The Hitchhiker's Guide to Data Structures

University of Rochester

Nathan Contino

April 8, 2016

Abstract

Introductory computer science students often have considerable difficulty extracting information from data structures textbooks with rather dense writing styles. These textbooks are usually intended to cover subjects *comprehensively*, but seldom focus on keeping content *comprehensible*. While this commitment to data preservation is admirable at heart, the author of this paper feels that we can do a better job for introductory students by attempting to balance comprehensibility and comprehensiveness. The Hitchhiker's Guide to Data Structures is an attempt to present data structures to introductory computer science students in this potentially more appealing manner.

Contents

1 Forward **2**

2 Introduction **2**

2.1 A Note on Running Time 3

2.2 A Note on Pointers and Values. 5

3 The Proto-Structures **7**

3.1 Nodes 7

3.2 Linked Lists 9

3.3 Arrays 14

3.4 Strings 17

4 Where Things Start to Get Complicated **19**

4.1 Doubly-Linked Lists 19

4.2 Stacks 23

4.3 Queues 26

4.4 Trees 29

4.5 Binary Search Trees 34

5 Advanced Data Structures **38**

5.1 AVL Trees 38

5.2 Binary Heaps 41

5.3 Graphs 43

5.4 Hash Tables 46

6 Summary **48**

1 Forward

Before delving into the subject of this paper, it's important to establish exactly what purpose this paper is designed to serve, and, indeed, to what audience we target this paper. The purpose of this paper seems simple, but has actually turned out to be rather difficult to convey, so we'll try to illustrate it clearly here: this paper is designed specifically as a guide on data structures for keen, interested students at the introductory level of computer science.

This guide is not designed to discuss the finest and most granular levels of data structures at a level that would satisfy a language or library designer. It also isn't intended to prove things in any formal way, as there are literally dozens of textbooks that fulfill that exact purpose. Instead, it is meant as a high to medium level overview of a broad spectrum of data structures. Proofs are omitted in favor of intuitions, and prose is kept informative and brief. Our hope is that a student in an introductory data structures course will find the writing style and approach of this guide more engaging, appealing, and concise than 500 pages of proofs and prose written by a professor who scarcely remembers what it was like to read a computer science textbook for the first time.

While this isn't necessarily a criticism of proofs and prose (as it's certainly necessary for pretty much anything in higher level computer science), my claim is that intuition and explanation might be a better way to teach introductory computer science, particularly in the case of students learning the material for the very first time.

2 Introduction

This paper is divided into a number of smaller, digestible chunks for the ease of the reader. Each of these chunks, in turn, is broken down into further chunks, many of which can be broken down into even smaller chunks, such as code examples and discussion, applications, important terms and definitions, and important or unusual methods. See the table of contents (page 1) for further details.

Some of these data structures, such as Nodes and Linked Lists, require very little algorithmic discussion, but a great deal of definitions, terms, figures, and code examples. Others, such as AVL Trees, will require significantly more algorithmic discussion, including examples and discussions of applications. Regardless, most sections follow the following form:

```
Data Structure Z
  <Fig. X: An artistic, hand-crafted example of the structure>
  Important Terms
  How Data Structure Z Works
  Code Sample
  Methods of Interest
  Uses of Data Structure Z
  Suggested Reading
```

These subsections allow us to cover the basics of each data structure whilst focusing on its important parts individually: as a result, certain data structures could have multiple subsections under any of the above sections, or no subsections at all.

Note that data structures falling under the section "Advanced Data Structures" lack a code sample – this is intentional. Code samples for AVL Trees, for instance, can easily take up around three or four pages of 12-point font, leading to an undesirable and unpleasant break in content. However, since programmatic examples are extremely helpful for complex data structures such as these, a "suggested code sample" has been provided in the "Suggested Reading" subsection, a section normally used to simply recommend interesting papers related to a data structure. It's also worth noting that all code samples have been provided in Java, since Java still represents the learning language of choice for many undergraduate computer science programs.

2.1 A Note on Running Time

Throughout this document, we will regularly reference the concept of "running time." There's no need to panic about big-O running time complexity and mathematical definitions here; instead, the reader only needs to understand a few basic running time ideas[22]:

1. Running time is not equivalent to wall-clock time (e.g. "my program took 3.25 seconds to exit") or CPU time (e.g. "the CPU was running my program for 1.337 seconds"). Instead, running time is merely a means of estimating the asymptotic complexity of a program, and gives the programmer an idea of how a program *scales* rather than the exact number of seconds that an execution will take for input of a given size. Beginners often confuse these three concepts, and not without reason – indeed, experienced programmers tend to use "runtime" as a term for all three, with meaning inferred solely from context! In this guide, however, you should expect the term "running time" to always refer to asymptotic complexity unless stated otherwise.

2. All running time estimates will be given as "worst case" estimates. That is; if we say that an operation takes "linear time", that particular operation could end up being much faster for select cases. However, our estimates account for the worst possible case, so actual running time should never exceed the running time estimate given here.

   If you're not sure what "worst case" running time estimates mean, picture the running time of a trip to Moe's Mexican Grill. While your average Moe's visit is likely quite brief – perhaps 15 minutes – your worst case Moe's visit involves a kitchen fire that delays the preparation of your delicious burrito by over an hour. A "worst case" running time estimate has to account for these kinds of edge cases.

3. You need only familiarize yourself with the following running times:

   - Constant: a flat cost (O(C)). Making one paper crane is a constant time operation, assuming that you already know something about origami. In this paper, we'll often use O(1) and O(C) interchangeably, since they mean effectively the same thing – a single "unit" of computation, regardless of input size.
   - Linear: The cost scales proportionately with input size (referred to as "n" throughout computer science) (O(n)). If you want to bake n batches of cookies, the amount of time you spend baking cookies scales linearly as n increases (assuming you're a robot who doesn't get tired).
   - Logarithmic: Specifically, base-2 logarithmic time (O(log n)). Basically, you can think of this as an operation whose efficiency doubles with each iteration. Let's say you're a robot designed purely for baking cookies. Every time you bake a batch of cookies, you get so much better that you can bake the next batch of cookies in half the time (let's not get caught up in real-world limitations here). So if you want to bake n batches of cookies, you can bake them in $\log_2(n)$ time.
   - Quadratic: A series of n operations, repeated n times. ($O(n^2)$). If you're familiar with programming basics, that can be nicely expressed by a doubly-nested for loop, where each loop iterates n times. A real world analogy: you have 50 books, all 50 pages long. Reading through all of these books requires reading 50^2 pages.
   - Exponential: x^n time, where x can be any real number ($O(R^n)$). This is the kind of running time that you really, really, really want to avoid, because it grows so quickly. If you're having trouble picturing this, think back to the cookie analogy – except, with exponential running time, your ovens are wearing out because you're baking an enormous volume of cookies. Because of this oven-wearing factor, each batch takes *twice* as long as the previous batch.

To put that in more formal terms: the k-th batch takes twice as long as the (k-1)-th batch.

If we want to visualize this running time in an appropriately data-structure-y way, we need only think of a tree. Let's say that a tree begins with its trunk, starting with the root. This trunk grows upward and eventually branches into two discrete branches. Each of those branches branches into two discrete branches of its own, which branch into two branches of their own, and so on and so on. The resulting tree will contain 2^n discrete branches, where n simply represents the number of times a set of branches branched.

2.2 A Note on Pointers and Values.

We shall use the term "pointer" in the following sections. However, the C-wary among you need not worry; understanding this "pointer" usage requires no mastery of ampersands and asterisks. Instead, you can simply think of the term "pointer" as a value that happens to point to another object (in this context, usually a Node), instead of a primitive value like an integer or a boolean. While this is intrinsically tied to the address vs. value notion that all computer science students should familiarize themselves with, understanding of that notion is by no means required for comprehension of this paper. Fig. 1 illustrates the relationship between pointers, addresses, and objects.

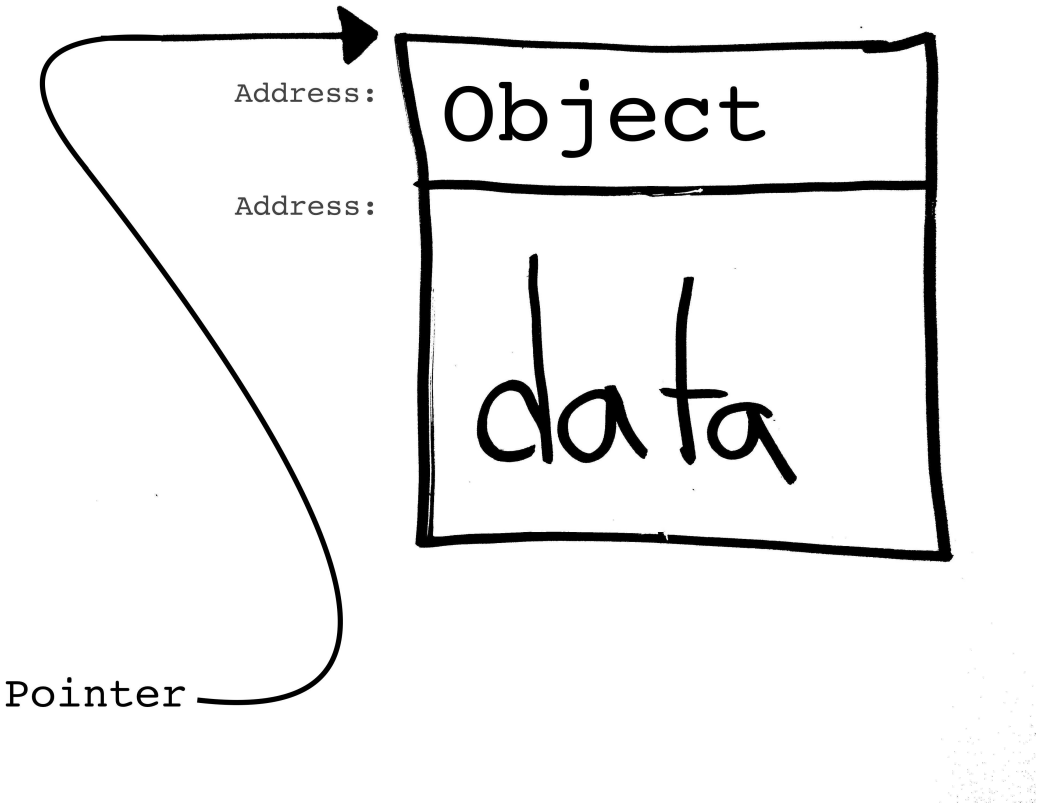

Figure 1: A pointer to a Node. Note that this really just means that the "pointer" object is just a reference to the address where the real object is stored in memory. In practice, pointers are used much in the way that a link to a Google Doc is used; instead of giving a "copy" of information, a method is given a "link" (that is, an address) to where that information is actually held in memory. If anyone changes or deletes the information at that location, the change effects *everyone* with a pointer to that location.

3 The Proto-Structures

"A list is only as strong as its weakest link."
-Donald Knuth[13]

3.1 Nodes

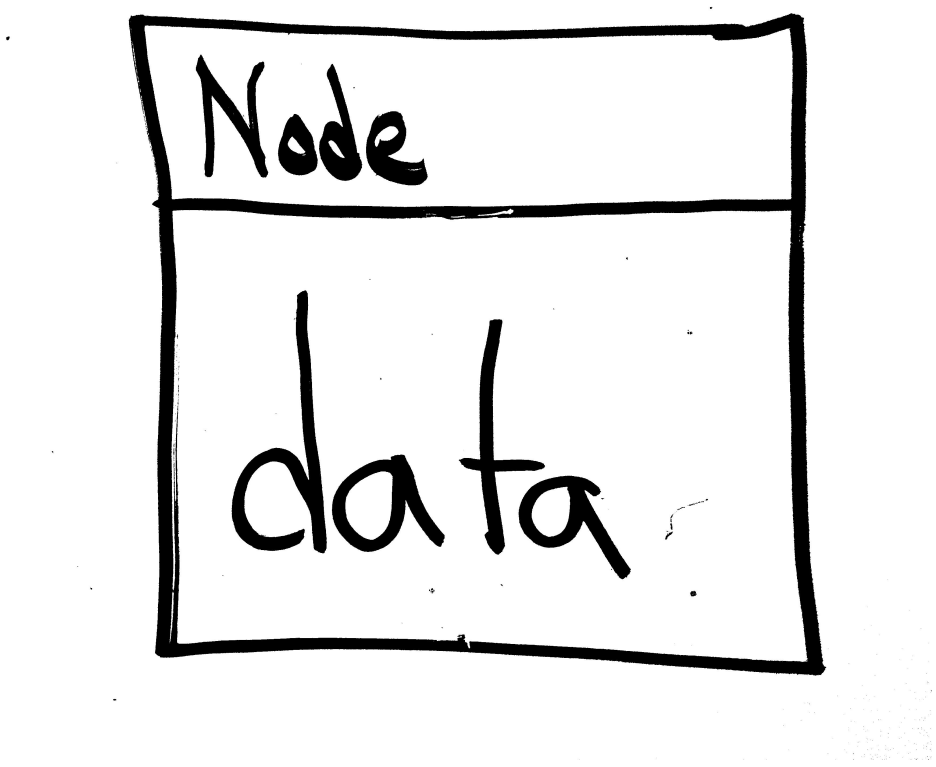

Figure 2: An artistic, hand-crafted example of the structure. Note that "Node" represents the object itself, and "data" represents a field *within* the Node object (see "Code Sample" for a programmatic version of this very same figure).

