up}(I,\ Low)) \cdot (High - Low)}{\textsc{Look\_Up}(I,\ High) - \textsc{Look\_Up}(I,\ Low)} \rfloor$$

8.2.3 Search in a static array

As a special case, we consider the best arrangement of data items in a static unordered array, where we search items in order by position, from first to last. We first observe that an unsuccessful search always results in the search of all positions. To determine the best ordering for successful search, we observe that the number of items checked in order to find the data item in position i is $i + 1$; we use this number as the cost. Thus, if each item is equally likely to be searched, the average cost of a search will be the sum of the product of each position and the cost of searching for the data item stored at that position, or

$$\sum_{i=0}^{n-1}(i+1) \cdot \frac{1}{n} = \frac{1}{n}\sum_{i=0}^{n-1}(i+1) = \frac{1}{n}(\frac{n(n+1)}{2}) = \frac{n+1}{2}.$$

The calculation above uses the result, established in Appendix B.2, that

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

as well as the fact that $\sum_{i=0}^{n-1}(i+1) = \sum_{i=1}^{n} i$.

If we knew the probability distribution of the likelihood of searching for each item, we would be able to minimize the average cost by placing the items in order of nonincreasing probability. In that way, the most likely events would be the ones with the smallest costs, as they would entail comparing the smallest numbers of data items.

8.2.4 Self-organizing heuristics

In contrast to an algorithm, a **heuristic** is a problem-solving approach for which there may be no guarantee on either correctness or running time. For the purposes of searching for items in an unordered array, our goal is to try to minimize the average running time even when the probabilities of searching for data items are unknown. Our heuristics entail reorganizing data items after each search is executed, based on the assumption that a data item that was sought is more likely to be sought again.

In the **move to front** heuristic, after an item has been found, it is moved to the front of the sequence, whereas in the **transpose** heuristic, after an item has been found it is exchanged with the item that precedes it (if any). Known results, details of which are beyond the scope of the course, include the fact that the average case cost of move to front is no worse than twice the average case cost of search in the optimal static data structure, and that except in special cases (when there are only two data items or when all data items are equally likely to be sought), transpose performs better than move to front.

8.3 Sorting

8.3.1 Types of sorting

Sorting may play a role either in the use of an ADT, where ADT operations may be used to extract the data items in sorted order, or in the implementation of an ADT, such as for an operation that consumes a sorted sequence of data items and creates a new ADT storing those data items.

When the input data items may not be distinct, instead of there being one correct ordering of the inputs, there may be multiple ways in which identical data items appear in the output. A sorting algorithm is **stable** if it guarantees that the order among identical items is the same in the input and the output to the algorithm.

8.3.2 Heapsort

The **heapsort** algorithm consists of adding n items to a heap and then repeatedly executing Delete_Min to extract the items in sorted order. As there are a linear number of Add and

DELETE_MIN operations, each of which can be executed in time $\Theta(\log n)$ in the worst case, the total cost of the algorithm is in $\Theta(n \log n)$.

The initial construction of the heap can be improved by introducing a new operation HEAPIFY that consumes n items and constructs a heap that contains them all. We will show that in contrast to executing ADD n times, HEAPIFY can be executed in linear time.

The idea behind the linear-time execution of HEAPIFY is that first all data items are placed in the tree, and then, in a series of phases, values are rearranged to ensure that the heap-order property is satisfied. In particular, in phase i, pairs of heaps of height at most $i-1$ are joined with roots to form heaps of height at most i, where the heaps are formed from the nodes at levels 0 through i in the tree.

By observing that the leaves of the tree are at level 0 and each one forms a heap of height 0, we can begin our processing at phase 1, ending after a logarithmic number of iterations due to the logarithmic height of the tree. Each joining is equivalent to the bubble-down procedure, as for each heap being formed, the heap-order property holds in each subtree of the root but not necessarily at the root itself.

To determine the total cost, we need to figure out the cost of each phase, which entails the joining of pairs of heaps. To obtain an upper bound, we can assume that the height is a perfect binary tree. At phase i, our task is to join together pairs of heaps of height at most $i-1$ by giving each pair a new root to form a heap of height i. Each joining can be accomplished in time $O(i)$, as the bubble-down procedure can be executed in time linear in the height of the heap. In the details in the next paragraph (not required for the course), we can show that the number of joinings at height i is at most $n/2^i$. From this observation, we can then show that the total cost of all the joinings is in $O(n)$.

