Computer Science Book

Written by Bryce Summers (BryceSummers.com)

Last Updated: June 3, 2015

Contents

Part I

Fundamentals

Chapter 1

Mathematical Objects

To fully appreciate the material covered in this book, it is important that you are familiar with many mathematical objects, including sets, functions, orderings, numbers, relations, graphs, trees. At the present time, most purely mathematical material will not be explicity developed withing this book. The Author may write material regarding these subjects that will eventually be found at BryceSummers.com

Chapter 2

Description, Interface, Implementation, and Analysis

2.1 Chapter Overview

When thinking like a computer scientist, we often want to mimic the workings of reality, especially with regards to procedures or processes that modify the state of reality. Before we can effectivly mimic reality it is important to explicitly list several distinctions or perspectives on how a facet of reality may be perceived. In this Chapter, we will use the proccesss of a woman asking her husband to cook dinner as an example to help illustrate the distinctions that we will make. By understanding these distinctions now, you will be able to ease your mental workload for the rest of your computer science related life.

2.2 Descriptions

A description is a somewhat informal way of communicating. For example, a ceo at a large corporation use descriptions to describe their goals to their colleagues, where the colleagues would be responsible for acting on the intent and getting the desried results. Often Computer Scientists will interact with non computer scientists, such as the media, clients, users, and potential sponsers at the description level to convey the motivation behind their work. The description level could also be though of as the idea level.

2.3 Interfaces

An interface is a formal mathematical **specification** for how a given system should act. If the user of the system behaves in a certain way, the specification should provide a detailed description of what behavior the system should exhibit. Often interfaces are used to get multiple independantly produced systems to work together such as with the APIs that are becoming better known by the general public. Interfaces are also used to rigourously outline the workings of a describable idea in a form that can be understood at a later point in time. Many humans think of great ideas, but forget them because they did not write them down in sufficient detail that they could be recreated. Once an idea is written in such a form that

it can be understood in the same way by multiple independant humans, it may be called a proper specification.

Interfaces are the primary means by which information and details may be abstracted from the user of a system. (This is why the development of knowledge is of linear complexity and not worse!)

2.3.1 Types

In Computer Science, a type is a way of labeling groups of objects. For example, if I added two entities of type number, I would expect the result to also be a number, as apposed to a letter, chicken, or asparagus. We will be using types to describe specific objects and how objects of given types interact with each other. A Type can be thought of as a type of interface that describes what information and procedures the object may be used for.

2.3.2 Abstract Data Types

An Abstract data type (ADT) is an interface that describes the behavior of a given type. ADTs are commonly associated with mathimatical objects, such as functions, trees, graphs, sets, etc. Abstract Data Types are implemented by Data Structures, which we will discuss in the Implementation section. The main idea is that Abstract Data Types describe **What** a system should do and Data Structures specify **how** a machine acomplishes the behavior specified in th ADT.

2.4 Implementations

An implementation is a specification of the particular details of **how** a system will peform a given undertaking. An implementation is said to **implement** a given interface if it corectly provides detailed procedures the satisfy the expected behavior as described in the interface.

For example, say you were watching 1 mile foot race, every spectator knows that the athletes will be running 1 mile from the starting line to the ending line, that the athletes must start at the same time, and that the objective is for each athlete to be the first one across the finish line. These behaviors are the **interface** for a 1 mile foot race participant.

The particular details and strategies employed by a runner in the race, such as their speed, foot positions, breathing rate, aggressiveness, energy management, conditioning, height, weight, hydration, etc are are the **implementation** of a participant.

2.4.1 Data Structures

A Data structure is a very important concept in the realm of programming, wherebye an abstract data type is implemented correctly by a given program that can be executed on an actual computational machine. In this book, we will discuss quite a few data structures. They are mostly used to abstract common data management tasks that are used as subroutines in complex programs and help reduce the cognitive load on programmers and Computer Scientists interested in the analysis of algorithms.