Important Terms

- Data: any piece of information that a data structure is intended to contain. This could be as simple as integers, or as complex as novels or entire accounts on a social media site. One of the primary concerns in data structure design is scalability– that is, the ability of a particular organization scheme to remain efficient even at massive sizes. If you're having trouble picturing what kind of growth this entails, picture the growth of Netflix's television and movie library, or Facebook's growth from a few thousand college students to over a billion worldwide users. A good data structure and associated algorithm doesn't break (that is, malfunction or slow to a

standstill) even when it increases in size from 10 pieces of data to 10 million pieces of data. See Fig. 2 for a visualization.

How The Node Works A generic node doesn't inherently contain anything other than a data element. While this may seem to simple to be interesting, nodes are not to be dismissed – they represent the basic building blocks of many data structures. Indeed, a data structure would be useless without the pieces of data it contains, since the primary purpose of a data structure is to enforce some kind of order on a set of elements of data. However, nodes are seldom as simple as class containing an integer data element; in linked lists, you'll commonly have a next field of type Node (3.2). In doubly linked lists, you'll find a prev (previous element) field of type Node, for backwards list traversal (4.1), in addition to the next field. Tree nodes can contain a wide variety of fields depending on the number of children allowed (4.4), and graph nodes will commonly contain linked lists or arrays of edges (5.3). However, there's no need to worry about this mere preview; we'll delve much deeper into all of these topics later.

Code Sample

```
public class Node{
  int data;
}
```

Methods of Interest Luckily for the reader, there aren't any methods that every Node class must contain. If you're studying data structures using Java, you'll likely want to include a basic constructor, but even that is not entirely necessary, as data can always be inserted manually after instantiation using the default empty constructor (which is precisely what you'd want to do if you were to use the above Node code example).

Uses of The Node Since nodes represent the most basic elements of most data structures, they have a vast multitude of uses. However, in isolation nodes are only capable of one thing: containing data.

3.2 Linked Lists

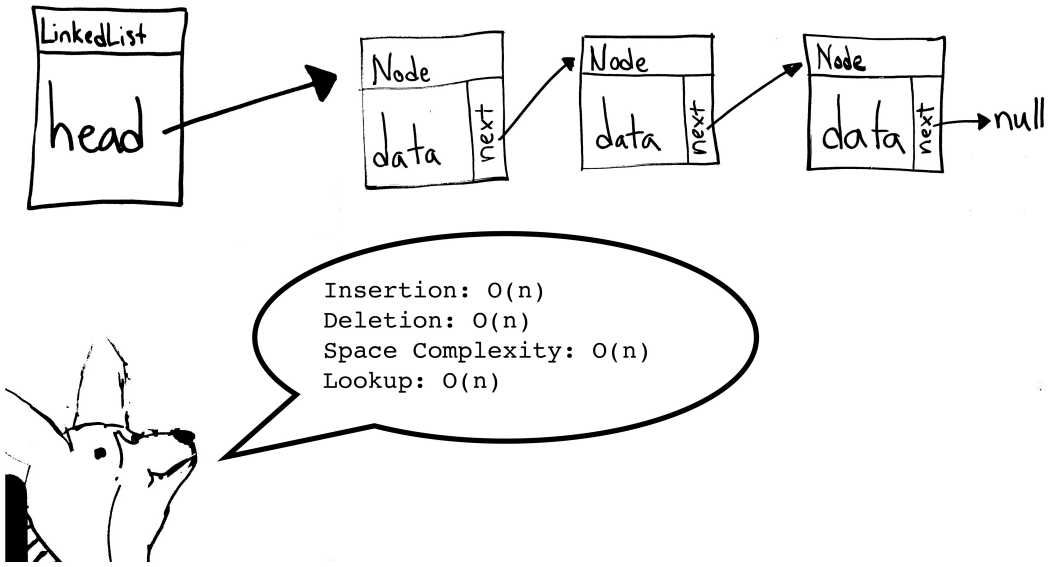

Figure 3: An artistic, hand-crafted example of the linked list. Note that this figure refers to insertion at the *end* of a linked list, which indeed requires O(n) time. Insertion at the beginning, on the other hand, requires only O(1) (constant) time.

Important Terms

- Nodes: simple data structures containing data and a pointer to the next node in the list. These pointers from Node to Node are the "links" in the "chain" of the list.
- Length: the number of nodes in the list with non-null data. The length of Fig. 3 is 3.
- Head: the first node in the list. In low level languages that lack classes (C, for instance), the pointer to this element is actually the pointer to the list as a whole. In Fig. 3, this is the leftmost node (the rightmost box, however, is the linkedlist object itself).
- Tail: the last node in the list. In Fig. 3, this is the element on the far right.

How The Linked List Works Linked lists represent perhaps the most basic data structure possible- a collection of unidirectionally linked nodes, each containing only data and a pointer to the next element of the list[19]. While this may seem like an overly simplistic concept, it represents a base from which many other data structures may extend functionality.

Linked lists are used to store arbitrary amounts of similar data. Linked lists have one chief advantage: scalability. That is, linked lists are capable of scaling from length 0 to length 50 to length 1000 to length 1 using insertions and deletions without ever copying the data elsewhere or reorganizing elements. This is a significant advantage over, for instance, arrays, which are created with a fixed size. If you create an array with size 100, and want to add a 101-th element, the only option is to copy the entire array into a larger array (a operation taking at least linear running time). But if you've got a linked list with 100 elements, you can insert a 101-th element in constant running time at the front of the list, and then you've got a linked list with 101 elements.

Linked lists do have some choice of design features, though. Some implementations assume a non-null head and tail at all times, to limit edge cases and null checks. Other implementations assume only a non-null head, or only a non-null tail. These non-null heads and tails contain no data and are referred to as "sentinel nodes." However, the sample implementation provided here assumes none of these- an empty instantiation of this linked list truly contains no nodes at all.

Code Sample

```
public class LinkedList{
  //in a basic linked list, we only need a reference to the head
  Node head;

  //insertion creates a node, points its 'next' element to the
  //current head, and then uses the new node as the new head.
  public void insert(int data){
    Node n = new Node();
    n.data = data;
    n.next = head;
    head = n;
    return;
  }

  /*  deletion searches for a node by checking data one element ahead
   *  of the current node. When we find data we want to delete, we just
```

```
     *  skip the deleted data in the 'chain' of pointers by replacing
     * our current node's next value with the deleted node's next value.
     */
    public void delete(int data, Node n){
      if(n.next == null)
        return;
      if(n.next.data == data){
        n.next = n.next.next;
        return;
      }
      delete(data, n.next);
      return;
    }

    //lookup- 'walks' the list, comparing lookup data to element data.
    //if element data matches lookup data, returns true. Returns false
    //only when the 'walk' reaches the end of the list.
    public boolean lookup(int data, Node n){
      if(n == null)
        return false;
      if(n.data == data)
        return true;
      return lookup(data, n.next);
    }
  }
```

Methods of Interest

- insert (O(n) or O(c) depending on implementation):. places an element into the linked list. Typically, implementations place elements at the front of the list since this makes insertion a constant time operation, and ordering doesn't matter. This is called prepending. However, many examples of linked list implementations for introductory classes insert elements at the end of the linked list instead, making this a linear operation. This is known as appending[19]. Some types of lists can provide insertion operations that are even better – sorted lists, for example, provide O(n/2) average insertion time [3]. The above code sample implements a O(C) insertion strategy.

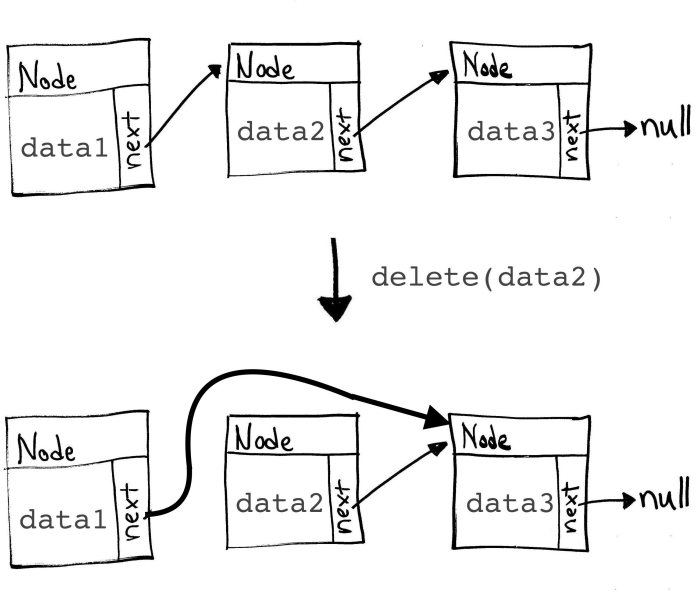

Figure 4: A delete operation on a linked list.

- delete (O(n)): remove an element from the linked list. In most implementations, this is quite simple: Picture elements A, B, and C in a linked list as follows:
  - A: node before the node you'd like to delete
  - B: node you'd like to delete
  - C: node after the element you'd like to delete

  In a barebones linked list implementation, these nodes will contain only two fields: the next Node and a piece of data. To remove element B, you simply have to make element A's next field point to element C[19]. This effectively "cuts" element B from the list, and you'll never have to worry about it again. This method takes linear time because if an element occurs at the very end of a list, you'll have to walk through every other element in the list to reach the desired element. However, there are some complications to this kind of "lazy" deletion strategy; for instance, this kind of deletion doesn't actually "get rid" of an element, so any pointers to said element will still function after deletion. In C, the offending element is typically "freed" from memory, an operation that simply tells your program "I'm not using this memory any more." However, Java provides no such option, leaving the burden of proper pointer handling squarely on the shoulders of the programmer. As a

general rule: keeping pointers to elements in a list is a dreadful thing to mix with deletion operations, and should generally be avoided. See Fig. 4 for an illustration of deletion in a linked list.

- lookup (O(n)): most implementations of a linked list will include some kind of method for detecting if a given piece of data is present in a list. This method is actually quite similar to the delete method, as it traverses the list starting at the head, checking to see if any element's data matches the data provided[19].

Uses of Linked Lists Unlike Nodes, which are really only useful for building other data structures, Linked Lists are useful for an incredible variety of applications. In general, however, linked lists are used whenever you don't know how many elements you might have to deal with, and there isn't an upper limit that could allow you to use an array instead.

Linked lists are often used as a basis for more complicated data structures with superior running time or more specific purposes, e.g. array lists, stacks, and queues. Advantages of the linked list include seamless deletion (you aren't left with a null element, as would happen in an array), effortless resizing, and a relatively small memory footprint compared to similar data structures like doubly linked lists, trees, and vectors, which are typically more pointer-reliant[10]. Disadvantages of the linked list include slow lookup times, slow deletion times, and difficulty of backwards traversal in the list[19].