In the rest of this subsection are the details of the calculation; you are not required to know them for the course. The number of joining tasks is the number of subtrees of height i, which as discussed in Appendix B.5 is 2^{h-i} for h the height of the heap. As also shown in the same appendix, the number n of vertices in the heap satisfies $n = 2^{h+1} - 1 \geq 2^h$, and hence the number of joinings is at most $2^{h-i} = \frac{2^h}{2^i} \leq \frac{n}{2^i}$.

The total cost of all the phases can thus be bounded above by the sum, over all phases, of the number of heaps formed in a phase multiplied by the cost of one joining, or $\sum_{i=1}^{\lceil \log n \rceil} ci \cdot \frac{n}{2^i}$ for some constant c. Our sum can be expressed as $cn \sum_{i=1}^{\lceil \log n \rceil} \frac{i}{2^i}$; using the fact that $\sum_{i=0}^{\infty} \frac{i}{2^i} = 2$ (Appendix B.2), we can show that $cn \sum_{i=1}^{\lceil \log n \rceil} \frac{i}{2^i} \leq cn \sum_{0}^{\infty} \frac{i}{2^i} \leq 2cn$, which is in $O(n)$.

8.3.3 Bucket sort

Borrowing terminology from hashing (Section 7.7.4) and the idea of a bit vector (Section 7.3), we develop a way of sorting nonnegative integers in a fixed range. By creating an array entry, or bucket, for each possible value, we can sort the data items by processing each data item in turn, where for a data item with value i, we set array entry i to be 1.

In more detail, in **bucket sort** we first initialize all array entries to 0 in time linear in the size of the range. Next, we process each of the data items in constant time, at a cost linear in the number of data items. Finally, we can construct the ordered list of items by scanning through the array in order by index, adding an item i for each array entry i containing a 1; this final step can also be accomplished in time linear in the size of the range. Since the number of

data items is at most the size of the range, the total cost is linear in the size of the range.

Various variants of bucket sort have been considered in which multiple items are mapped to the same bucket, for example by having each bucket correspond to a range of values rather than a single value. In such situations, the data items in each bucket are sorted before the results of each bucket are joined to form the final output.

8.3.4 Radix sort

The **radix sort** algorithm makes use of the idea of bucket sort, forming an algorithm on digital data. Here, functions are used to split a data item into multiple pieces, and then to sort data items in a series of phases based on the pieces. By sorting in order from least signficant to most signficant information, by the end of all the phases, all the data items will be sorted in nondecreasing order.

As an example, suppose that the data items were all dates, including day, month, and year. Because the day is the least significant piece of information, we first sort on the basis of the day. After that, we sort based on the month, ensuring that we use a stable sorting algorithm to preserve the order of data items that resulted from the first phase. In the last phase, we sort by year. Figure 8.1 shows the ordering of the data after each phase, where the first column represents the input, the second column the ordering after sorting by day, the third column after sorting by month, and the fourth column after sorting by year. Due to the use of stable sorting algorithms, data items with pieces of the same value for one phase maintain the order established in the previous phase. Thus, 2 January 1900 appears before 4 January 2000 in the third column, retaining the order established in the second column, since they both have the same value of January in this phase.