2.5 Analysis

Analysis is the proccess by which we verify that a given implementation correctly implements a given interface. For example, banks rely on analysis to ensure that their monetary systems are secure from unlawful activity such as malicious programs that steal money.

In addition to purely correctness focused analysis, is the analysis of the amount of resources used for a particular task. Important resources in the realm of computer programming include the time a computation takes, the amount of memory a computation requires, and the amount of energy consumed. Analysis of this form is very important, because many correct algorithms are not useful if they cannot be run in a reasonable amount of time with the memory capabilities of an actual machine.

We will have a much more detailed discussion of different types of analysis in chapter 6.

2.6 "Cooking" Example

To conclude this chapter, we will properly investigate how these distinctions can be used to view a non technical scenario. For a thorough technical investigation of Arrays using these distinctions, please see Section 8.2.

2.6.1 Scenario

Please consider the common situation in which a woman asks her husband to cook dinner.

2.6.2 Description

The task is that of cooking dinner. The husband and wife each have likely eaten countless dinners in their life and both understand that making dinner is the proccess of producing food in an edible and pleasing form that may be consumed in the evening. The Wife merely needs to declare her intent in a conscise fashion using the word dinner for the husband to understand the intent. This is possible through the power of a shared definition.

2.6.3 Interface

In this example, the wife is said to be using her husband's interface. The husband's interface lists all of the actions that a husband is capable of performing such as that of preparing dinner. The details of how the husband decides on dinner, prepares it, cooks it, and the timings of his actions are abstracted from the wife so she does not need to worry about them. If the wife wanted more information, then we could extend the husband's interface to allow for him to answer questions and provide more details.

2.6.4 Implementation

The husband's implementation of the making dinner action would be the details and descsions that he makes, including a procedure for deciding whether a hamburger or a salad should be served, when to set the table, which beverages to serve, whether to serve dinner inside or outside, and all other relevant details pertaining to the action.

2.6.5 Analysis

To Analyze whether the husband makes an appropiate dinner in an appropiate manner would require us to look at the expectations of the wife and see if they are met by the husband's performance of the action, and in general by proving that the husband will always prepare the meal appropiatly given his internal descision procedures. In the real world, humans are not very likely to err at points in their lives, so it may be appropiate to only determine whether a paricular instance of the husband performing the action is correct or we could make statements about the probability of the husband succeeding at the action.

Chapter 3

Memory, Data, and Interpretation

3.1 Overview

In this section we will be developing the basics of how information is represented and interpreted on a machine. These ideas are of paramount importance to the Computer Scientists' world view.

3.2 bits

How does one differentiate between two things? In the real world, we are constantly differentiating between ideas, objects, people, etc every minute of our lives. A computer is essentially a machine that is able to differentiate and act upon a huge, but finite number of states. The basic building block of state information is the **bit**, which is defined to take on two distinct states, which we will call

$$1 \texttt{ and } 0$$

Depending on the context the states of a bit can be called alternative names such as *true* and $false$, but what is important is that the states are *different*. Bits become more and more powerful as they work *together* as you will see in the following sections.

3.3 memory

Actual Computers are capable of representing quite a few more than 2 states and they acomplish this representational feat by combining bits into larger structures. If 1 bit can represent 2 states, then if two of them are combined, then together they can represent $2 \cdot 2 = 4$ states. More explicitly, 2 bits can represent the following states:

$$00, 01, 10, \texttt{and } 11$$

It is important to note the these bits must be interpretted as acting together, because with an alternative interprettation the bits could be seen as independant unrelated entities. We will discover that in order to effectivly represent, modify, and interpret the states of a machine, it is very important to both think of combinations of bits and to think of a separation between

different groups of bits. **Memory** is an array of bits stored on a computer that represents the entirety of the current state of the computer. One can think of it as a very long line of boxes, each of which contains a bit in one of its two states, as in Figure 3.1 the boxes are called **memory locations**.

