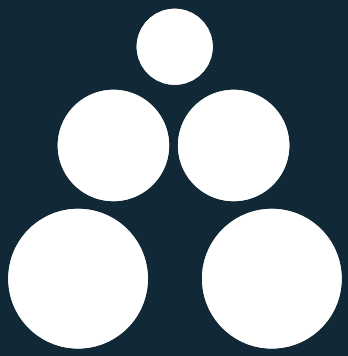
Data Structures and Algorithms

DSA



Annotated Reference with Examples

Granville Barnett

Luca Del Tongo

Data Structures and Algorithms: Annotated Reference with Examples

First Edition

Contents

Preface

Every book has a story as to how it came about and this one is no different, although we would be lying if we said its development had not been somewhat impromptu. Put simply this book is the result of a series of emails sent back and forth between the two authors during the development of a library for the .NET framework of the same name (with the omission of the subtitle of course!). The conversation started off something like, "Why don't we create a more aesthetically pleasing way to present our pseudocode?" After a few weeks this new presentation style had in fact grown into pseudocode listings with chunks of text describing how the data structure or algorithm in question works and various other things about it. At this point we thought, "What the heck, let's make this thing into a book!" And so, in the summer of 2008 we began work on this book side by side with the actual library implementation.

When we started writing this book the only things that we were sure about with respect to how the book should be structured were:

1. always make explanations as simple as possible while maintaining a moderately fine degree of precision to keep the more eager minded reader happy; and

2. inject diagrams to demystify problems that are even moderatly challenging to visualise (... and so we could remember how our own algorithms worked when looking back at them!); and finally

3. present concise and self-explanatory pseudocode listings that can be ported easily to most mainstream imperative programming languages like C++, C#, and Java.

A key factor of this book and its associated implementations is that all algorithms (unless otherwise stated) were designed by us, using the theory of the algorithm in question as a guideline (for which we are eternally grateful to their original creators). Therefore they may sometimes turn out to be worse than the "normal" implementations—and sometimes not. We are two fellows of the opinion that choice is a great thing. Read our book, read several others on the same subject and use what you see fit from each (if anything) when implementing your own version of the algorithms in question.

Through this book we hope that you will see the absolute necessity of understanding which data structure or algorithm to use for a certain scenario. In all projects, especially those that are concerned with performance (here we apply an even greater emphasis on real-time systems) the selection of the wrong data structure or algorithm can be the cause of a great deal of performance pain.

Therefore it is absolutely key that you think about the run time complexity and space requirements of your selected approach. In this book we only explain the theoretical implications to consider, but this is for a good reason: compilers are very different in how they work. One C++ compiler may have some amazing optimisation phases specifically targeted at recursion, another may not, for example. Of course this is just an example but you would be surprised by how many subtle differences there are between compilers. These differences which may make a fast algorithm slow, and vice versa. We could also factor in the same concerns about languages that target virtual machines, leaving all the actual various implementation issues to you given that you will know your language's compiler much better than us...well in most cases. This has resulted in a more concise book that focuses on what we think are the key issues.

One final note: never take the words of others as gospel; verify all that can be feasibly verified and make up your own mind.

We hope you enjoy reading this book as much as we have enjoyed writing it.

Granville Barnett
Luca Del Tongo

Acknowledgements

Writing this short book has been a fun and rewarding experience. We would like to thank, in no particular order the following people who have helped us during the writing of this book.

Sonu Kapoor generously hosted our book which when we released the first draft received over thirteen thousand downloads, without his generosity this book would not have been able to reach so many people. Jon Skeet provided us with an alarming number of suggestions throughout for which we are eternally grateful. Jon also edited this book as well.

We would also like to thank those who provided the odd suggestion via email to us. All feedback was listened to and you will no doubt see some content influenced by your suggestions.

A special thank you also goes out to those who helped publicise this book from Microsoft's Channel 9 weekly show (thanks Dan!) to the many bloggers who helped spread the word. You gave us an audience and for that we are extremely grateful.

Thank you to all who contributed in some way to this book. The programming community never ceases to amaze us in how willing its constituents are to give time to projects such as this one. Thank you.

About the Authors

Granville Barnett

Granville is currently a Ph.D candidate at Queensland University of Technology (QUT) working on parallelism at the Microsoft QUT eResearch Centre[1]. He also holds a degree in Computer Science, and is a Microsoft MVP. His main interests are in programming languages and compilers. Granville can be contacted via one of two places: either his personal website (`http://gbarnett.org`) or his blog (`http://msmvps.com/blogs/gbarnett`).