Suggested Readings The following articles are suggested to readers interested in learning more about linked lists:

Paul F Dietz. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127. ACM, 1982

Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, 1984

K.L. Kluge. Method and apparatus for managing a linked-list data structure, July 13 1999. US Patent 5,924,098

3.3 Arrays

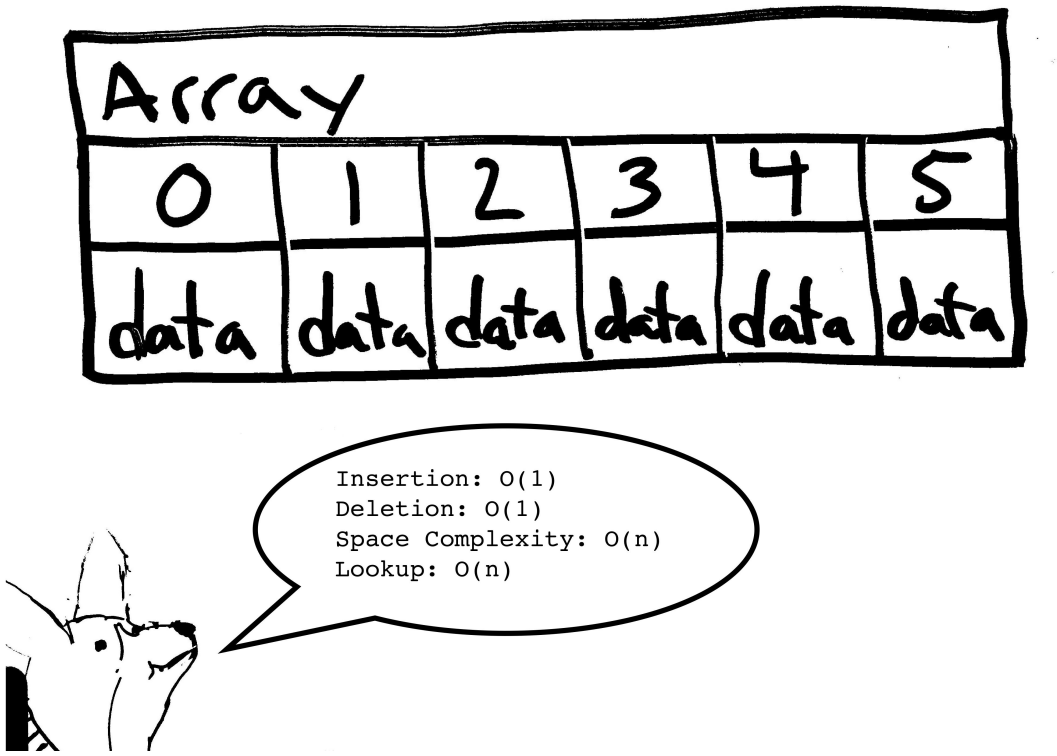

Figure 5: An array.

Important Terms

- Index: individual labels for each element used for quick access, used much like the "next" field in nodes of a linked list. In most imperative programming languages (including C, C++, Java, and many others), these indices begin at 0 and end at length-1. In Fig. 5, indices include 0, 1, 2, 3, 4, and 5.
- Length: the number of elements in the list. There are 6 elements in Fig. 5, so it has a length of 6.

How The Array Works Arrays are actually quite simple: let's say you have an integer called x in a program. Maybe that integer contains your favorite number. Your roommate, Steven, notices that you've got this nifty favorite number program, and wants to store his favorite number in it as well so he doesn't forget. You help him out and create another integer, cleverly named "x2", to contain his favorite number. He tells some friends about it, and soon enough half of your town is knocking on your door, asking you to store their favorite numbers in your incredible machine. But you don't want to store their numbers in individual integers- keeping track of that many names

could get messy. Instead, you cleverly solve the problem with an integer array, which is just a fancy way of saying: a bunch of integers automatically lined up one after one another in your computer's memory. This solves the naming issue through the use of a single name and an integer label (an index), like `favnumbers[2]` and `favnumbers[77]`.

While arrays are a perfectly legitimate data structure, they usually aren't implemented directly by the programmer- even in introductory computer science classes that revel in "teaching by doing." However, it is useful to know how arrays work. Essentially, your computer just grabs a chunk of memory of size `<lengthOfArray> * <sizeOfDataType>` when you create an array. When you access an individual element using the a[i] syntax, your computer is actually simply performing a quick calculation and accessing the memory found in the `a + i * <sizeOfDataType>` address of your memory space[22]. In this case, sizeOfDataType is measured in bytes, which is how every data type, from integers to BufferedImages, is actually stored in memory[22].

Code Sample

```
//create a new empty array
int[] a = new int[2];
//insert -11 into index 0 and 42 into index 1
a[0] = -11;
a[1] = 42;
//create an array with the same values in the
//same indices in a single line
int[] sameAsA = {-11, 42};
```

Methods of Interest

- Length (O(C)): Returns the length of the array. Typically, high-level language implementations of arrays store their length in a field so that it can be accessed in constant time. However, extremely lightweight forms of arrays – including those in the C programming language – don't store the length at all, forcing programmers to store the length of an array in some other way.

- Index access (O(C)): Technically, accessing a particular index of an array is also a method call (at least, in java): you can think of the a[i] syntax as the same thing as calling index(a, i). This simply returns the ith element of the array. Index access can also be used to store a value in a particular index. Lines 3 and 4 of the code sample above illustrate index access.

- Lookup (O(N)): Unfortunately, lookup in an array, as in many simplistic data structures, is still a linear time operation. You still have to traverse the entire array

to search for any individual element, so there's no avoiding this[10]. A sorted array can slightly improve the average running time prospects for a lookup operation, but the worst case remains O(N).

Uses of Arrays Arrays are used everywhere in computer applications. Whether they're being used to hold somebody's favorite numbers, pixel values in an RGB image, or even the calculations for a dynamic programming method, arrays are used pretty much anywhere that a programmer needs more than one or two of an element. While linked lists serve this same purpose, arrays really stand apart because of their constant-time indexing, which is extremely handy in many computational tasks where a programmer would like to avoid potentially linear linked-list access time. Arrays are of particular interest in vectorized languages, including Mathwork's MATLAB and Wolfram's Mathematica, which allow a programmer to perform scalar operations- that is, to apply a single function, like multiplication, to an entire array of elements[17]. Arrays are so common that we even have special vectorized hardware designed specifically for the manipulation of arrays in certain optimized languages and array implementations, like FORTRAN[20].

Suggested Readings The following articles are suggested to readers interested in learning more about arrays:

Edgar Chávez, José L Marroquín, and Gonzalo Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, 2001

Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011

3.4 Strings

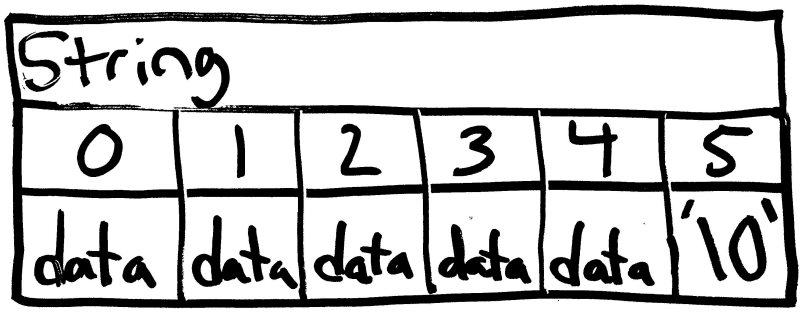

Figure 6: A String. Note the null terminator in the final bucket of the array.

Important Terms

- Length: how many characters occur in the string. Due to the null terminating character, C-style strings have a length of `<numCharacters> + 1`. Strings may be of variable or fixed length depending on programming language[10].
- Null terminator: '\0' is used to indicate the end of a string in C-style strings. Some programming languages omit the null terminator entirely[22]. In Fig. 6, index 5 contains the null terminator.

How The String Works A string is really just a highly specialized array that contains only characters. Interestingly enough, Java Strings are immutable – any time even a single character is added to a string in java, all of the string's contents have to be copied over into another array of slightly larger size[27].

Code Sample

```
//create some strings
String myStr = "All of these worlds are yours-";
```

```
String myWrn = "except Europa. Attempt no landing there.";
String cat = myStr + myWrn;
```

Methods of Interest

- Concatenation (O(n + m)): Appends one string of length n to another string of length m. This is done by creating a new array of length n+m, copying the first string into the first n indices (sans null terminator, if the language uses one), and then copying the second string into the last m indices. In the above code sample, `cat` is the concatenation of `myStr` and `myWrn`.

Uses of Strings Inevitably, even the most stubborn programmer will eventually find himself forced to output text in some way or another. Any time you interact with text in a program, you're working with Strings. Strings also have the advantage of easy, typically built-in conversion to most other data types. For this reason, Strings are many language's type of choice when accepting *runtime arguments*, input given to a program when it's run. For instance, if you're running an instance of a java class named "Test", you can simply run the program like so:

```
java Main myFirstArgument mySecondArgument 42
```

This runs the program with "myFirstArgument" as the first argument, "mySecondArgument" as the second argument, and "42" (note that this is a string, not an integer) as the third argument. In java, you can access these arguments as args[0], args[1], and args[2], respectively. Conversion of strings to integers (and, indeed, most other data types) is built into most high-level languages, making data manipulation a simple matter from here.

Suggested Readings The following articles are suggested to readers interested in learning more about strings:

Stuart E Madnick. String processing techniques. *Communications of the ACM*, 10(7):420–424, 1967

4 Where Things Start to Get Complicated

"Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."

-Donald Knuth[13]

4.1 Doubly-Linked Lists

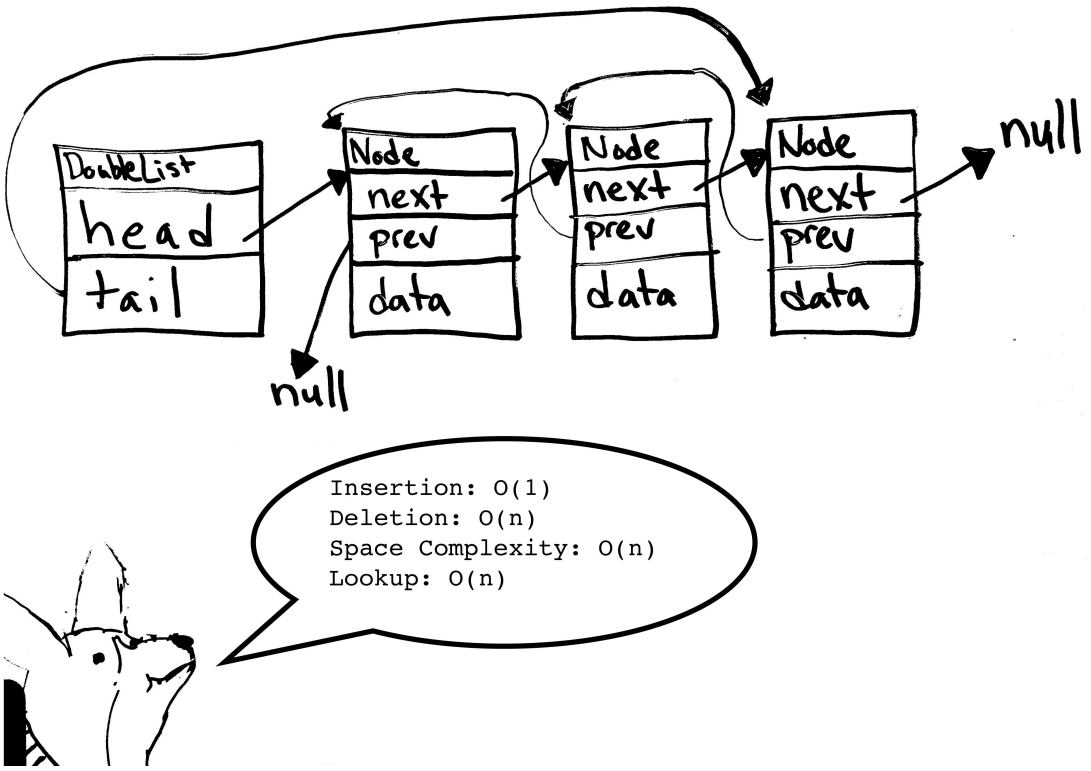

Figure 7: An artistic, hand-crafted example of a doubly-linked list. Note the "prev" field in each node.

Important Terms

- Prev: pointers to the node occurring before the current node in the list. In Fig. 7, you can see that the first element of the list contains a null prev pointer, much like how the last element of the list contains a null next pointer.