0	1	0	1	0	1	0	0	0	0	1	0	1	0	$\cdots$

Table 3.1: Memory Array

3.4 byte

A **Byte** is a group of 8 bits, which is capable of representing 2^8 different states. If we interpret each state as a binary number composed of bits representing binary digits, then bytes range from the state 0 to $2^8 - 1 = 255$. Please see Figure 3.2 for an representation of the number 100. Other primitive number sizes such as integers, shorts, and longs are represented by combinations of multiple bytes. Primitive number are the basic finitie sized numbers that are used as fundamental states in the execution of programs.

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	0	0	1	0	0

Table 3.2: A Byte Representing the number 100. $(2^6 + 2^5 + 2^2)$

3.5 2's Complement

Negative numbers are represented by mapping half of the possible states of a primitive number of size N to negative numbers between -1 and 2^{N-2} inclusive. The particulars of this mapping are up to the implementation, but a popular mapping is what is known as 2's complement arithemetic, where the numbers are represented such that when any two numbers are interpreted as positive numbers, added together, and then remapped to appropiate positive or negative numbers, then the result is correct modulo 2^N. One way to convert from a positive number to its negation is by flipping, i.e. changing the state, of every bit and then incrementing the number by 1 using base 2 arithemetic. Please see Figure 3.3 for an example of the conversion of 100 to -100. Of particular note, please see that $100 + (-100)$ overflows to 0!

100	0	1	1	0	0	1	0	0
~ 100	1	0	0	1	1	0	1	1
-100	1	0	0	1	1	1	0	0

Table 3.3: Bit representations of 100, bit flipped 100, and -100 ($=\sim 100 + 1$).

3.6 Booleans

Booleans are a primitive type supported by many programming languages that can take on 2 states, *true* and *false*. They are like bits, except for memory alignment reosons are usually implemented as byte sized or greater sections of memory that are restricted to only take on 2 states. It may be more memory and time efficient to use the boolean operations on larger types such as integers when trying to represent arrays of booleans in software, such as when using bit sets. Booleans are used to represent the results of any question or function that returns yes or no answers.

3.7 Pointers

Every memory location has an address that the computer can use to locate the memory location, much like mailmen use our home addresses to located our houses. A pointer is a primitive type that represents an adress. On a 32 bit machine, pointers take up 32 bits and can therefore address up to 2^{32} distinct memory locations, whereas 64 bit machines have pointers that are 64 bits wide. Pointers are used to form connections between disparate groups of data. One can say that a region of memory representing a pointer p **points to** the region of memory that is addressed by the value of p.

FIXME : Put in a simple diagram with an 4 bit computer.

3.8 Structures

Structures are types formed by placing several different types in a predetermined sequence. For instance I might make a structure containing 3 chars to represent a person's three main initials or a structure containing a pointer to a structure that represents a university and an integer representing the current school year. Structures will be very important to packaging data types into useful forms and allow for the creation of metaphors that ease the cognitive load on the programmer. Structures are called objects in the paradigm of object oriented programming.

3.9 Pairs

Pairs are simple structures that associate two typed objects. They are often used to represent simple utility objects, to represent connections, or to represent simple not quite primitive objects.

3.10 Static Memory Allocation

Also known as the Stack.

3.11 Dynamic Memory Allocation

Also known as the Heap.

3.12 Memory Deallocation

Garbage Collected vs. Manual Memory Deallocation

3.13 Chars

Also known as characters, chars are regions of memory the size of a byte that can represent states for various letters and other symbols used in the representation of text. As long as the computer interprets the charters state in the same way each time when it encodes, decodes, and displays the characters to the user on a visual display, it does not matter which bit states coorespond to which character. This has lead to many different encodings for sets of characters including the Ascii encoding and the Unicode encoding. Programmers type using english characters, instead of sequences of 1s and zeros, but the machine views their programs as sequences of characters.