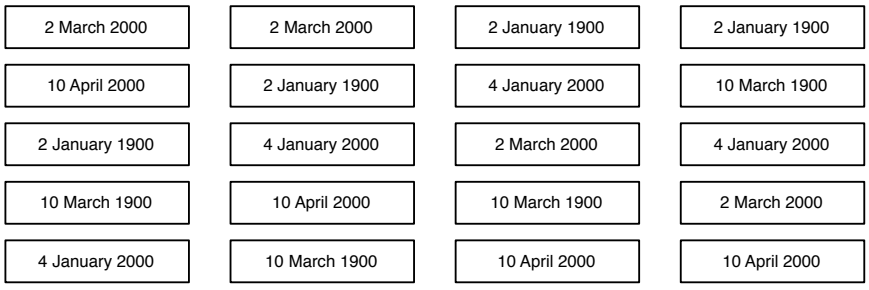

Figure 8.1: Sorting data items using different pieces of information

To analyze the cost of the sorting algorithm, we need to know the number of phases we use. In radix sort, we divide the bits into chunks. If we have n items, each of at most ℓ bits each, then to use chunks of b bits will result in a total of $\lceil \ell/b \rceil$ phases (or equivalently, $\lceil \ell/b \rceil$ chunks of size at most b). The cost of a single phase is the cost of bucket sort on n items using 2^b buckets, since b bits can express 2^b different numbers. The cost is thus $O(n + 2^b)$ per phase; by choosing b carefully, we can assume the cost is $O(n)$. Since the total number of phases is the number of chunks, the total cost is $O(n\ell/b)$ or $O(\ell n)$.

Appendix A

Other sources of information

You are also welcome (but not required) to use other sources, subject to the warnings below:

- Terminology may be not consistent from source to source; such inconsistencies may even apply to such basic terms as *abstract data type* and *data structure*.
- Different sources may use the same name for abstract data types that use different sets of operations, or different names for the same concept. In addition, the preconditions and postconditions of operations may differ from those used in the course.
- Material on data structures and algorithms may be interwoven. Although algorithms arise naturally in the implementations of operations supported by abstract data types, algorithm paradigms are covered in CS 231, not CS 234.
- Some sources are overreliant on the choice of a particular programming language, blurring the distinction between planning and coding.
- Algorithm analysis in the course will be based on a specific set of assumptions about underlying costs; this model (and the memory model) may vary from source to source. As the details are less important than the process, such sources may be useful in providing additional examples of the reasoning used in analysis.
- Various concepts covered in the course will use terms and categories that are unlikely to be found elsewhere, such as the roles of *user* and *provider* and the stages of *planning* and *coding*. In addition, the groups into which ADTs are placed for pedagogical purposes are unlikely to be presented in other sources.

The main utility of other sources is to find examples that demonstrate the concepts covered in the course, such as selection of an ADT, design of a data structure, and analysis of pseudocode. The details of the ADTs, data structures, and costs of operations are less important than the ideas used. That said, in any exam or assignment question for which the definition is not provided, you are expected to use definitions given in the course (either in class or here, as appropriate).

In the rest of this appendix, a few different sources are discussed.

A.1 Necaise textbook

Rance D. Necaise's "Data Structures and Algorithms Using Python", Wiley, 2011, was used as a textbook for the course until it went out of print. Errata for the textbook can be found at the textbook's student companion site, linked off the course web page on resources; further comments on its limitations can be found below.

Material to ignore

Many textbooks that teach this material using a specific programming language end up being driven by the language chosen; this often ends up resulting in a blurring of the distinctions between the planning and coding phases. We will de-emphasize Python lists and provide an alternate way of coding arrays.

Terminology to ignore

As a general guideline, if terminology is not introduced in lecture, you do not need to be concerned about learning it. Examples include the terms "simple" and "complex" as types of ADTs and the "base" of a stack.

The book uses the misspelling "psuedocode" instead of "pseudocode" and the phrase "run time" instead of "running time". Please do not emulate these errors.

For other errors, please see the textbook site. As one example, on page 22, "bagVector" should be "bagItems" in the figure.

Style to ignore

The Python code in the textbook does not conform to standard conventions. Examples of non-standard style include indenting by two spaces, adding extra spaces before and after parentheses, and using "magic numbers" instead of defining constants.

As noted on page 13, comments are often omitted to save space, so be aware that examples may not be complete.

Examples to ignore

Except for the purpose of exercises, for the most part the course will focus on standard ADTs. The special-purpose Student file ADT introduced on page 24 is a particularly confusing example, as the implementation depends on how a date is stored.

A.2 CS 240 textbooks and recommended books

Although we will not typically go as far into analysis as is done in CS 240, the same basic concepts are covered in both courses. Textbooks for CS 240 are good sources for examples of both ADTs and data structures, as are many other textbooks on the subject.

A.3 Handbook of Data Structures and Applications

For surveys on various topics by experts in the field, you may wish to consult the appropriate chapters in the "Handbook of Data Structures and Applications," available as an electronic resource at the University of Waterloo library. You can think of it as an encyclopedia: It can give you a sense of the breadth of the field while providing references for more depth. Most of the contents are beyond the scope of the course, but serve as a great starting point for explorations by those who wish to know more.