How The Doubly-Linked Lists Works A doubly linked list is really just a simple extension of a standard singly linked list. Instead of nodes connected only by forward-facing "next" node connections, a doubly linked list also contains backward-facing "previous"

node connections. Thus the list can be traversed in either direction – starting at the head and ending at the tail, or starting at the tail and ending at the head – or, indeed, starting anywhere in between the head and the tail and proceeding in either direction! While this choice doesn't necessarily make it any easier to access arbitrary elements in a list (since the elements in the middle will still take n/2 movements to access, which is still linear time), it does open up the possibility of constant-time access of the tail. However, the most useful aspect of the doubly linked list isn't access time for elements; instead, doubly linked lists are generally considered more useful because backwards-facing traversal can make many operations a great deal simpler and a great deal faster for certain data structures that extend list functionality, like queues (4.3).

However, doubly linked lists are not without their costs. Perhaps the most significant cost lies in the space taken up by the additional "previous' pointer in each node in the list. This seems insignificant for a small list, but consider the amount of space that those n pointers will take up for a linked list of a million elements. Doubly linked lists also have the downside of additional maintenance- deleting the i-th element suddenly takes more work, since not only do you have to modify the i-1th element's previous pointer- you also have to modify the i+1th element's previous pointer[10]. While this operation retains the constant running time of the singly linked list, it becomes much more complex and error prone from a coding perspective.

As a result, operations as simple as sorting a doubly linked list become more tedious to write than their singly-linked counterparts.

Code Sample

```
public class DoubleList{
  Node head;

  //inserts an element of data in a new node
  //at the head of the list
  public void insert(int data){
    //create a new node in which to store our data
    Node n = new Node();
    n.data = data;
    //this node points to the head
    n.next = head;
    //as long as head exists, point it back to our new node
    if(head != null)
      head.prev = n;
    //our node becomes the new head
    head = n;
```

```
      return;
    }

    //walks through the list, searching the NEXT element for
    //matches with our data
    public void delete(int data, Node n){
      //cover the special case of empty list
      if(n == null)
        return;
      //cover the special case of head contains data
      if(n.data == data){
        head = head.next;
      }
      //if we're at the end of the list, we didn't find this data
      if(n.next == null)
        return;
      //but if we get a match
      if(n.next.data == data){
        //splice n.next out of the list by altering
        //the previous pointer in the node after n.next
        if(n.next.next != null)
          n.next.next.prev = n;
        //and the next pointer in our current node, n
        n.next = n.next.next;
        return;
      }
      //if we haven't found the element yet or terminated, keep searching
      delete(data, n.next);
      return;
    }

//lookup walks the list, returning true if any piece of data matches data
    public boolean lookup(int data, Node n){
      if(n == null)
        return false;
      if(n.data == data)
        return true;
      //'walk' the list here by recursively calling on next
      return lookup(data, n.next);
```

```
    }
}
```

Methods of Interest The standard doubly-linked list uses the same functions as the singly-linked list – insertion, deletion, and lookup – but with slightly more overhead, since the doubly-linked list functions have to account for prev pointers in addition to next pointers, and sometimes update the doubly-linked list's tail pointer (if such a pointer is used) [10]. Simply compare the linked list code sample to the doubly-linked list code sample to see just how large a difference these back pointers make in the insert and delete methods.

Uses of The Doubly-Linked List Perhaps the most compelling usage of doubly linked lists exists in the construction of queues (4.3). Since queues typically require access to both the front and the back of a list, it's helpful to have prev-pointers – that way, you can insert at one end of the list and delete from the other end of the list with both operations costing constant time. This is also possible using arrays, but with a catch- arrays have a set size, and queues can vary wildly in size, making doubly-linked lists a more suitable choice.

In a singly linked list, one of these operations would have to cost linear time[22]. If you're having trouble picturing this, consider deletion from the end of a singly linked list. Even if you keep a pointer to the last element in the list, you'll have to traverse the entire list after every deletion to get a fresh pointer to the new last element in the list, making deletion a linear time operation no matter what.

This ability to traverse a doubly linked list backwards in addition to standard forward traversal allows one to treat the head and the tail of the list similarly, since you can walk from the tail to the head, or from the head to the tail with the same running time. This is an exceptionally powerful tool, as we'll see later when we investigate data structures built using doubly linked lists, such as the queue.

Suggested Readings The following articles are suggested to readers interested in learning more about doubly linked lists:

Mary E d'Imperio. Data structures and their representation in storage. *Annual Review in Automatic Programming*, 5:1–75, 1969

4.2 Stacks

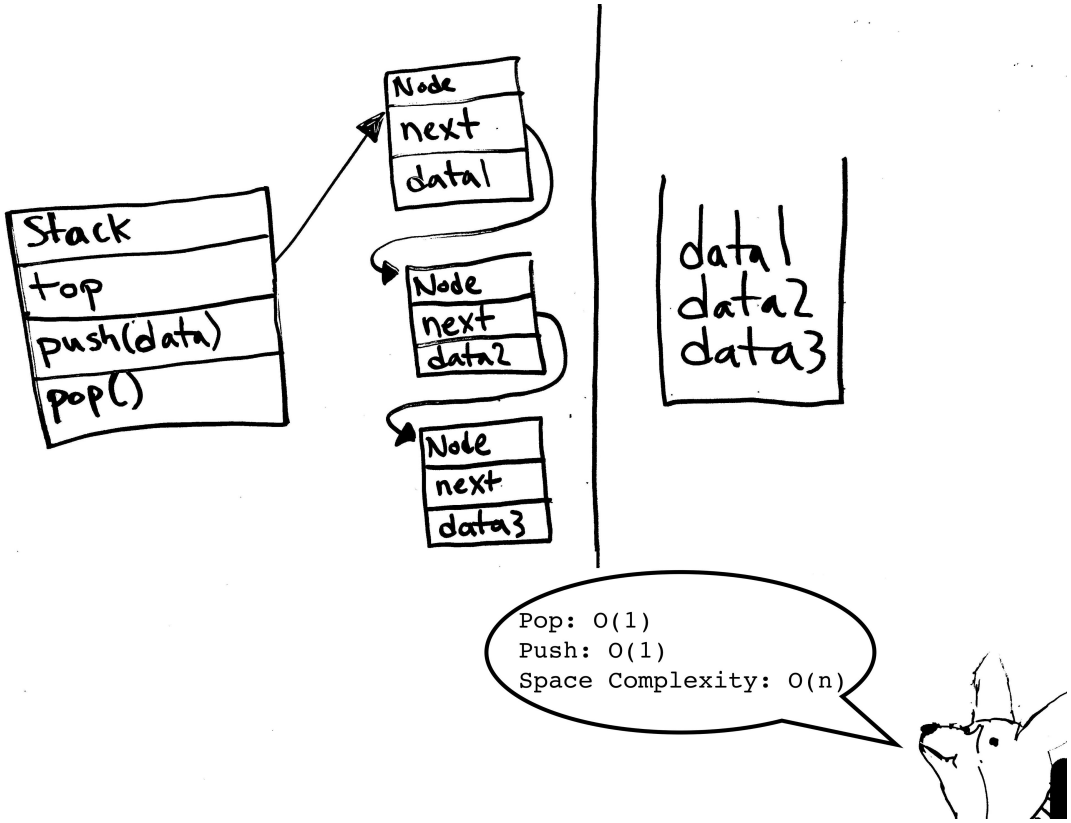

Figure 8: On the left: a diagram of a stack in memory. On the right: a common, more concise representation of the same stack.

Important Terms

- LIFO (Last In, First Out): describes a data structure in which elements are removed in the opposite order in which they were added. In Fig. 8, data1 is at the top of the stack, and thus it was the "last in" element– that is, the last element added to the stack via the push method. It is also currently the "first out" element– that is, it will be the first element returned if the pop method is called.

How The Stack Works Stacks are commonly built on top of a standard singly linked list. Indeed, the structure of a singly linked list is highly appropriate for the stack's last-in-first-out behaviour, as it ensures that a user will only ever access the element at the "top" of the stack – in other words, the element at the front of the list – making push and pop operations in a stack constant time operations[22]. As with the relationship between doubly linked lists and queues, it's also entirely possible to implement a stack using an array– but this type of stack implementation is inherently hamstrung by the immutable length of an array, which requires potential resizing, awkward left-hand shifting of elements when the pop operation is called, and right-hand shifting of elements

when the push operation is called.

If you're still confused about how a stack would actually behave for arbitrary input, consider the following: if you add 1, then 2, then 3 to a stack, removal will happen in the ordering 3, 2, 1. Each element is "stacked" on top of the previous element, and since you can only remove elements from the top of a stack, you're left with LIFO behaviour.

Code Sample

```
  public class Stack{
    Node top;

    //push inserts an item at the top of the stack
    public void push(int data){
      Node n = new Node();
      n.data = data;
      n.next = top;
      //new node replaces the old top node
      top = n;
      return;
    }

    //pop deletes the item at the top of the stack
    public int pop(){
      if(top == null)
        return -1;
      //save the top's data, replace top with next element
      int data = top.data;
      top = top.next;
      return data;
    }

    //returns the value at the top of the stack without deleting
    public int peek(){
      if(top != null)
        return top.data;
      else return -1;
    }
}
```

Methods of Interest

- Push (O(C)): adds a piece of data to the top of the stack. In the above code sample, this method works exactly the same as our linked list insert method.
- Pop (O(C)): removes a data from the top of the stack, returns that element
- Peek (O(C)): returns the data at the top of the stack without removing it from the stack

Uses of The Stack Perhaps the most common usage of stacks exists in the form of arithmetic and boolean statement evaluation. Calculators, both logical and arithmetic, are a common application of this ability. While infix notation (e.g. "2+2") is easy to understand for a human, it's significantly more complicated for a machine, since such a task requires the preservation of precedence and association. Postfix notation (e.g. "22+") is significantly easier for machines to understand, and can actually be evaluated in a highly efficient and relatively simple manner using a stack. Indeed, this efficiency is exactly why TI-84 graphing calculators have scarcely evolved in the past 20 years, as extra computational beef isn't really necessary when it comes to evaluating basic arithmetic.

Stacks are used in the popular *Shunting-Yard Algorithm*, which uses a combination of a stack and a queue to convert infix mathematical notation into postfix mathematical notation[4]. Indeed, this algorithm actually represents the easiest way for a computer to evaluate infix notation – conversion into postfix!

Suggested Readings The following articles are suggested to readers interested in learning more about stacks:

Rohit Rastogi, Pinki Mondal, and Kritika Agarwal. An exhaustive review for infix to postfix conversion with applications and benefits. In *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, pages 95–100. IEEE, 2015

Richard E Ladner, Richard J Lipton, and Larry J Stockmeyer. Alternating pushdown and stack automata. *SIAM Journal on Computing*, 13(1):135–155, 1984

4.3 Queues

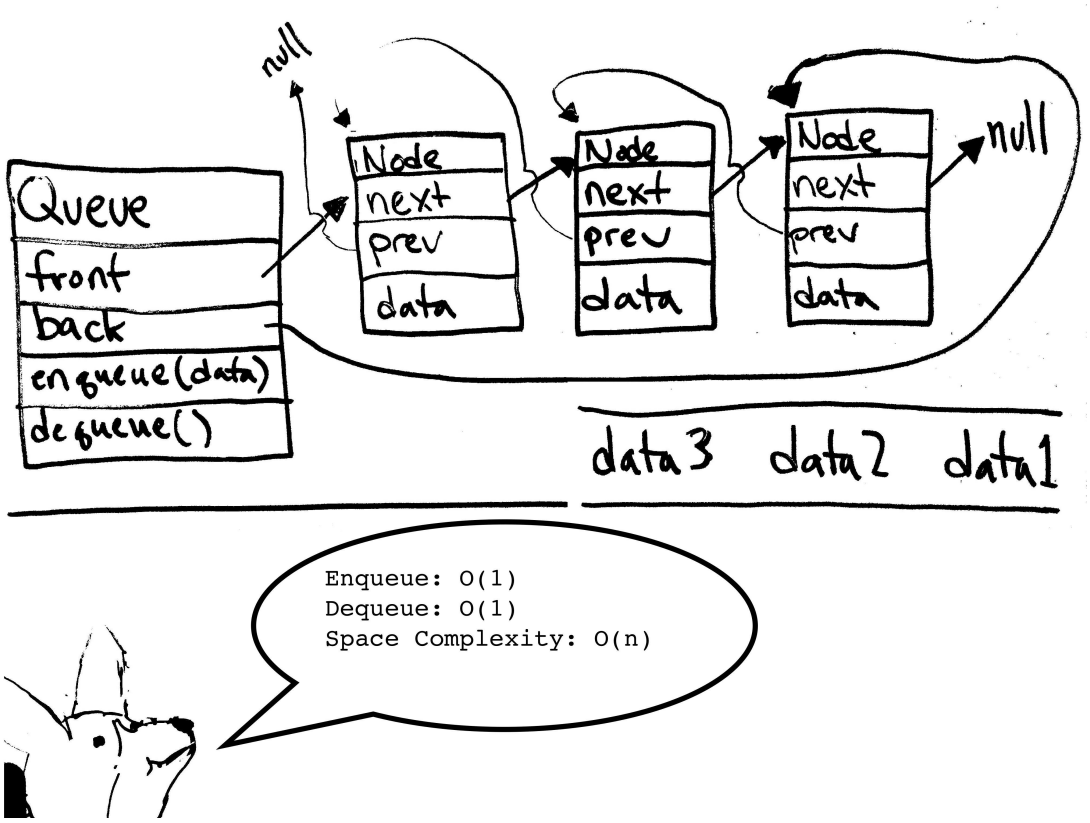

Figure 9: To the top left: a diagram of a queue in memory. To the bottom right: a common, more concise representation of the same queue.