3.14 Strings

Strings are implemented by a sequence of characters followed by a terminal character. The reason is that strings vary from the trivial string of size 0 to strings containing the entire works of William Shakespeare. If the languages were to package size information for a strings, then they would need to add a field to the beginning of every string that is of sufficient bit size to store the number cooresponding to the largest possible string. This would increase the amount of size required for all strings in existance and would not be very useful as most operations, such as printing and concatenation on strings will iterate through all characters anyways. If the size of the string is desired by the programmer, then it can be found in linear time by iterating through all of the characters until the terminal character is found and counting how many characters are seen. Strings do have an overhead of one character, because they need to allocate space for and store the terminal character. Because strings are implemented in this fashion, they require the presense of a dynamic memory allocator.

Chapter 4

General Definitions

4.1 Imperative Programming

Imperative programming is a programming paradigm wherebye programmer directly command (hence "imperative") the computer to modify regions of data to perform a computation. They do this by writing lines of code that act as instructions for how the computer should proceed at any point in time, including which instructions to execute next, which sections of memory to modify, and how to group lines of code into modular tasks called functions.

4.2 Purely Functional Programming

Purely Functional Programming is a programming paradigm whereby programmers specify Types and operations to map between types. The programmer does not give the computer instructions, but rather informs it as to how the decomposition between types work. A computation is thought of as the evaluation of a root function that may have many sub functions used in determining its output type. A hallmark of Purely Functional Programming is that their are no side effects, all input values to functions are unmodified, and all output values to functions are determined only as a function of the input values.

Chapter 5

Data Structure Definitions

5.1 Overview

When describing Data Structures, we will use terminology to help us to explicity defferentiate between the properties of various structures. You may wish to refer to Chapter 7 for descriptions of Abstract data types that may be used in this section to provide examples of the properties. You may also with to refer to Chapter 8.2 for a discussion of arrays, Chapter 8.3 for a discussion of lists, and Section 3.14 for a discussion of strings.

5.2 Ephemeral vs. Persistant

These terms are also know as mutable vs. immutable or Dynamic vs. Frozen repectively.

Let S and S' be the states of a data structure before and after an operation respectivly.

A data structure is said to be **ephemeral** if succeeding operations on S are not guranteed to be correct. Ephemeralness is a hallmark of data structures that permit operations that mutate the internal state of a structure, such as an oepration that replaces a the value of a byte with another byte. Traditionally, ephemeral data structures are associated with imperative programming, but they may be programmed in functional languages that are not pure in the sense that they allow mutations or side effects.

A data structure is said to be **Persistant** if both S and S' may be correctly used in future operations. Another way of saying this is that the data structure stores all of its previous states. Most well designed persistant data structures are capable of maintaining their history of states very memory and time efficiently and may be used in applications that require the same data to be used in different branches of execution.

5.3 Packaged vs. Unpackaged.

A data structure is said to be **Packaged** if it exists with some metadata or external interpretational knowledge that specifies information about the structure such as the locations in memory of all of its particular parts or how large the structure is. For instance, bytes can be viewed as packaged, because externally we know that all bytes are of size 8 bits and take up the contiguous region of memory starting at their address and extending for 8 bytes in order of increasing memory address. Arrays are also packaged, because they exist with

a given element size and a total number of bytes. They only work because we have these pieces of information.

A data structure is said to be **Unpackaged** if it exists without rigourous structural data. When viewing an unpackaged structure, it is impossible to determine all of the properties of the structure merely upon inspection. Most Programming languages implement strings as unpackaged structures, because they are defined as a sequence of char data fields followed by a terminal character instead of explicitly store the size of the string at the beginning. Please see the section 3.14 for more information regarding strings.

5.4 Indexed vs. Non-Indexed

A data structure of size S is said to be **Indexed** if every natural number between 0 and $S-1$ cooresponds to a distinct element that is contained within the data structure and an efficient lookup operation exists returns the proper element given one of these natural numbers. In this case, we generally consider efficient lookup functions to be those that run in constant time. The canonical example of an indexed collection would be arrays. I suppose a $\mathcal{O}(log(S))$ function would be efficient for an Augmented tree structure with stored tree size fields.