Appendix B

Math overview

As students in the course have varying amounts of background in mathematics, this appendix has been designed for those who could use strengthening of various mathematical concepts used in the course as well as those who may wish to see a deeper explanation of some of the foundational material used in the course.

B.1 Floors and ceilings

In expressing running times as a function of the number of data items, we often use a function $f(n)$ that is defined on all non-negative values n, without being concerned that for our applications, n is always an integer. However, when referring explicitly to the number of data items, we need to ensure that we are using an integer value. For example, when we are splitting an integer number of data items into two or more groups of roughly the same size, we might need a way to express "almost half" (or, more generally, "almost n/k" for various integer values k) as an integer.

If we have n numbers that we wish to split into two groups, then using $\frac{n}{2}$ as the size of each group works only if n is even. If instead n is odd, then one group will have $\frac{n}{2} - \frac{1}{2} = \frac{n-1}{2}$ numbers and the other will have $\frac{n}{2} + \frac{1}{2} = \frac{n+1}{2}$ numbers, for a total of $\frac{n-1}{2} + \frac{n+1}{2} = \frac{2n}{2} = n$ numbers.

To express both the even and the odd cases at the same time, we can refer to the sizes as $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$, the *floor* and *ceiling* of $\frac{n}{2}$, respectively. The floor and ceiling functions produce the nearest integer below or above, respectively, what is enclosed in the symbols. Thus, when n is even, $\lfloor \frac{n}{2} \rfloor = \lceil \frac{n}{2} \rceil = \frac{n}{2}$, since $\frac{n}{2}$ is already an integer. When n is odd, $\lfloor \frac{n}{2} \rfloor = \frac{n}{2} - \frac{1}{2}$, as that is the closest integer below $\frac{n}{2}$ and $\lceil \frac{n}{2} \rceil = \frac{n}{2} + \frac{1}{2}$, as that is the closest integer above $\frac{n}{2}$.

Because floors and ceilings "move" a value by less than one to reach the nearest integer, we know that $\frac{n}{2} - 1 < \lfloor \frac{n}{2} \rfloor \leq \frac{n}{2}$ and that $\frac{n}{2} \leq \lceil \frac{n}{2} \rceil < \frac{n}{2} + 1$. In particular, if we are using order notation, we can drop the floors and ceilings as we are only shifting the value by a constant.

B.2 Summation

In class we discuss a simple way of calculating the cost of a loop in pseudocode. When the costs of bodies of loops differ at each iteration, instead of taking the product of the number of iterations and the cost of each iteration, we instead calculate the summation of a series of

terms. Although in general such calculations can become quite complicated, in this course we will typically have relatively simple sums to calculate.

An *arithmetic sequence* is a sequence of numbers in which each number differs from the previous number by the addition of some fixed quantity. The simplest one, in which the quantity is 1, can be expressed as follows:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$$

In a *geometric sequence*, each number in the sequence differs from the previous number by the multiplication of some fixed quantity. The following sum is a simple example:

$$\sum_{i=1}^{n} c^{i-1} = 1 + c + c^2 + c^3 + \cdots + c^{n-1} = \frac{1 - c^n}{1 - c}$$

Sometimes you will remove a term from a sum, such as removing the term for $i = 1$ in the previous sum to obtain:

$$\sum_{i=2}^{n} c^{i-1} = c + c^2 + c^3 + \cdots + c^{n-1} = \frac{1 - c^n}{1 - c} - 1$$

When using sums, you can easily make a substitution of variables, such as setting $j = i - 1$ to show that

$$\sum_{i=1}^{n} c^{i-1} = \sum_{j=0}^{n-1} c^j.$$

More complex summations may require more complex techniques; although we will see an occasional summation in lecture, you will not be expected to solve anything more complex than those mentioned above. In the discussion below, you are not responsible for understanding omitted details.