Important Terms

- FIFO (First In, First Out): As you might have noticed, this is precisely the reverse of LIFO behaviour. Elements are removed from a queue in precisely the *same* order that they're inserted. Since data1 was the first element inserted into the stack via enqueue in Fig. 9, it will also be the first element removed in the event of a dequeue.

How The Queue Works Similarly to stacks, queues are typically implemented as a slight modification of linked lists. Queues, however, lend themselves much more readily to a doubly-linked list implementation, since elements need to be accessed on both ends. With a doubly-linked list, both accesses can be completed easily and in constant time. Singly-linked lists, on the other hand, will invariably require some linear time operations to implement deletion at the end of the list[22].

Code Sample

```
public class Queue{
```

```
//this queue implementation has two nodes with null value
//at all times, to simplify edge cases (otherwise we'd
//have to account for null nodes all the time
Node front;
Node back;

//creates a new queue with an empty front and back node
public Queue(){
  front = new Node();
  back = new Node();
  front.next = back;
  back.prev = front;
}

//inserts a new item into the front of the queue
//NOTE: actually inserts AFTER front node (which has null values)
public void enqueue(int data){
  //create a new node, insert data, point prev at front of queue
  Node n = new Node();
  n.data = data;
  n.prev = front;
  //next value becomes front value's current next value
  n.next = front.next;
  //the current node after the head needs to point at this node
  n.next.prev = n;
  //front of queue now points to this node
  front.next = n;
}

//deletes (and returns) an item from the end of the queue
//NOTE: actually inserts BEFORE back node (which has null values)
public int dequeue(){
  //if the queue is empty, return error value
  if(back.prev.prev == null)
    return -1;
  //grab data from the last non-null node in the queue
  int data = back.prev.data;
  //'cut' that node out of the queue
  back.prev = back.prev.prev;
```

```
      back.prev.next = back;
      //return data we just grabbed
      return data;
    }
}
```

Methods of Interest

- Enqueue: add a piece of data to the front of the queue (the head of the list)
- Dequeue: remove and return a piece of data from the back of the queue (the tail of the list)

Uses of The Queue Queues are one of the most common data structures, making it difficult to pin down just the "most popular" use for queues. One of the areas where queues present the most advantages is concurrency and messaging- whether you're dispatching jobs to 50 threads performing gene folding, processing transactions for a national bank, or dispatching movement updates to 16 players in a Call of Duty online multiplayer match, queues are a magnificent equalizer, ensuring that each element of data spends a (roughly) equivalent amount of time waiting for processing.

Suggested Readings The following articles are suggested to readers interested in learning more about queues:

John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977

4.4 Trees

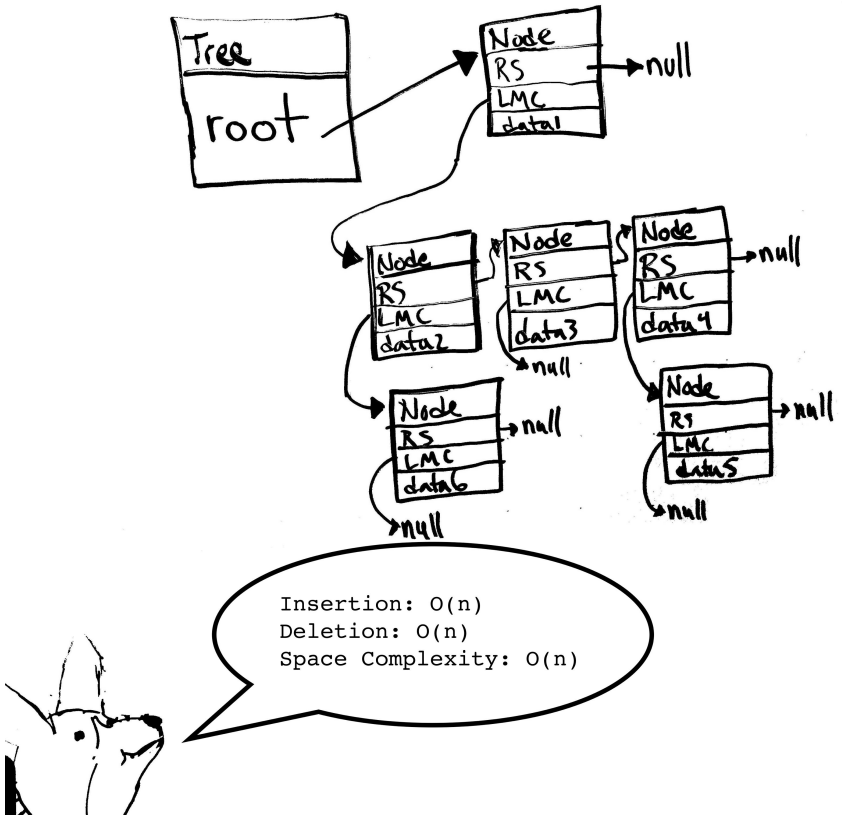

Figure 10: An artistic, hand-crafted example of a right-sibling, leftmost-child tree.

Important Terms

- Root: the first and "top" node in the tree.
- Leaves: All nodes with no children in a given tree. data1 is the data contained in the root of the tree in Fig. 10.
- Height of Node: The number of edges on the longest path between a given node and a leaf[27].
- Height of Tree: The height of the root[22].
- Children: Nodes accessible from a given parent node in a tree that differ in depth by 1. In a binary tree, these are commonly simply "left" and "right" children. In other tree schemes, there can be arbitrary numbers of children[27].
- Parent: The node through which another node is accessible in the tree; that is, the last element of the path from the root to any given node[27].
- Depth-first Search: Often abbreviated to DFS, depth-first search represents one of the most popular tree search algorithms in use today. In any depth-first search,

a specific order is chosen for child node exploration. Often, this order is simply left to right, but it's perfectly within the bounds of a depth-first search to explore child nodes in a right to left order, or any other order of preference. In such a search, the *entire subtree* with its root at the first explored child is explored before the second child. Then the entire subtree rooted at the second explored child is explored before the third child node is explored[22]. This property repeats within each subtree. This search is particularly applicable in state search, where a heuristic (that is, an *inexact rule of thumb*) is commonly used to determine which child node should be explored[27].

- Breadth-first Search: Often abbreviated to BFS because most computer scientists struggle to pronounce the word "breadth," BFS works in a way practically quite different from that of DFS – indeed, in a breadth-first search, each child node of a given node is *guaranteed* to have been explored before any of the given node's grandchild (child of child) nodes are explored[22]. However, this search also means that you have to "remember" lots of nodes at any point during the search, since a lot of time is spent waiting for visitation sibling nodes before a node's children can be explored. This is typically implemented using a stack. This type of search is most useful when no heuristic seems applicable for a problem. Interestingly enough, this search behaves in a manner quite similar to that of topological orderings in graphs, which we'll explore later (5.3). Often, this exploration can be written quite simply using a queue[27].

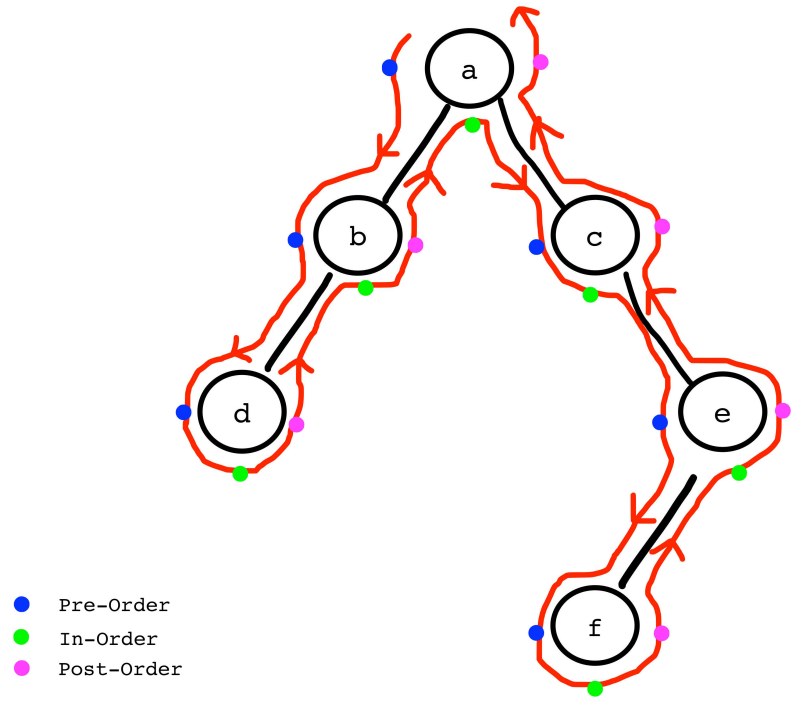

Figure 11: An illustration of various traversals of a tree. The line indicates the order in which a depth-first search visits the nodes of the tree, and the dots indicate the points in the three different traversals that the contents of a node would actually be printed.

- Pre-Order Traversal and Printing: Pre-order tree traversal is simple: nodes are printed (or compared) in exactly the order that they're visited in a depth-first search recursive method. In other words: a node is visited and printed. The first child is visited and printed. If that node has no children, the second child of the original node is visited and printed; otherwise, the first child of the first child of the original node is visited and printed (Fig. 11).

- In-Order Traversal and Printing: In-order tree traversal is slightly less intuitive, and makes little sense for trees in which nodes can have arbitrary numbers of children. However, the general idea is this: a node is visited. Its left child (or left children) is visited and printed. Once all of the left children of the original node have been printed, the original node is printed. Only then are the right children visited and printed (Fig. 11).

- Post-Order Traversal and Printing: Post-order tree traversal is essentially the opposite of pre-order traversal: instead of printing a given node before any of its children have been printed, a given node's value is only printed once each and every child of that node has already been printed (Fig. 11).

How The Tree Works Trees are everywhere in computer science, and have a wide variety of specialized forms in literature. Luckily for you, this section only details the most generic idea of n-ary trees. First, it's important to establish the different methods of representing a tree, which can vary significantly for trees without set limitations on numbers of children. Unlike binary (2-child) trees, which have the luxury of maintaining just two child pointers (left and right), n-ary trees must account for arbitrary numbers of children.

One common approach to this problem: simply maintain a linked list of children in every single parent node, allowing any number of children. This has the advantage of simple traversal, but the disadvantage of significant space overhead in addition to significant traversal running time overhead. Such an approach isn't perfect, though, since any traversal of such a tree must traverse both the tree and several linked lists, significantly complicating the code. Another popular approach is the LMC-RS (LeftMost Child-Right Sibling) representation (Fig. 10), in which each node need only contain three fields: a piece of data, a pointer to the node's right sibling, and a pointer to the node's leftmost child. In this way, trees of immense complexity with highly varying numbers of children may be constructed and traversed with ease.