Luca Del Tongo

Luca is currently studying for his masters degree in Computer Science at Florence. His main interests vary from web development to research fields such as data mining and computer vision. Luca also maintains an Italian blog which can be found at `http://blogs.ugidotnet.org/wetblog/`.

[1] `http://www.mquter.qut.edu.au/`

Page intentionally left blank.

Chapter 1

Introduction

1.1 What this book is, and what it isn't

This book provides implementations of common and uncommon algorithms in pseudocode which is language independent and provides for easy porting to most imperative programming languages. It is not a definitive book on the theory of data structures and algorithms.

For the most part this book presents implementations devised by the authors themselves based on the concepts by which the respective algorithms are based upon so it is more than possible that our implementations differ from those considered the norm.

You should use this book alongside another on the same subject, but one that contains formal proofs of the algorithms in question. In this book we use the abstract big Oh notation to depict the run time complexity of algorithms so that the book appeals to a larger audience.

1.2 Assumed knowledge

We have written this book with few assumptions of the reader, but some have been necessary in order to keep the book as concise and approachable as possible. We assume that the reader is familiar with the following:

1. Big Oh notation
2. An imperative programming language
3. Object oriented concepts

1.2.1 Big Oh notation

For run time complexity analysis we use big Oh notation extensively so it is vital that you are familiar with the general concepts to determine which is the best algorithm for you in certain scenarios. We have chosen to use big Oh notation for a few reasons, the most important of which is that it provides an abstract measurement by which we can judge the performance of algorithms without using mathematical proofs.

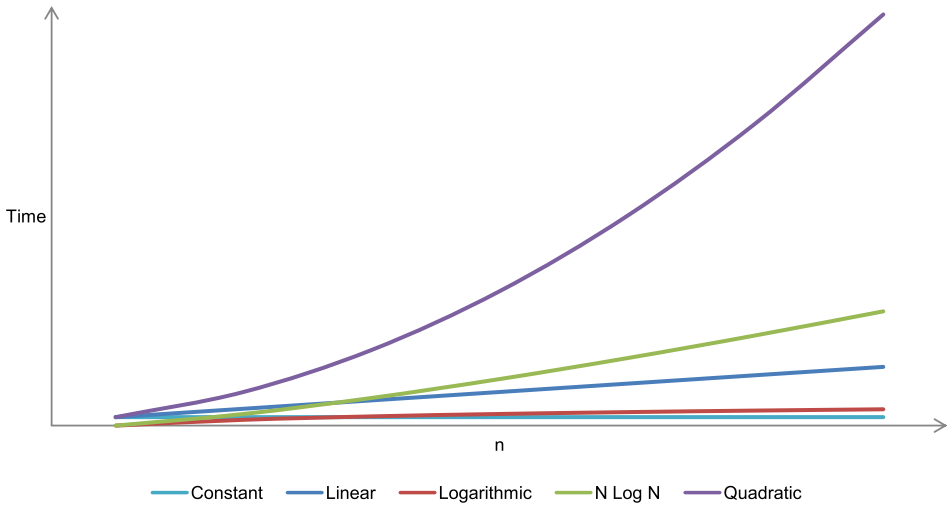


Figure 1.1: Algorithmic run time expansion

Figure 1.1 shows some of the run times to demonstrate how important it is to choose an efficient algorithm. For the sanity of our graph we have omitted cubic $O(n^3)$, and exponential $O(2^n)$ run times. Cubic and exponential algorithms should only ever be used for very small problems (if ever!); avoid them if feasibly possible.

The following list explains some of the most common big Oh notations:

$O(1)$ constant: the operation doesn't depend on the size of its input, e.g. adding a node to the tail of a linked list where we always maintain a pointer to the tail node.

$O(n)$ linear: the run time complexity is proportionate to the size of n.

$O(log\ n)$ logarithmic: normally associated with algorithms that break the problem into smaller chunks per each invocation, e.g. searching a binary search tree.

$O(n\ log\ n)$ just $n\ log\ n$: usually associated with an algorithm that breaks the problem into smaller chunks per each invocation, and then takes the results of these smaller chunks and stitches them back together, e.g. quick sort.

$O(n^2)$ quadratic: e.g. bubble sort.

$O(n^3)$ cubic: very rare.