In the remainder of this section, we discuss a result used to analyze the operation HEAPIFY used in heapsort (Section 8.3.2). The details given here are for those who are interested, and are not considered to be testable material for the course. We first observe that when $|d| < 1$, we can modify the geometric sequence to become infinite

$$\sum_{i=0}^{\infty} d^i = \frac{1}{d}$$

and we can differentiate both sides of the result to obtain

$$\sum_{i=0}^{\infty} i d^i = \frac{d}{(1-d)^2}.$$

By setting $d = \frac{1}{2}$, we thus obtain

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = \frac{\frac{1}{2}}{(1 - \frac{1}{2})^2} = 2.$$

B.3 Logarithms

All logarithms used in the course are logarithms base 2, which arise naturally in computer science; because of their prevalence, the notation log is typically used instead of $\log_2$. Intuitively, the logarithm of a number base 2 is the number of times the number can be divided by 2 before 1 is reached. Unless n is a power of 2, $\log n$ will not be an integer; if you need an integer value, you might need to use floors or ceilings, discussed in Section B.1.

The ceiling of the logarithm base two of a number is the length of its binary representation. To see why, you can observe that just like the digits in a decimal number give the quantities of powers of 10, the bits in a binary number give the quantities of powers of 2. When there are k bits, the numbers represented are 0 through $2^k - 1$, for a total of 2^k different numbers. To see why the largest value is $2^k - 1$, we observe that it is represented by k 1's, where each 1 corresponds to a power of 2. That is, the largest value is the sum (using the formula seen in Section B.2) $\sum_{i=1}^{k} 2^{i-1} = \frac{(1-2^k)}{(1-2)} = 2^k - 1$.

When discussing lower bounds, we will take the logarithms of various functions. Here are some formulas you might wish to use in manipulating logarithms:

- $\log xy = \log x + \log y$
- $\log(x/y) = \log x - \log y$
- $\log x^k = k \log x$

B.4 Mapping digital data

When handling digital data, we often wish to map numerical data items to a smaller range. To correlate the number of bits used to store binary numbers and the number of values that can be represented, we observe that if we have 1 bit to represent numbers, there are two numbers that can be represented, namely 0 and 1 by setting the bit to 0 or 1. If we have two bits, then we have two choices for the first bit (0 or 1) and two choices for the second bit (0 or 1), for a total of four different binary numbers. Using this argument, in general we can represent 2^k different binary numbers using k bits.

We can also use **modular arithmetic** to map data items to integers from 0 to $M - 1$, for a positive integer M. To map a data item a to an integer in the desired range, we use the **modulo operation** $a \bmod M$, which produces the remainder obtained when dividing a by M. As a simple example, when $M = 2$, the value of $a \bmod M$ is either 0, if a is even, or 1, if a is odd. The notation $a \equiv b \pmod{M}$ is used to show that a and b are **congruent modulo** M, that is, that $a \bmod M$ and $b \bmod M$ are equal.

In forming a probe sequence for double hashing, the effectiveness of the sequence depends on the relationship between the size of the hash table and the value of the secondary hash function, as discussed in more detail in Section 7.7.4. Two numbers a and b are **relatively prime** if their only common divisor is the number 1. It is not hard to see that any prime number p and any integer $q < p$ are relatively prime. However, pairs of integers do not need to include primes to be relatively prime: for example, neither 14 nor 35 is prime, but they are still relatively prime.

B.5 Trees

We use trees as the basis of many data structures. To assess a particular implementation, we are often concerned about the height of the tree (as it gives an indication of the number of data items to be searched) as well as the relationship between the height of a tree and the number of nodes or leaves it contains. At times we will also consider the number of nodes that appear on a particular level.

To determine the maximum number of nodes in a binary tree of a particular height, we consider a perfect binary tree. In such a tree, the number of nodes at each level is exactly double the number of nodes at the previous level, with 2^i nodes at level i. The total is then $\sum_{i=0}^{h} 2^i = \sum_{j=1}^{h+1} 2^{j-1} = 2^{h+1} - 1$; replacement of variables is discussed in Section B.2 and the solution to the second summation is discussed in Section B.3. The number of leaves of this tree is the number of nodes at level h, or 2^h. For ℓ the number of leaves, this tree has height $\lceil \log \ell \rceil$.