Code Sample

```
public class Tree{
    Node root;

    //inserts a sibling immediately to the right of a tree node
    public void insertSibling(int data, Node n){
      Node temp = new Node();
      temp.data = data;
      //the new node now points to the old right sibling of the original node
      temp.rs = n.rs;
      n.rs = temp;
    }

    //inserts a child beneath a node
    public void insertChild(int data, Node n){
      Node temp = new Node();
      temp.data = data;
      //the new child now points to the old child node
      //through its right sibling field
      temp.rs = n.lmc;
      n.lmc = temp;
```

```
    return;
  }

  //traverses the tree using lookupHelper
  public boolean lookup(int data){
    if(root == null)
      return false;
    else return lookupHelper(data, root);
  }

  //if the data of the current node matches data, return true.
  //if the current node is null, we've hit a dead end- return false.
  //otherwise, explore the children and siblings of the current node
  public boolean lookupHelper(int data, Node n){
    if(n == null)
      return false;
    if(n.data == data)
      return true;
    //uses OR for return because data is present if we return true anywhere
    return lookupHelper(data, n.lmc) || lookupHelper(data, n.rs);
  }

  //delete method cut for the sake of brevity
}
```

Methods of Interest Unspecialized trees contain few, if any, methods of interest. As you can see in the above example, lookup, and insert work in much the same way as they do in any linked list.

Uses of The Tree A clever use of trees with arbitrary numbers of children is tries. Tries, unlike many other specialized forms of trees, never hold a full value at any node- instead the value of a leaf node is the entire path from the root of the tree to that node. Tries can be used to, for instance, create a tree for every word in the English language, where the root node of the tree has 26 children (a through z) and every path from that root node to a leaf is a word in the English language. Such constructs can be immensely useful when, for instance, one needs to traverse every word in a language without saving each element discretely.

Suggested Readings The following articles are suggested to readers interested in learning more about trees:

Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983

Donald E Knuth. Dynamic huffman coding. *Journal of algorithms*, 6(2):163–180, 1985

4.5 Binary Search Trees

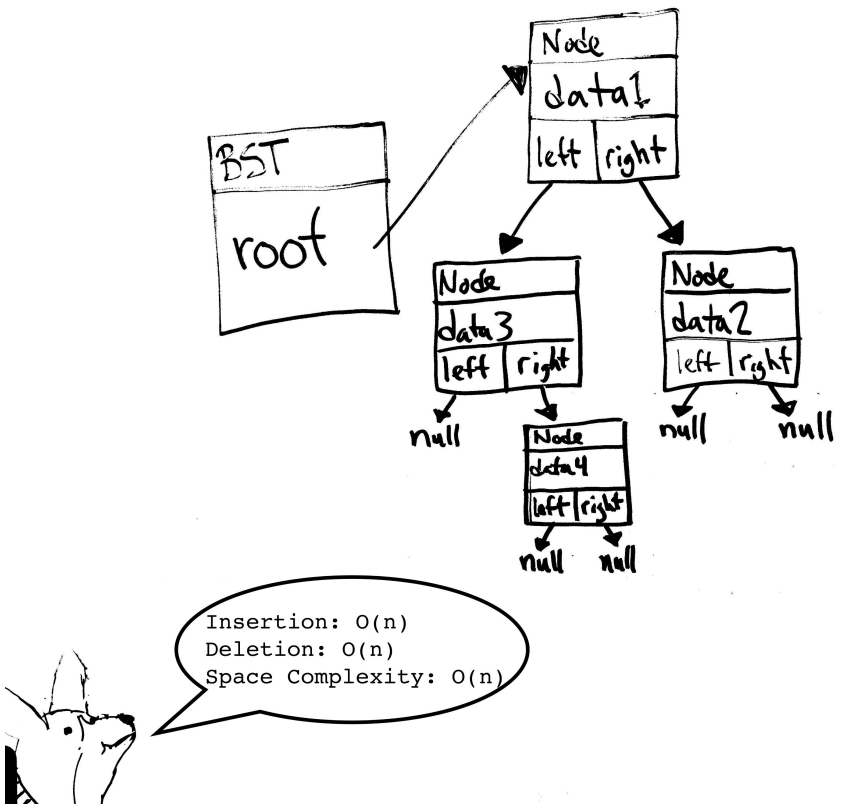

Figure 12: An artistic, hand-crafted example of a BST. In this hierarchy, `data3 < data4 < data1 < data2`.

Important Terms

- Left Child: in a BST, the left child contains a data value of lesser value than its parent. In Fig. 12, `data3 < data1` because `data3` is `data1`'s left child.
- Right Child: similarly, the right child contains a data value of higher value than its parent. In Fig. 12, `data4 < data3` because `data4` is `data3`'s right child.

How The Binary Search Tree Works Binary Search Trees are actually an elegant extension of the traditional tree structure. A BST is binary (only two children allowed,

a left and a right) and obeys a single organizational rule: any given left child's data is less than its parent's data, which in turn is less than that parent's right child's data[22]. This ordering makes BSTs extremely useful for search algorithms, as it makes it much easier to "narrow down" the location of a particular element.

Code Sample

```
public class BST{
  Node root;

  //starts at the root, so insert calls are cleaner
  public void insert(int data){
    Node n = new Node();
      n.data = data;
    //if the root is currently null, our job is done-
    //just save our new node as the new root.
    if(root == null){
      root = n;
      return;
    }
    insertHelper(n, root);
  }

  //walk the tree, using comparisons to decide on left/right move
  public void insertHelper(Node temp, Node n){
    //if our data is larger than the current node's data,
    //we belong to the right
    if(temp.data > n.data){
      //call recursively on right child, unless
      //the right child doesn't exist- in that case,
      //insert our node as a new right child.
      if(n.right != null){
        insertHelper(temp, n.right);
      } else{
        n.right = temp;
      }
    } else {
      //call recursively on left child, unless
      //it doesn't exist, in which case we insert.
      if(n.left != null){
```

```
      insertHelper(temp, n.left);
    } else{
      n.left = temp;
    }
  }
}

//starts our lookup at the root, handling null root case, matching root case,
//and starting lookuphelper otherwise based on comparison to root
public boolean lookup(int data){
  if(root == null)
    return false;
  if(root.data == data)
    return true;
  //search to the left is data is less than root's data
  if(root.data > data)
    return lookupHelper(data, root.left);
  //otherwise search to the right
  return lookupHelper(data, root.right);
}

//walk the tree, comparing our data to nodes to move left or right.
//if we hit a null node, our data doesn't exist in the tree- return false.
//if we hit a node with matching data, return true!
public boolean lookupHelper(int data, Node n){
  if(data == n.data)
    return true;
  if(data > n.data){
    //search to right
    if(n.right == null){
      return false;
    } else{
      return lookupHelper(data, n.right);
    }
  } else {
    //search to left
    if(n.left == null){
      return false;
    } else{
```

```
        return lookupHelper(data, n.right);
      }
    }
  }

  public void delete(int data){
    //delete has been omitted for the sake of brevity
  }
}
```

Methods of Interest

- insert (O(n)): The binary search tree's insertion method begins at the root and traverses down the tree, comparing the item to be inserted with each parent node and traveling down the appropriate path until it finds a null path. The item is then inserted in a new node in that location[22].
- lookup (O(n)): Works similarly to the insertion method, except returns true if the item is already present and false otherwise[22].
- delete (O(n)): Works similarly to lookup and insert, except it deletes the matching node if present. When the matching node is deleted, its right child takes its place, and the left subtree of its right child is transplanted to the far right of its left subtree[22].

Uses of The Binary Search Tree Binary search trees have fewer uses than one might think due to one major flaw: balancing. If this isn't immediately clear, consider the following: insert the values 1, 2, 3, 4, 5 into a binary search tree. The resulting tree is extremely lopsided, and has no left-children branches at all. In fact, this resulting tree actually behaves exactly like a linked list with an extra null pointer at each node. It's this weakness of binary search trees that results in a worst-case linear running time, causing most implementations to prefer self-balancing trees- including red-black trees and AVL trees (5.1), which we'll explore in the Advanced Data Structures section.

Suggested Readings The following articles are suggested to readers interested in learning more about binary search trees:

Thomas N Hibbard. Some combinatorial properties of certain trees with applications to searching and sorting. *Journal of the ACM (JACM)*, 9(1):13–28, 1962

Edward H Sussenguth Jr. Use of tree structures for processing files. *Communications of the ACM*, 6(5):272–279, 1963

5 Advanced Data Structures

"An algorithm must be seen to be believed."
-Donald Knuth[13]

5.1 AVL Trees

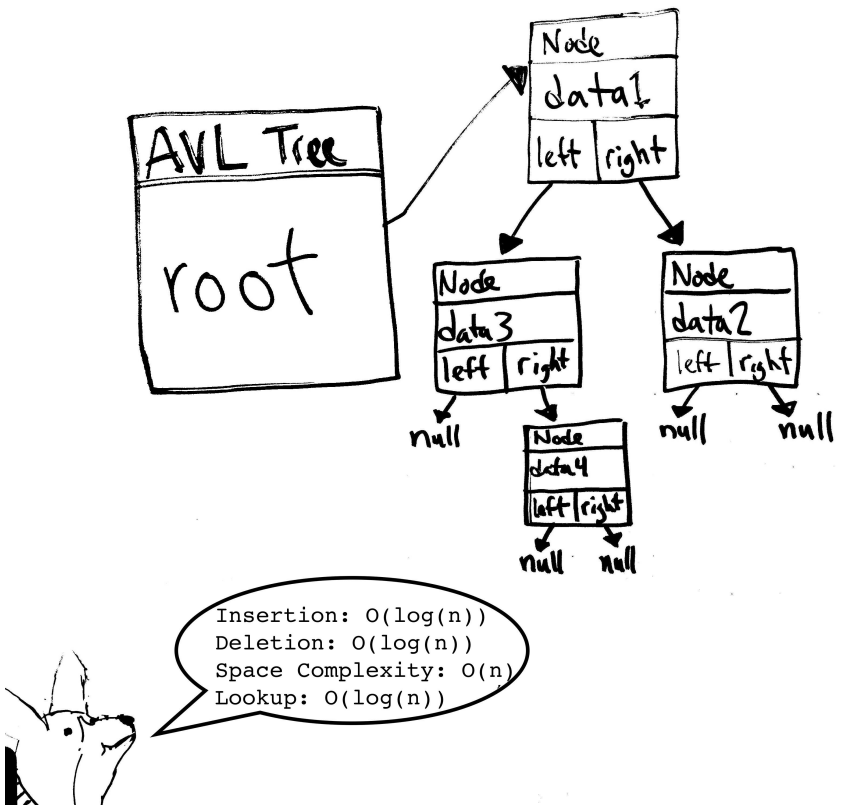

Figure 13: An artistic, hand-crafted example of an AVL tree. Note the similarity to the BST we explored at the end of section 4.

Important Terms

- Self-balancing property: AVL trees are called "self balancing" because they enforce a very particular rule: no two children of a given node may differ in height by more than one[22]. As you can see from Fig. 13, AVL Trees look exactly the same as a balanced BST – the two differ only in their balance strategies (namely, AVL trees follow a balancing rule, while BSTs do not).

- Rotation: When a value is inserted into an AVL tree, the insert method traverses the tree, comparing the value to be inserted with each subsequent node and picking the right or left path based on the outcome of the comparator operation. Once the

new node has been inserted, the insert method then travels *back* up the tree from the point of insertion. At each node on the path back to the root, a comparison of the left and right child heights is made; if any given node happens to fail this comparison (that is, the height of the left node differs by more than one from the height of the right node), the tree "rotates" at this position to fix the imbalance. Similarly, a deletion must also traverse the path back to the root from the point of deletion to account for potential new imbalances[13].

How The AVL Tree Works The heart of the AVL tree lies in the rotation method- without this strategy to resolve imbalances, the AVL tree simply could not function. This rotation method must account for the following imbalance cases, each of which is named after the location of the misbehaving subtree (of excess height). Two of these cases are simply slightly more elaborate cases of the other two, and require a single rotation to reduce to a simpler case.