$O(2^n)$ exponential: incredibly rare.

If you encounter either of the latter two items (cubic and exponential) this is really a signal for you to review the design of your algorithm. While prototyping algorithm designs you may just have the intention of solving the problem irrespective of how fast it works. We would strongly advise that you always review your algorithm design and optimise where possible—particularly loops

and recursive calls—so that you can get the most efficient run times for your algorithms.

The biggest asset that big Oh notation gives us is that it allows us to essentially discard things like hardware. If you have two sorting algorithms, one with a quadratic run time, and the other with a logarithmic run time then the logarithmic algorithm will always be faster than the quadratic one when the data set becomes suitably large. This applies even if the former is ran on a machine that is far faster than the latter. Why? Because big Oh notation isolates a key factor in algorithm analysis: growth. An algorithm with a quadratic run time grows faster than one with a logarithmic run time. It is generally said at some point as $n \to \infty$ the logarithmic algorithm will become faster than the quadratic algorithm.

Big Oh notation also acts as a communication tool. Picture the scene: you are having a meeting with some fellow developers within your product group. You are discussing prototype algorithms for node discovery in massive networks. Several minutes elapse after you and two others have discussed your respective algorithms and how they work. Does this give you a good idea of how fast each respective algorithm is? No. The result of such a discussion will tell you more about the high level algorithm design rather than its efficiency. Replay the scene back in your head, but this time as well as talking about algorithm design each respective developer states the asymptotic run time of their algorithm. Using the latter approach you not only get a good general idea about the algorithm design, but also key efficiency data which allows you to make better choices when it comes to selecting an algorithm fit for purpose.

Some readers may actually work in a product group where they are given budgets per feature. Each feature holds with it a budget that represents its uppermost time bound. If you save some time in one feature it doesn't necessarily give you a buffer for the remaining features. Imagine you are working on an application, and you are in the team that is developing the routines that will essentially spin up everything that is required when the application is started. Everything is great until your boss comes in and tells you that the start up time should not exceed n ms. The efficiency of every algorithm that is invoked during start up in this example is absolutely key to a successful product. Even if you don't have these budgets you should still strive for optimal solutions.

Taking a quantitative approach for many software development properties will make you a far superior programmer - measuring one's work is critical to success.

1.2.2 Imperative programming language

All examples are given in a pseudo-imperative coding format and so the reader must know the basics of some imperative mainstream programming language to port the examples effectively, we have written this book with the following target languages in mind:

1. C++
2. C#
3. Java

The reason that we are explicit in this requirement is simple—all our implementations are based on an imperative thinking style. If you are a functional programmer you will need to apply various aspects from the functional paradigm to produce efficient solutions with respect to your functional language whether it be Haskell, F#, OCaml, etc.

Two of the languages that we have listed (C# and Java) target virtual machines which provide various things like security sand boxing, and memory management via garbage collection algorithms. It is trivial to port our implementations to these languages. When porting to C++ you must remember to use pointers for certain things. For example, when we describe a linked list node as having a reference to the next node, this description is in the context of a managed environment. In C++ you should interpret the reference as a pointer to the next node and so on. For programmers who have a fair amount of experience with their respective language these subtleties will present no issue, which is why we really do emphasise that the reader must be comfortable with at least one imperative language in order to successfully port the pseudo-implementations in this book.

It is essential that the user is familiar with primitive imperative language constructs before reading this book otherwise you will just get lost. Some algorithms presented in this book can be confusing to follow even for experienced programmers!

1.2.3 Object oriented concepts

For the most part this book does not use features that are specific to any one language. In particular, we never provide data structures or algorithms that work on generic types—this is in order to make the samples as easy to follow as possible. However, to appreciate the designs of our data structures you will need to be familiar with the following object oriented (OO) concepts:

1. Inheritance
2. Encapsulation
3. Polymorphism

This is especially important if you are planning on looking at the C# target that we have implemented (more on that in §1.7) which makes extensive use of the OO concepts listed above. As a final note it is also desirable that the reader is familiar with interfaces as the C# target uses interfaces throughout the sorting algorithms.

1.3 Pseudocode

Throughout this book we use pseudocode to describe our solutions. For the most part interpreting the pseudocode is trivial as it looks very much like a more abstract C++, or C#, but there are a few things to point out:

1. Pre-conditions should always be enforced
2. Post-conditions represent the result of applying algorithm a to data structure d

3. The type of parameters is inferred
4. All primitive language constructs are explicitly begun and ended

If an algorithm has a return type it will often be presented in the post-condition, but where the return type is sufficiently obvious it may be omitted for the sake of brevity.