By considering the levels of the tree from top to bottom, we can determine the number of subtrees rooted at the nodes at each level of a perfect binary tree of height h. The root is at level 0, forming the root of the entire tree, a tree of height h. At level 1 are two nodes, each the root of a subtree of height $h - 1$. Continuing in this way, we observe that at level i there are 2^i nodes, each a root of a subtree of height $h - i$. At level h there are thus 2^h leaves, each the root of a subtree of height 0.

Suppose instead that we focus on the number of leaves ℓ, and wish to create the tree of minimum height that has exactly ℓ leaves. When ℓ is a power of two, we know the answer from our previous discussion. But if ℓ is not a power of two, we cannot place all leaves in the same level in the tree. Instead, we can fit all the leaves into the last two levels of a tree of height $\lceil \log \ell \rceil$.

B.6 Average-case analysis

At times we will consider the average-case behaviour of a data structure or an algorithm. Here we consider a generic way of looking at probability in terms of **events**. Each event is associated with a **probability** (a number in the range between 0 and 1) such that the sum of the probabilities of all events is equal to 1. As a very simple example, for a fair coin, the probability of heads is 1/2 and the probability of tails is 1/2, together summing to 1. The term **probability distribution** refers to the set of all probabilities for all events; if all probabilities are equal, as in our example, it is a **uniform distribution**.

We will often have a cost associated with an event, so that if we know both the probability distribution and the cost of each event, we can determine the **average-case cost** as $\sum_e Pr[e] \cdot Cost[e]$ where $Pr[e]$ is the probability of event e and $Cost[e]$ is the cost of event e. For example, if we are determining the average-case cost of an algorithm, for a particular input size, there is a probability and a cost associated with each input. Similarly, when considering a search algorithm, we can associate a probability and a cost with each data item.

Appendix C

Computer basics overview

In our assessment of the time and memory costs (using a model of computation to capture the essence of computation and a memory model to capture the essence of memory), we are making a rough translation from the ideas behind our data structures and algorithms to the physical properties of the underlying machine. In doing so, we are simplifying a complex, multi-stage relationship between idea and machine, focusing on the commonalities in hardware and software design rather than taking into account a myriad of details.

When you code an algorithm or a data structure, exactly how it performs depends not only on your choice of a programming language and the design of the machine on which you run your program, but also on several layers of software. The prospect of trying to analyze performance may seem daunting, given the number of details that are out of your control. The outline that follows is intended to give you a sense of ideas common across various designs of hardware and software, providing both a justification of the simplying assumptions made in our models and assurance that our way of estimating costs is justified. For a more in-depth discussion, please consider taking CS 230.

At the most basic level, the hardware of the computer can be seen as consisting of a **central processing unit** (**CPU**) and **memory**. We can think of the individual units of memory as each being big enough to store only a single bit (more specifically, as an electronic circuit using high voltage to denote 1 and low voltage to denote 0); we call such a unit a **cell**. Without going into technical details, we can think of each cell as having an **address** which indicates where in memory it is stored. In general we will be interested in grouping together cells to store more complex information. To reduce the amount of terminology we need, instead of being concerned with names for groupings of specific numbers of cells, we will use the term **chunk** to refer to any contiguous sequence of memory cells. (The choice of the non-technical term "chunk" is to avoid confusion with such terms as **word** and **block**, which have specific meanings outside of the scope of this course.) Depending on how memory is allocated, there might be different sizes of chunks used for different types of data. For simplicity, we'll assume that we have chunks that are big enough for what we want to store.

The role of the CPU is to follow a sequence of **instructions**, where both the input data and the instructions themselves are stored in memory. Although various hardware designs (or **architectures**) may differ in the choice of specific sets of instructions supported by the **machine language** (the language used to express the instructions read by the CPU), all seek to be as efficient as possible. Accordingly, operations that occur frequently can be executed quickly.

These include simple modifications to data, such as addition and subtraction, and simple ways of determining which instruction to execute next. A key idea used in processing instructions is that addresses can be handled like other types of data. An instruction is identified by its address, with the default being to execute the next instruction in order by address. It is also possible to navigate through instructions by making use of the addresses in a more complex way; for example, branching can be implemented by specifying which instruction to execute depending on the value of a condition.