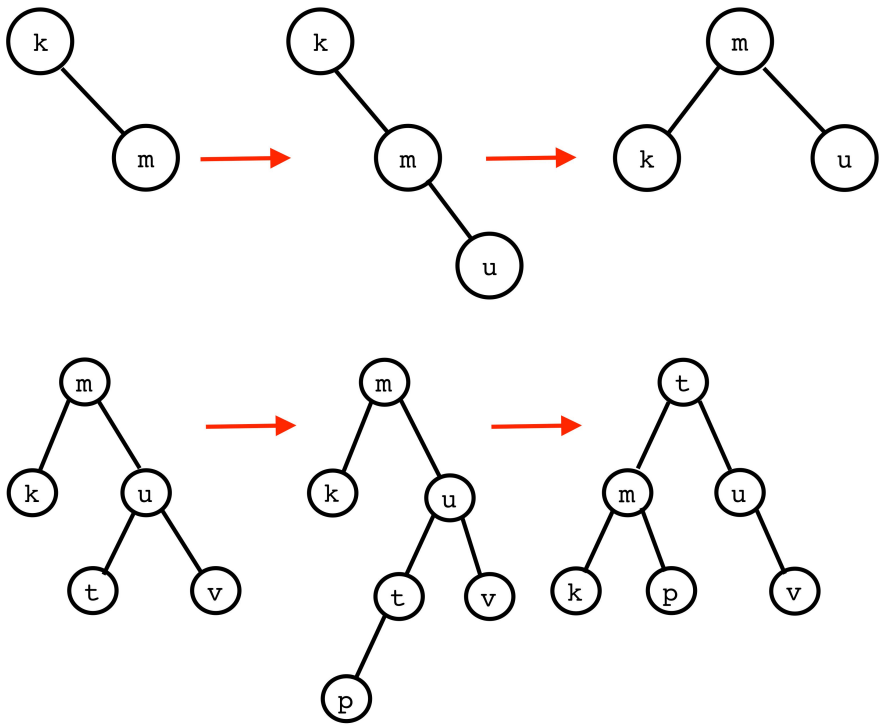

Figure 14: A single LL rotation (top) and a double LR rotation (bottom).

1. RL (Right Left): The left subtree of the right child of the current node is too high. To resolve this case, we rotate the tree into RR form. To do this, we "rotate" without touching our current node; instead, the left child of the right child becomes our new right child, with our original right child as its new right child and its original child as our original right child's new left child[13].

2. RR (Right Right): The right subtree of the right child of the current node is too high. To resolve this case, simply "rotate" the right child so that the current node is its new left child. It should retain its original right child, and its original left child should become the new right child of the current node[13].

3. LR (Left Right): The right subtree of the left child of the current node is too high. To resolve this case, we simply rotate into LL form in the exact opposite way that we rotate RL form into RR form[13]. You can see an example of this rotation at the bottom of Fig. 14.

4. LL (Left Left): The left subtree of the left child of the current node is too high. To fix this case, we simply rotate the left child of the current node so that our current node becomes its right child. Just like in the RR case, our current node inherits a new left child- the former left child's right child[13]. You can see an example of this rotation at the top of Fig. 14.

Methods of Interest Typically, AVL trees only use the standard tree methods- insert, delete, and lookup. However, the insert and delete methods include a twist – after completing an insertion or deletion (which involves a traversal from the root to a location in the tree), these methods have to return back up the tree, checking for newly-introduced imbalances. To do this, most AVL tree implementations include a `balance(Node n)` method for these functions to call as they recurse back up to the root[13]. The balance method may then call a `rotate(Node n)` method to balance a node with an identified imbalance in accordance with the rules discussed above.

Uses of AVL Trees AVL trees are commonly used for sorting, data storage, and a variety of other tasks.

Suggested Readings The following articles are suggested to readers interested in learning more about binary search AVL Trees:

Caxton C Foster. A generalization of avl trees. *Communications of the ACM*, 16(8):513–517, 1973

A fine implementation of an AVL Tree can be found at the following address, courtesy of Florida International University:
`http://users.cis.fiu.edu/~weiss/dsaajava/code/DataStructures/AvlTree.java`

5.2 Binary Heaps

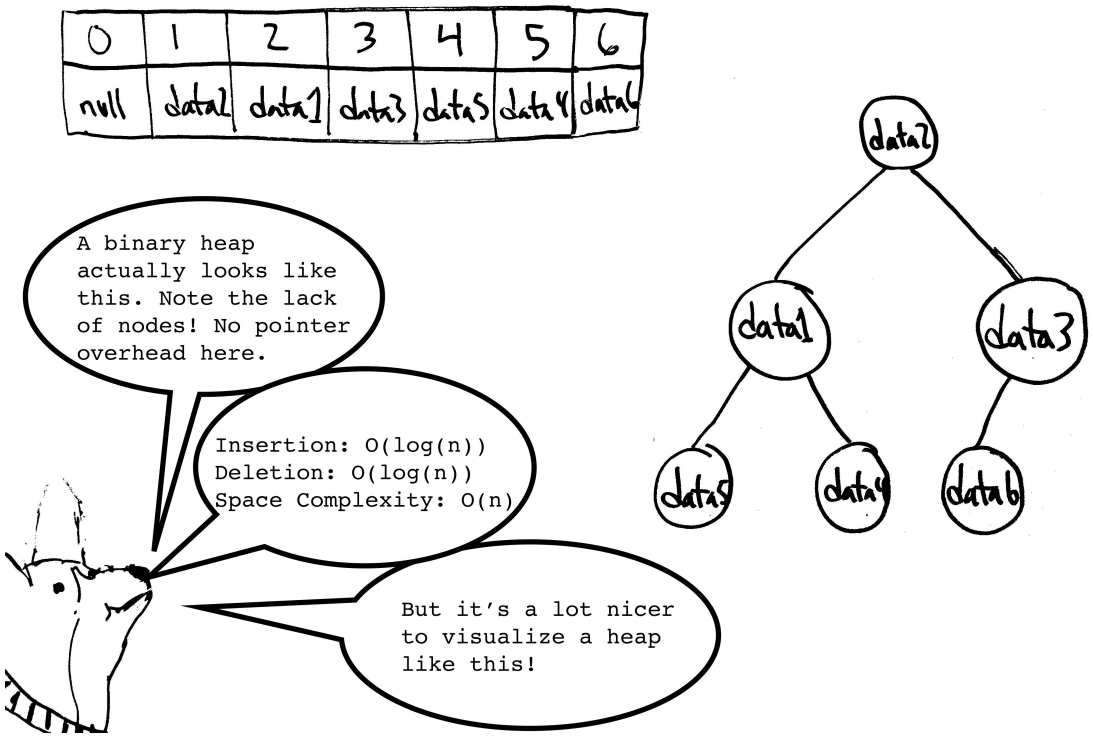

Figure 15: A beautiful representation of a binary heap.

Important Terms

- Heap property: There are two main variants of binary heaps- min heaps and max heaps. Both types of heap follow a special heap property: each parent node must be either less than or equal to its children (for min heaps), or greater than or equal to its children (for max heaps)[13]. Note that there is no explicit relationship defined between children through this rule, unlike BSTs.

- Min heap: All parent nodes are less than or equal to their children. In a min heap, for example, `data2 < data1` and `data2 <data3` in Fig. 15.

- Max heap: All parent nodes are greater than or equal to their children. In a max heap, parents must be strictly greater than their children, so `data2 > data1` and `data2 > data3`.

How The Binary Heap Works Though heaps are typically visualized as trees, they are typically actually implemented using an array. As such, they usually can't be navigated using the left and right child pointers that you'd normally expect a tree structure to have—instead, navigation of a heap is typically done using indices. (see "Methods of

Interest") Traditionally, the root of the heap occurs at index 1 in the array rather than 0[13]. (If this doesn't make sense, consider the basic case (traditionally starting at index 1 for ease of computation) of a three-element heap. The root occurs at index 1 (aka "i"), the left child occurs at index 2 (aka "2i"), and the right child occurs at index 3 (aka "2i+1").

Methods of Interest

- leftchild: To access the left child of the node with index i in the array, simply access element 2i [13].
- rightchild: To access the right child, access element 2i+1 [13].
- parent: This same property can be used to traverse back up the array using parents- indeed, to access the parent of a given node with index i, one need only access element floor(i/2), that is: i divided by 2, rounded down to the nearest whole number[13].
- insert: unlike a traditional tree, insertion can only occur in a single position in any heap- the next open space in the array used to represent the heap. As a result, values must "bubble up" after insertion into a correct position, as it's quite likely they won't be placed in the right part of the heap initially.
- remove: Typically, only the root node is ever removed from a heap. As a result, there is a very particular procedure in place to ensure that a removal doesn't break the heap property. When a value is removed from the heap, the root is replaced with the final element from the bottom level of the heap. Then the "bubble down" procedure begins, wherein an element (starting at the root) is compared to its children and swapped with whichever one is greater (in a max heap) or smaller (in the case of a min heap) until its position no longer a violation of the heap property.

Uses of The Binary Heap Min heaps are often used as cheap, ready-made priority queues. A combination of a max heap and a min heap may be used to keep track of a running median of a set of elements. Heaps have also found popularity through the heapsort sorting algorithm, which, as you might expect, involves dumping a set of elements into a heap and then removing them to produce a sorted set.

The heap occupies a very interesting position in the computer science world, since it is a low-overhead (since it uses no pointers) data structure with significant benefits- most notably the ability to sort input with minimal effort due to the bubble-up/bubble-down process. Special variants of heaps, including the Fibonacci Heap and the Pairing Heap, are often used to prioritize elements of input in a variety of different algorithms.

Suggested Readings The following articles are suggested to readers interested in learning more about binary heaps:

Michael Barbehenn. A note on the complexity of dijkstra's algorithm for graphs with weighted vertices. *IEEE transactions on computers*, (2):263, 1998

A fine implementation of a binary heap can be found at the following address, courtesy of Washington University:
`http://courses.cs.washington.edu/courses/cse373/11wi/homework/5/BinaryHeap.java`

5.3 Graphs

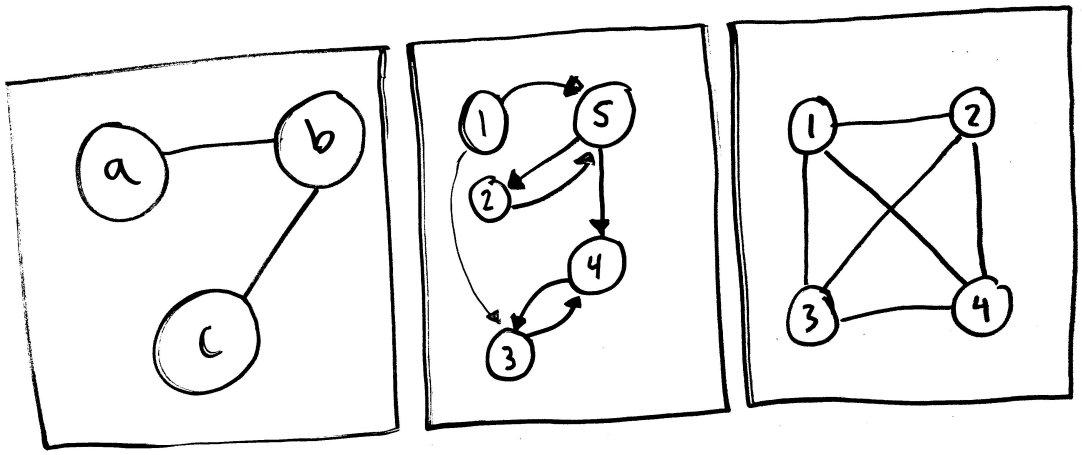

Figure 16: Three possible graphs. The first is undirected and sparse, the second is directed, and the third is undirected and dense. All three are unweighted.

Important Terms

- Edge: two nodes may be connected by an edge. For example, in Fig. 16, nodes `a` and `b` are connected by an edge.
- Sparseness: graphs can either be sparse (with relatively few direct connections between nodes, as in the graph on the left in Fig. 16) or dense (most nodes are connected to all other nodes directly, e.g. the center graph in Fig. 16).
- Adjacency list: in sparse graphs, adjacency lists are used to indicate the existence of edges between points. Typically an adjacency list is full of pointers (or indices in an array used to represent the graph) used to access other nodes in the matrix[22].
- Adjacency matrix: Dense graphs commonly make use of adjacency matrices to indicate edges, made with 2D arrays, due to their quick access times (constant,

compared to linear for an adjacency list). However, adjacency matrices are typically inappropriate in the context of sparse graphs due to their quadratic space complexity[22].