Most algorithms in this book require parameters, and because we assign no explicit type to those parameters the type is inferred from the contexts in which it is used, and the operations performed upon it. Additionally, the name of the parameter usually acts as the biggest clue to its type. For instance n is a pseudo-name for a number and so you can assume unless otherwise stated that n translates to an integer that has the same number of bits as a WORD on a 32 bit machine, similarly l is a pseudo-name for a list where a list is a resizeable array (e.g. a vector).

The last major point of reference is that we always explicitly end a language construct. For instance if we wish to close the scope of a **for** loop we will explicitly state **end for** rather than leaving the interpretation of when scopes are closed to the reader. While implicit scope closure works well in simple code, in complex cases it can lead to ambiguity.

The pseudocode style that we use within this book is rather straightforward. All algorithms start with a simple algorithm signature, e.g.

1) **algorithm** AlgorithmName($arg1$, $arg2$, ..., $argN$)
2) ...
n) **end** AlgorithmName

Immediately after the algorithm signature we list any **Pre** or **Post** conditions.

1) **algorithm** AlgorithmName(n)
2) **Pre:** n is the value to compute the factorial of
3) $n \geq 0$
4) **Post:** the factorial of n has been computed
5) // ...
n) **end** AlgorithmName

The example above describes an algorithm by the name of *AlgorithmName*, which takes a single numeric parameter n. The pre and post conditions follow the algorithm signature; you should always enforce the pre-conditions of an algorithm when porting them to your language of choice.

Normally what is listed as a pre-conidition is critical to the algorithms operation. This may cover things like the actual parameter not being null, or that the collection passed in must contain at least n items. The post-condition mainly describes the effect of the algorithms operation. An example of a post-condition might be "The list has been sorted in ascending order"

Because everything we describe is language independent you will need to make your own mind up on how to best handle pre-conditions. For example, in the C# target we have implemented, we consider non-conformance to pre-conditions to be exceptional cases. We provide a message in the exception to tell the caller why the algorithm has failed to execute normally.

1.4 Tips for working through the examples

As with most books you get out what you put in and so we recommend that in order to get the most out of this book you work through each algorithm with a pen and paper to track things like variable names, recursive calls etc.

The best way to work through algorithms is to set up a table, and in that table give each variable its own column and continuously update these columns. This will help you keep track of and visualise the mutations that are occurring throughout the algorithm. Often while working through algorithms in such a way you can intuitively map relationships between data structures rather than trying to work out a few values on paper and the rest in your head. We suggest you put everything on paper irrespective of how trivial some variables and calculations may be so that you always have a point of reference.

When dealing with recursive algorithm traces we recommend you do the same as the above, but also have a table that records function calls and who they return to. This approach is a far cleaner way than drawing out an elaborate map of function calls with arrows to one another, which gets large quickly and simply makes things more complex to follow. Track everything in a simple and systematic way to make your time studying the implementations far easier.

1.5 Book outline

We have split this book into two parts:

Part 1: Provides discussion and pseudo-implementations of common and uncommon data structures; and

Part 2: Provides algorithms of varying purposes from sorting to string operations.

The reader doesn't have to read the book sequentially from beginning to end: chapters can be read independently from one another. We suggest that in part 1 you read each chapter in its entirety, but in part 2 you can get away with just reading the section of a chapter that describes the algorithm you are interested in.

Each of the chapters on data structures present initially the algorithms concerned with:

1. Insertion
2. Deletion
3. Searching

The previous list represents what we believe in the vast majority of cases to be the most important for each respective data structure.

For all readers we recommend that before looking at any algorithm you quickly look at Appendix E which contains a table listing the various symbols used within our algorithms and their meaning. One keyword that we would like to point out here is **yield**. You can think of **yield** in the same light as **return**. The **return** keyword causes the method to exit and returns control to the caller, whereas **yield** returns each value to the caller. With **yield** control only returns to the caller when all values to return to the caller have been exhausted.

1.6 Testing

All the data structures and algorithms have been tested using a minimised test driven development style on paper to flesh out the pseudocode algorithm. We then transcribe these tests into unit tests satisfying them one by one. When all the test cases have been progressively satisfied we consider that algorithm suitably tested.