A data structure is said to be **Non-Indexed** if it is not said to be Indexed.

5.5 Finite vs. Infinite

A data structure is said to be of **Finite** size their is a finite bound to the number of elements stored in the structure.

A data structure is said to be of **Infinite** size if the structure theoretically represents a collection of an infinite amount of elements. One example of an infinite data structure would be a stream that represents the natural numbers and gives the user a incrementally larger number every time. Although it never explicitly stores all of its elements in memory, it is able to represent all of its elements through its mathematical procedure. Another property of infinite structures is that they are often iterable, but not entirly enumeratable. Infinite data structures are usually associated with explicit transformation functions and mathematical definitions.

Please see section 8.18 for more information on streams and 9.3 for more information about essentially infinite game trees.

5.6 Fixed Size vs. Resizable

A data structure is said to be of **Fixed Size** if it requires a constant amount of memory throughout its lifespan. Arrays and most primitive data types are of fixed size as well as some specilized versions of many of the more complex data structures. Electrical and Comptuer Engineers often use fixed size data structures for applications that lack a dynamic memory allocator and have constrained memory resources.

A data structure is said to be **Resizable** if it may enlarge of decrease its total memory requirements throughout its lifespan. Resizable structures require the presense of a dynamic

memory allocator

Please see sections 3.10 and 3.11 for discussions of static and dynamic memory allocation.

5.7 Ordered vs. Unordered

A data structure with size n is said to be **Total ordered** if there is a total order associated with the elements contained withing the structure and all of the elements are arranged in the structure in a way that respects this ordering. The presense of an ordering enables data structures to provide $\mathcal{O}(log(n))$ time complexity searchs through the use of binary search, whereas the fastest search time for an ordered collection is a linear search. Please see sections 9.1.1 and 9.1.2 for more information about basic collection searching.

A data structure is said to be **Ordered** if there is a well defined transversal for the data structure. In other words all iterations through the structure are guranteed to yield the same partial enumerations. Every element is also guranteed to be found at well defined locations throughout the structure. Most data structures that are ordered such as arrays and lists will have the oldest added elements nearer to the start of the structure than the youngest added elements.

A data structure is said to be **Unordered** if there there is no well defined transversal through the data structure and elements are not guranteed to be found at consistant locations. The primary example of such as structure would be a set, which cares only about inclusion rather than the relative locations of the elements.

5.8 Enumeratable and Iterable

These terms can mostly be used interchangably, but I have decided to define them in the context of this book in subtly different ways.

A data structure D is said to be **Enumeratable** if it can be converted into an indexable data structure such as an array. Naturally this translation would ideally be done in linear time. The output structure may be called the **enumeration** of D

A data structure D is said to be **Iterable** if it can provide an iterator that cooresponds to the current collection of elements inside of the structure. The order in which the elements are iterated through by the iterator coorespond to their positions in a **partial enumeration** of D.

Both the enumeration and partial enumeration of D will be ordered if and only if D is an ordered data structure. The enumeration of infinite sized data structures is infinite, so it may not be explicity produced on an actual finite memory machine. This is why infinite data structures may be used practically with lazy evaluation or streaming based algorithms or those that are of a mathematical nature that require properties of the sturcture, but not the structure itself.

5.9 Transversals

A **Transversal** of a data structure is the procedure for determining the order of its enumeration.

The most common transversal for an indexed data structure such as an array is to iterater from the lowest indice to the highest indice. There are countless other possible transversals possible, such as the reverse transversal or a transversal that visits every even element first and then visits the odd elements. You will find that transversals will be implicitly mentioned many times throughout this book, so please keep your eyes peeled.

Please see section **??** for a discussion of more complicated tree transversals.