In much of our analysis, we ignore the existence of different levels of memory, which range from **registers** (small chunks of memory that are stored within the CPU itself, and where all modifications by the CPU are executed) to **external memory** (large chunks of memory that are stored externally, such as on hard disks or in the cloud), with various other levels in between. For much of our computation, we will view the cost of accessing all memory as the same, covering situations in which the fine distinctions between faster and slower memory are not important. At other times, we may take into account the **memory hierarchy**, that is, the idea that the time to access a particular kind memory is inversely proportional to the amount of that kind of memory. For example, there are only a few fast registers but a lot of slow external memory. For most situations, it suffices to consider just two layers, typically viewed as **main memory** (or **primary storage**) and **external memory** (or **secondary storage**). Memory is moved between the two layers in fixed-size chunks (**pages**), where various **paging algorithms** are used to determine which page or pages to move out of main memory when a new page is moved in.

To manage the use of the machine by various programs and to translate from a high-level programming language (like Python) down to machine code, various intervening levels of software are used. A program is translated into a lower-level code by means of either a **compiler** (which translates the entire program at once and then executes it) or an **interpreter** (which translates and then executes one line at a time); without knowing details of such software, the writer of the program may not know exactly how the program is to be translated. In addition, the **operating system** has the task of managing the use of memory and allocating resources to various processes running on the same machine. Typically the writer of a program does not have control of the exact translation of lines of code to specific machine language instructions or to particular locations in memory. In most cases, programs can only access addresses given in **virtual memory**, maintained by the operating system, and have no information about the physical location of chunks of memory being used.

Appendix D

Python modules for data structures

The intention of the course is to teach the basic concepts of ADTs and data structures so that they can be applied in any programming language. The use of Python in the course has both advantages and disadvantages in the study of manipulation of data.

The idea of an ADT as an interface lines up nicely with the concepts of classes, objects, and methods; the user of the ADT can use methods without knowing how they are implemented, and the provider of a class and methods can create them without knowing how they are to be used.

More challenging is the implementation of a data structure using contiguous or linked memory. Python has been designed to support sophisticated data types such as the Python list and the Python dictionary rather than to allow access to the memory at a bit-by-bit level. (Note: Because both lists and dictionaries are common types of ADTs, the prefix "Python" will be used whenever the reference is to a Python data type instead of an ADT.)

To allow the design of data structures in a straightforward way, you have been provided with modules that simulate the manipulation of contiguous and linked memory. The modules `contiguous.py` and `linked.py`, described in the next sections, are available on the course website. Although these modules will in fact be implemented using Python data types, you will be able to simulate more direct access to memory by restricting your access to the data by using only the operations provided by the modules.

In assigning costs to operations in Tables D.1, D.2, and D.3, we are deliberately ignoring how memory is used in Python, as our intention is to be able to design data structures independent of the programming language being used. We instead assign costs based on what the costs would be if we were able to manipulate data more directly. Throughout the course, we assume that each of the following are constant-time operations:

- Allocating a chunk of memory of any size (but not initializing it)
- Using a pointer to move to a chunk of linked memory
- Writing a pointer
- Using an index to find the location of an item in contiguous memory
- Reading or writing a simple value such as a number or string

In contrast, the cost of initializing newly-allocated memory to empty will be linear in the size of the chunk.

D.1 Contiguous memory

The module `contiguous.py` provides the class `Contiguous` which can be used to simulate contiguous memory. Table D.1 indicates the methods and their worst-case costs; we use the term **size** to refer to the number of data items that can be stored in the contiguous memory. Note: For simplicity, the creation and initialization operation are joined into one.

Method	What it does	Cost
`Contiguous(s)`	produces contiguous memory of size `s` and initializes all entries to `None`	$\Theta(s)$
`repr(array)`	produces a string with the values in `array`	$\Theta(s)$
`array.size()`	produces the size of `array`	$\Theta(1)$
`array_one == array_two`	produces `True` if both arrays have the same length and the same values at each index, else `False`	$\Theta(s)$
`array_one != array_two`	produces `True` if the two arrays differ in length or the values at some index, else `False`	$\Theta(s)$
`array.access(i)`	produces the value stored at index `i` in `array`; requires $0 \leq$ `i` $<$ size	$\Theta(1)$
`array.store(i, value)`	stores `value` at index `i` in `array`; requires $0 \leq$ `i` $<$ size	$\Theta(1)$

Table D.1: Code interface for module `contiguous.py`

D.2 Linked memory

The module `linked.py` provides various types of nodes for linked memory. Table D.2 gives the methods and worst-case costs of methods for nodes with a single link each, defined using the class `Single`, and Table D.3 gives the methods and worst-case costs of methods for nodes with two links each, defined using the class `Double`.