For the most part algorithms have fairly obvious cases which need to be satisfied. Some however have many areas which can prove to be more complex to satisfy. With such algorithms we will point out the test cases which are tricky and the corresponding portions of pseudocode within the algorithm that satisfy that respective case.

As you become more familiar with the actual problem you will be able to intuitively identify areas which may cause problems for your algorithms implementation. This in some cases will yield an overwhelming list of concerns which will hinder your ability to design an algorithm greatly. When you are bombarded with such a vast amount of concerns look at the overall problem again and sub-divide the problem into smaller problems. Solving the smaller problems and then composing them is a far easier task than clouding your mind with too many little details.

The only type of testing that we use in the implementation of all that is provided in this book are unit tests. Because unit tests contribute such a core piece of creating somewhat more stable software we invite the reader to view Appendix D which describes testing in more depth.

1.7 Where can I get the code?

This book doesn't provide any code specifically aligned with it, however we do actively maintain an open source project[1] that houses a C# implementation of all the pseudocode listed. The project is named *Data Structures and Algorithms* (DSA) and can be found at `http://codeplex.com/dsa`.

1.8 Final messages

We have just a few final messages to the reader that we hope you digest before you embark on reading this book:

1. Understand how the algorithm works first in an abstract sense; and
2. Always work through the algorithms on paper to understand how they achieve their outcome

If you always follow these key points, you will get the most out of this book.

[1] All readers are encouraged to provide suggestions, feature requests, and bugs so we can further improve our implementations.

Part I

Data Structures

Chapter 2

Linked Lists

Linked lists can be thought of from a high level perspective as being a series of nodes. Each node has at least a single pointer to the next node, and in the last node's case a null pointer representing that there are no more nodes in the linked list.

In DSA our implementations of linked lists always maintain head and tail pointers so that insertion at either the head or tail of the list is a constant time operation. Random insertion is excluded from this and will be a linear operation. As such, linked lists in DSA have the following characteristics:

1. Insertion is $O(1)$
2. Deletion is $O(n)$
3. Searching is $O(n)$

Out of the three operations the one that stands out is that of insertion. In DSA we chose to always maintain pointers (or more aptly references) to the node(s) at the head and tail of the linked list and so performing a traditional insertion to either the front or back of the linked list is an $O(1)$ operation. An exception to this rule is performing an insertion before a node that is neither the head nor tail in a singly linked list. When the node we are inserting before is somewhere in the middle of the linked list (known as random insertion) the complexity is $O(n)$. In order to add before the designated node we need to traverse the linked list to find that node's current predecessor. This traversal yields an $O(n)$ run time.

This data structure is trivial, but linked lists have a few key points which at times make them very attractive:

1. the list is dynamically resized, thus it incurs no copy penalty like an array or vector would eventually incur; and
2. insertion is $O(1)$.

2.1 Singly Linked List

Singly linked lists are one of the most primitive data structures you will find in this book. Each node that makes up a singly linked list consists of a value, and a reference to the next node (if any) in the list.

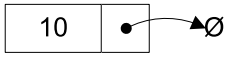


Figure 2.1: Singly linked list node

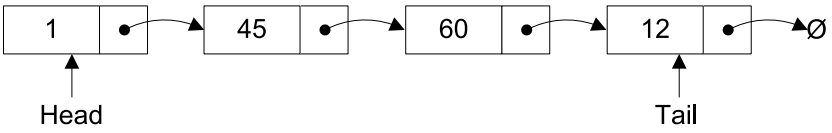


Figure 2.2: A singly linked list populated with integers

2.1.1 Insertion

In general when people talk about insertion with respect to linked lists of any form they implicitly refer to the adding of a node to the tail of the list. When you use an API like that of DSA and you see a general purpose method that adds a node to the list, you can assume that you are adding the node to the tail of the list not the head.

Adding a node to a singly linked list has only two cases:

1. $head = \emptyset$ in which case the node we are adding is now both the *head* and *tail* of the list; or

2. we simply need to append our node onto the end of the list updating the *tail* reference appropriately.

```
1) algorithm Add(value)
2)     Pre: value is the value to add to the list
3)     Post: value has been placed at the tail of the list
4)     n ← node(value)
5)     if head = ∅
6)         head ← n
7)         tail ← n
8)     else
9)         tail.Next ← n
10)        tail ← n
11)    end if
12) end Add
```

As an example of the previous algorithm consider adding the following sequence of integers to the list: 1, 45, 60, and 12, the resulting list is that of Figure 2.2.