5.10 Eager vs. Lazy

A data structure is said to be **Eager** if all computations are explicitly carried out in full at the moment its operations are called.

A data structure is said to be **Lazy** if it allows for the data structure to store operational requests and delay executing them until the user makes a request that requires an element to be revealed from the structure. For example, consider an implementation of a stream that represents all of the natural numbers. If the stream was implemented eagerly, then it would spend an infinite amount of time multiplying all of the natural numbers by 2, which would not be practical. If instead we implemented the stream lazily, then we could store the multiplication operation inside the stream and just multiply all numbers by 2 as they come out of the stream. Lazy evaluation is of paramount importance to computations on infinite data structures as well as aplications where the user does not need to query every element's state after an operation. Some functional programming languages use lazy evaluation to increase the efficiency of purely functional computations.

5.11 Simple vs. Performant vs. Efficient

An implementation of a data structure is said to be **Simple** if it can easily be read and understood by an programmer that did not write the code.

An implementation of a data structure is said to be **Performant** if it runs to completion relatively quickly compared to the standards of its real world time period.

An implementation of a data structure is said to be **Efficient** if its asymptotic complexity is competitive with the asymptotic complexity of the leading simple implementations of its time period.

In this book we will focus on simple and efficient data structures, because the author hopes to convey broad concepts to the reader. We will try to eschew data structures that are too complicated, because they are not as fun to program and to not shed as much light on the beautiful general themes of computer science.

5.12 Amortized versus Non Amortized

A data structure is said to be **non Amortized** if its operations are guranteed to run efficiently every time they are called. These data structure are safe to use in real time applications such as visual systems and Aircrafts, where an operation that takes too much time might jepardize the proper performance of the system.

A data structure D is said to be **Amortized** if it runs efficiently relative to the number of operations that it has performed. In other words an amortized data structure has efficient time complexities for each its operations averaged over all of the times the operations have been called. Amortization is a powerful way of viewing data structures that allows us to use many efficient data structures that would not be possible without amortization.

Please see section 6.6 for a discussion of amortized analysis.

5.13 Endogeneous vs. Exogeneous

A data structure is said to be **Endogeneous** if the values of the elements are stored withing the structure. For instance Arrays are endogenous because they directly contain the memory used to stor their elements. Endogeneous structures allow us to conserve space and ensure the locality of data by directly storing the element data within the structure. A data structure is said to be **Exogeneous** if pointers to the elements are stored within the structure, instead of the values of the elements themselves. This allows us to add the exact same memory addressed element to multiple structures and modifications to the element affect all of the exogeneous structures that the element has been added to.

5.14 Null-terminated vs. Null-node Terminated.

5.15 Sequential versus Parallel

A **sequential** computation is one that is run on one processor.

A **parallel** computation is one whose work is distributed to more than one processor. Paralllel computations can only be performed if the work of the computation can be decomposed into independant pieces or pieces that may be processed concurrently. Please see section 6.7 for a discussion of the analysis of parallel algorithms.

5.16 Concurrency safe.

FIXME : Expand this section or move it somewhere else.

5.16.1 Motivation

Sometimes we would like separate processors to modify the same state of a structure at the same time. These proccessors are said to be running concurrently. FIXME : Make sure to define concurrency and mention concurrent uniproccessors.

5.16.2 Mutual Exclusion

Mutual exclusion is a technique for ensuring the safety of concurrent modification to data by only affording one processor permission to make reads and writes at any given time. Some standardized techniques for implementing mutual exclusion are semaphors, mutexes, and locks. FIXME : Expand this a bit.

5.16.3 Lock-Free

If it is desirable for multiple processors to access a structure without waiting for permission, then lock free data structures may be implemented that ensure the correctness of the operations no matter which order the proccessors perform their concurrent modifications. These data structures are generally difficult to implement and require the use of atomic system provided operations such as compare and swaps.