- Undirected graph: a graph in which edges (X,Y) may be traversed both ways ((X,Y) or (Y,X)).
- Directed graph: a graph in which edges may only be traversed in a single direction. That is; the edge (A, B) from A to B means that there is a way to get from A to B, but does not mean that there is a direct path from B back to A, which would be called (B, A).
- Shortest path: the path with the minimum edge cost (or number of edges) between two specific nodes. Dijkstra's algorithm can find such a path in quadratic time.
- Minimum Spanning tree: an acyclic subset of edges (that is, a group of edges where there aren't any loops) in a graph through which any node may be reached. Prim's algorithm and Kruskal's algorithm represent two popular minimum spanning tree algorithms (though Kruskals' may also produce a minimum spanning forest, if used on a graph containing islands)[22].
- Minimum Spanning forest: Like a spanning tree, except it isn't necessarily connected. In graphs that contain islands, spanning trees are impossible since it's impossible to connect a component to another component if there's no edge between those components in the first place. As a result, spanning forests are often used as a form of compromise, since they satisfy the spanning component in as satisfactory a way as possible in such cases.
- Weighted edge: an edge with a cost associated with traversal (such as distance)
- Topological ordering: create a subgraph through the traversal of edges to non-visited nodes beginning at a particular node.

How The Graph Works A graph is a collection of nodes and a collection of edges. However, don't be fooled into thinking that a graph is some kind of geometric object; instead, it's better to think of a graph as a combinatorial object- a collection of points and a subset of possible connections between those points.

Methods of Interest Typically, general graphs don't require any particular methods. More interesting are the algorithms used to analyse graphs, including Dijkstra's Algorithm, Prim's Algorithm, and Kruskal's Algorithm, to name a few[22].

Uses of Graphs In recent years, Google Maps has become the most evident example of a graph in daily life. However, graphs have found a wide variety of uses in daily life: bus networks for public schools and cities, shipping routes for companies like UPS and the USPS, utility routing for power, gas, and water, and even garbage collection in Java can all heavily exploit graphs for optimization.

While the usage of graphs to find shortest paths and minimal spanning trees is certainly evident to even an individual inexperienced individual, graphs can also be useful for more abstract problems. Maximal matching, k-coloring, and flow problems all represent more advanced applications of graphs.[smith]

Suggested Readings The following articles are suggested to readers interested in learning more about graphs:

Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*, volume 57. Elsevier, 2004

A fine implementation of a graph can be found at the following address, courtesy of Princeton University:
`http://algs4.cs.princeton.edu/41graph/Graph.java.html`

5.4 Hash Tables

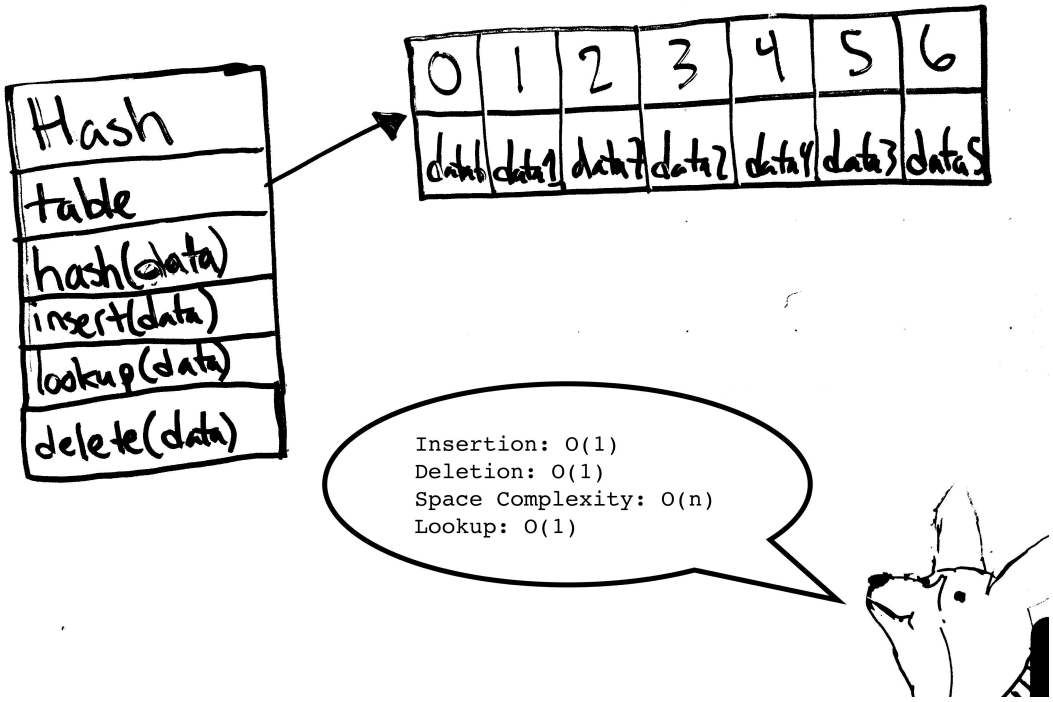

Figure 17: An artistic, hand-crafted example of the structure.

Important Terms

- Load factor: The load factor of a hash table is simply the portion of the hash table that is currently occupied with non-null elements[26].
- Modulo: `n mod x` is the same as the integer remainder when dividing n by x.
- Hash Function: maps a value to an index in the array that a hash table uses for actual value storage[26].
- Perfect Hash: a hash table in which every element of the value space maps to a unique element of the index space, and every index maps to a unique value[26]. `n` by `x`.

How The Hash Table Works Hash tables, also known as "hash maps," are mappings of values to indices using a hash function. Hash functions are so immensely complicated that we won't be able to go into much detail about them for this guide, but we can focus on the general gist: a hash function takes any value that your data can possibly assume, and outputs an index in the hash table's back-end array[26]. For this reason, it's quite

common to use `modulo sizeOfTable` in your hash function, so you can only ever get input producing a valid index in your table. It's also important that your hash function always behaves the same for given input (that is, it must be deterministic)- so a hash function can't contain, for instance, randomly generated numbers[26].

The simplest "hash table" is really just a normal array – for instance, if you want to store integer values between 0 and 50, you can easily make a hash table that uses the exact value of the integer to be stored as its index in the table. If you wanted to store integer values between 50 and 100, you could just use an array of length 50 with a hash function of input – 50. Both of these examples also produce what we call a "perfect hash"- that is, a hash table in which each element of the input maps to a discrete, unique index, and each index has a unique value that maps only to that index. However, most hash table implementations aren't quite so ideal[26]. Hash tables are commonly used to store all kinds of data, but as an example, we'll consider the storage of strings.

A common, easily implemented hash function for strings simply sums the integer values of each character, and takes the `modulo sizeOfTable` of the number produced. This hash table certainly won't be a "perfect hash," though- consider the word "a", which maps to 97 using this hash function. All lowercase letters have a higher integer value than a, and since there's no word "b" in the English language, the 98th index in the hash table will never contain a value. And that doesn't even account for the 97 positions in the array before the 97th index. Since we seldom see perfect hashes in the real world, our hash tables are often at least half full of empty elements[26].

As a result, hash tables are often designed with a specific load factor in mind – typically no more than 50% fill [22]. Keeping the load factor below 50% is extremely important depending on your hash table's strategy for collision resolution. Collision, the term we use to describe two values that hash to the same index, is usually resolved through one of two common methods, though other collision resolution strategies do exist. The first of these two strategies is known as separate chaining, and essentially makes every element in the hash table into a linked list. When a collision occurs, the offending element is simply added to the linked list at its index. While this solves the problem of non-unique hash values, it does make the running time of the lookup, delete, and insert methods worst-case linear, which essentially defeats the purpose of using a hash table[22].

Luckily, separate chaining isn't the only collision resolution strategy. A different strategy, called "double hashing" is quite popular, and essentially involves computing a second hash function in the event of a conflict. This second hash value is then added to the original hash value to create an entirely new index, which is then used to store the value. If the new index also contains a value, the second hash value is simply added again, and again, and again, until an empty slot in the array is found. In its most basic implementation, doubly hashing can be done without a second hash function at all- instead, conflicts merely use 1 as their second hash value in all cases. Unfortunately, this tends to create

clusters of filled indices, which can inhibit lookup, delete, and insert running times[22].

If you're still having difficulty picturing how a hash table works, consider the United States population. It sounds like it would be awfully difficult to map a unique array index to each US citizen, but in fact, a simple perfect hash function already exists – in the form of social security numbers.

Methods of Interest

- hash (O(C)): Calculates an index in the hash table given a value.

Uses of Hash Tables Hash tables are easily one of the most useful data structures in computer science today. Any task with large quantities of data and a need for constant access time can benefit from hash tables. While this task likely sounds quite nebulous right now, that's becomes it's such a pervasive task in society – social networks, insurance companies, game developers, and even the US government all use hash tables for an immense variety of tasks.

Suggested Readings The following articles are suggested to readers interested in learning more about hash tables:

Ward Douglas Maurer and Theodore Gyle Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975

A fine implementation of a hash table can be found at the following address, courtesy of Princeton University:
`http://algs4.cs.princeton.edu/34hash/LinearProbingHashST.java.html`

6 Summary

The study of data structures is a study quite unlike any other. While many fields of study are constrained by the natural world, or the laws of physics, data structures are limited only by the bounds of human creativity. Any task that can benefit from data organization and increased efficiency can benefit from the implementation of a data structure.

Because data structures are so useful in so many different contexts, they have become an essential tool for any and all computer scientists. However, understanding data structures fully also requires a proper understanding of concepts like memory allocation, space and time complexity, average running time, and worst-case running time. Hopefully this guide has helped you understand these concepts (slightly) more thoroughly.

References

[1] Michael Barbehenn. A note on the complexity of dijkstra's algorithm for graphs with weighted vertices. *IEEE transactions on computers*, (2):263, 1998.

[2] Edgar Chávez, José L Marroquín, and Gonzalo Navarro. Fixed queries array: A fast and economical data structure for proximity searching. *Multimedia Tools and Applications*, 14(2):113–135, 2001.

[3] Paul F Dietz. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 122–127. ACM, 1982.

[4] EW Dijkstra. Algol 60 translation: An algol 60 translator for the x1 and making a translator for algol 60, report mr 35/61. *Mathematisch Centrum, Amsterdam*, 1961.

[5] Mary E d'Imperio. Data structures and their representation in storage. *Annual Review in Automatic Programming*, 5:1–75, 1969.

[6] Caxton C Foster. A generalization of avl trees. *Communications of the ACM*, 16(8):513–517, 1973.

[7] Martin Charles Golumbic. *Algorithmic graph theory and perfect graphs*, volume 57. Elsevier, 2004.

[8] John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977.

[9] Thomas N Hibbard. Some combinatorial properties of certain trees with applications to searching and sorting. *Journal of the ACM (JACM)*, 9(1):13–28, 1962.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 2 edition, 4 1988.

[11] K.L. Kluge. Method and apparatus for managing a linked-list data structure, July 13 1999. US Patent 5,924,098.

[12] Donald E Knuth. Dynamic huffman coding. *Journal of algorithms*, 6(2):163–180, 1985.

[13] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[14] Richard E Ladner, Richard J Lipton, and Larry J Stockmeyer. Alternating pushdown and stack automata. *SIAM Journal on Computing*, 13(1):135–155, 1984.

[15] Stuart E Madnick. String processing techniques. *Communications of the ACM*, 10(7):420–424, 1967.

[16] Ward Douglas Maurer and Theodore Gyle Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1):5–19, 1975.

[17] Yen-Ching Pao. *Engineering analysis: interactive methods and programs with FORTRAN, QuickBASIC, MATLAB, and Mathematica*. CRC Press, 1998.

[18] Rohit Rastogi, Pinki Mondal, and Kritika Agarwal. An exhaustive review for infix to postfix conversion with applications and benefits. In *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference on*, pages 95–100. IEEE, 2015.

[19] Alastair Reid. Designing data structures. In *Proceedings of the 1989 Glasgow Workshop on Functional Programming*. Springer-Verlag.

[20] Randolph G Scarborough and Harwood G Kolsky. A vectorizing fortran compiler. *IBM Journal of Research and Development*, 30(2):163–171, 1986.

[21] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.

[22] Harry F. Smith. *Data Structures: Form and Function*. Saunders College Publishing, 1 edition, 6 1995.

[23] Edward H Sussenguth Jr. Use of tree structures for processing files. *Communications of the ACM*, 6(5):272–279, 1963.

[24] Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, 1984.

[25] Stefan Van Der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[26] Tom van Dijk. Analysing and improving hash table performance. In *10th Twente Student Conference on IT. University of Twente, Faculty of Electrical Engineering and Computer Science*, 2009.

[27] Mark Allen Weiss. *Data structures and algorithm analysis in Java*. Addison-Wesley Longman Publishing Co., Inc., 1998.