Method	What it does	Cost
`Single(value, after)`	produces a node storing `value` linked to node `after` or `None`; if omitted, default value for `after` is `None`	$\Theta(1)$
`repr(node)`	produces a string with the value in `node`	$\Theta(1)$
`node.access()`	produces the value stored in `node`	$\Theta(1)$
`node.next()`	produces the node to which `node` is linked, if any, else `None`	$\Theta(1)$
`node.store(value)`	stores `value` in `node`	$\Theta(1)$
`node.link(other)`	links `node` to `other`, which can be a node or `None`	$\Theta(1)$

Table D.2: Code interface for `Single`

Method	What it does	Cost
`Double(value, after, before)`	produces a node storing `value` linked to next node `after` or `None` and previous node `before` or `None`; if omitted, default values for `after` and `before` are `None`	$\Theta(1)$
`repr(node)`	produces a string with the value in `node`	$\Theta(1)$
`node.access()`	produces the value stored in `node`	$\Theta(1)$
`node.next()`	produces the node to which `node` is linked using the `next` pointer, if any, else `None`	$\Theta(1)$
`node.prev()`	produces the node to which `node` is linked using the `prev` pointer, if any, else `None`	$\Theta(1)$
`node.store(value)`	stores `value` in `node`	$\Theta(1)$
`node.link_next(other)`	links `node` to `other`, which can be a node or `None`, using the `next` pointer	$\Theta(1)$
`node.link_prev(other)`	links `node` to `other`, which can be a node or `None`, using the `prev` pointer	$\Theta(1)$

Table D.3: Code interface for `Double`

Appendix E

Pseudocode style examples

Various styles of pseudocodes can be observed in these examples of the binary search algorithm. Not all of these examples are worth emulating, as some are too detailed and some are hard to understand.

Example 1 The Design and Analysis of Computer Algorithms, Aho, Hopcroft, and Ullman, 1974, page 114.

```
procedure SEARCH(a, f, ℓ):
if f > ℓ then return "no"
else
     if a = A[⌊(f+ℓ)/2⌋] then return "yes"
     else
          if a < A[⌊(f+ℓ)/2⌋] then
               return SEARCH(a, f, ⌊(f+ℓ)/2⌋ - 1)
          else return SEARCH(a, ⌊(f+ℓ)/2⌋ + 1, ℓ)
```

Example 2 Algorithms, Robert Sedgewick, 1988, p. 198.

```
function binarysearch(v:integer):integer;
   var x, ℓ, r:integer;
   begin
   ℓ:=1; r:=N;
   repeat
      x:=(ℓ+r)div 2;
      if v<a[x].key then r:=x-1 else ℓ:=x+1
   until (v=a[x].key) or (ℓ >r);
   if v=a[x].key
      then binarysearch:=x
      else binarysearch:=N + 1
   end;
```

Example 3 Data Structures and Their Algorithms, Lewis and Denenberg, 1991, p. 182.

```
function BinarySearchLoopUp(key K, table T[0..n-1]):  info
{Return information stored with key K in T, or Λ if K is not T}
     Left ← 0
     Right ← n-1
     repeat forever
          if Right < Left then
               return Λ
          else
               Middle ← ⌊ (Left + Right)/2 ⌋
               if K = Key(T[Middle]) then return info(T[Middle])
               else if K < Key(T[Middle]) then Right ← Middle - 1
               else Left ← Middle + 1
```

Example 4 Computer Algorithms: Introduction to Design and Analysis, Baase and Van Gelder, 2000, p. 129

```
int binarySearch(int[], E, int first, int last, int K)
   if (last < first)
      index = -1;
   else
      int mid - (first + last)/2;
      if (K == E[mid])
         index = mid;
      else if (K < E[mid])
         index = binarySearch(E, first, mid-1, K);
      else
         index = binarySearch(E, mid+1, last, K);
   return index;
```