Please see the following project repository for an example project using Lock-Free Union Find structures Project Repository: `https://github.com/Bryce-Summers/Randomized_Acyclic_Connected_Sub_Graphs`

Chapter 6

Analysis

6.1 Correctness Analysis

6.2 Resource Analysis

6.2.1 Models

6.2.2 Big O

6.3 bigO

6.4 Exact Analysis

6.5 Asymptotic Analysis

6.5.1 The Adversary.

6.5.2 Worst Case

6.5.3 Average Case

6.5.4 Best Case

6.5.5 Randomized

6.6 Amortized Analysis

6.6.1 Motivation

6.6.2 Banker's Method

Turnstile vs. Cash Register Model

6.6.3 Potential Functions

6.6.4 Potential Method

6.7 Analysis of Parallel Algorithms

Part II

Abstract Data Types and Data Structures

Chapter 7

ADTs

7.1 Overview

In this Section we will discuss the mathematical definitions of various Abstract Data Types. For particular implementations of these Types, please see the part of this book discussing Data Structures.

7.2 Function

7.2.1 Interface

7.2.2 Finite Functions

7.2.3 Infinite Functions

7.2.4 Real World Examples.

7.3 Stack

7.3.1 Interface

7.3.2 Real World Examples.

7.4 Queue

7.4.1 Interface

7.4.2 Real World Examples.

7.5 Deque

7.5.1 Interface

7.5.2 Real World Examples.

7.6 Priority Queue

7.6.1 Interface

7.6.2 Real World Examples.

7.7 Set

7.7.1 Interface

7.7.2 Real World Examples.

7.8 Multi-Set

7.8.1 Interface

7.8.2 Real World Examples.

7.9 Relations

7.9.1 Interface

7.9.2 Real World Examples.

7.10 Trees

Chapter 8

Data Structures

8.1 Overview

Throughout this part of the book, we will be developing many of the standard data structures use in imperative computation. There is quite a lot of beauty in the interelations between each of these structures. You will find that many of these structures provide that same capabilities albeit with different resource complexities and that many of the earlier data structures may be seen as special cases of the later structures. Of particular note are graphs, which are ADTs cooresponding to mathematical relations, which are sufficiently general that they can represent most of the other data structures presented in this book.

8.2 Arrays

8.2.1 Fixed Size Arrays

Description

Interface

Implementation

Analysis

8.2.2 Strings

8.3 Lists

8.3.1 Endogenous vs. exogenous

8.3.2 Single vs. Double

8.3.3 Linear vs. Circular

8.4 Unbounded Arrays (UBA)

8.4.1 Overview

8.4.2 Amortized Analysis.

8.5 Stacks

8.5.1 Interface

8.5.2 Array Implementations

Amortized Analysis

8.5.3 List Implementations

8.5.4 Persistant Stacks

8.5.5 Max/Min Stacks

8.5.6 Function Call Stack

8.6 Queues

8.6.1 Interface

8.6.2 List Implementation

8.6.3 Array Implementation

8.6.4 Fairness

8.6.5 Pipelining

8.6.6 Queuing Theory

Part III

Fixed Size Data Structures.

Part IV

Inefficient Data Structures

Part V

Obscure Data Structures.

Part VI

Algorithms

Chapter 9

Search Algorithms

9.1 Collection Searches

9.1.1 Linear Search

9.1.2 Binary Search

9.2 Graph Searches

9.2.1 BFS

9.2.2 DFS

9.2.3 A* Search

9.2.4 Weighted A* Search

9.2.5 A* recomputations

9.3 Game Tree Searches

9.3.1 MiniMax

9.3.2 AlphaBeta Pruning

9.3.3 Iterative Deepening

Bibliography

[1] Robert Tarjan, *Data Structures and Network Algorithms*, SIAM, Philadelphia, PA, 1983.

[2] Chris Okasaki, *Purely Functional Data Structures*, FIXME: publisher, location YEAR.

[3] The author received an Undergraduate Education in Computer Science at Carnegie Mellon University in Pittsbugh, PA.