2.1.2 Searching

Searching a linked list is straightforward: we simply traverse the list checking the value we are looking for with the value of each node in the linked list. The algorithm listed in this section is very similar to that used for traversal in §2.1.4.

```
1) algorithm Contains(head, value)
2)     Pre:  head is the head node in the list
3)           value is the value to search for
4)     Post: the item is either in the linked list, true; otherwise false
5)     n ← head
6)     while n ≠ ∅ and n.Value ≠ value
7)         n ← n.Next
8)     end while
9)     if n = ∅
10)        return false
11)    end if
12)    return true
13) end Contains
```

2.1.3 Deletion

Deleting a node from a linked list is straightforward but there are a few cases we need to account for:

1. the list is empty; or
2. the node to remove is the only node in the linked list; or
3. we are removing the head node; or
4. we are removing the tail node; or
5. the node to remove is somewhere in between the head and tail; or
6. the item to remove doesn't exist in the linked list

The algorithm whose cases we have described will remove a node from anywhere within a list irrespective of whether the node is the *head* etc. If you know that items will only ever be removed from the *head* or *tail* of the list then you can create much more concise algorithms. In the case of always removing from the front of the linked list deletion becomes an $O(1)$ operation.

```
1) algorithm Remove(head, value)
2)     Pre: head is the head node in the list
3)          value is the value to remove from the list
4)     Post: value is removed from the list, true; otherwise false
5)     if head = ∅
6)         // case 1
7)         return false
8)     end if
9)     n ← head
10)    if n.Value = value
11)        if head = tail
12)            // case 2
13)            head ← ∅
14)            tail ← ∅
15)        else
16)            // case 3
17)            head ← head.Next
18)        end if
19)        return true
20)    end if
21)    while n.Next ≠ ∅ and n.Next.Value ≠ value
22)        n ← n.Next
23)    end while
24)    if n.Next ≠ ∅
25)        if n.Next = tail
26)            // case 4
27)            tail ← n
28)        end if
29)        // this is only case 5 if the conditional on line 25 was false
30)        n.Next ← n.Next.Next
31)        return true
32)    end if
33)    // case 6
34)    return false
35) end Remove
```

2.1.4 Traversing the list

Traversing a singly linked list is the same as that of traversing a doubly linked list (defined in §2.2). You start at the head of the list and continue until you come across a node that is ∅. The two cases are as follows:

1. $node = \emptyset$, we have exhausted all nodes in the linked list; or
2. we must update the *node* reference to be *node*.Next.

The algorithm described is a very simple one that makes use of a simple *while* loop to check the first case.

```
1) algorithm Traverse(head)
2)     Pre: head is the head node in the list
3)     Post: the items in the list have been traversed
4)     n ← head
5)     while n ≠ 0
6)         yield n.Value
7)         n ← n.Next
8)     end while
9) end Traverse
```

2.1.5 Traversing the list in reverse order

Traversing a singly linked list in a forward manner (i.e. left to right) is simple as demonstrated in §2.1.4. However, what if we wanted to traverse the nodes in the linked list in reverse order for some reason? The algorithm to perform such a traversal is very simple, and just like demonstrated in §2.1.3 we will need to acquire a reference to the predecessor of a node, even though the fundamental characteristics of the nodes that make up a singly linked list make this an expensive operation. For each node, finding its predecessor is an $O(n)$ operation, so over the course of traversing the whole list backwards the cost becomes $O(n^2)$.

Figure 2.3 depicts the following algorithm being applied to a linked list with the integers 5, 10, 1, and 40.

```
1) algorithm ReverseTraversal(head, tail)
2)     Pre: head and tail belong to the same list
3)     Post: the items in the list have been traversed in reverse order
4)     if tail ≠ ∅
5)         curr ← tail
6)         while curr ≠ head
7)             prev ← head
8)             while prev.Next ≠ curr
9)                 prev ← prev.Next
10)            end while
11)            yield curr.Value
12)            curr ← prev
13)        end while
14)        yield curr.Value
15)    end if
16) end ReverseTraversal
```

This algorithm is only of real interest when we are using singly linked lists, as you will soon see that doubly linked lists (defined in §2.2) make reverse list traversal simple and efficient, as shown in §2.2.3.

2.2 Doubly Linked List

Doubly linked lists are very similar to singly linked lists. The only difference is that each node has a reference to both the next and previous nodes in the list.

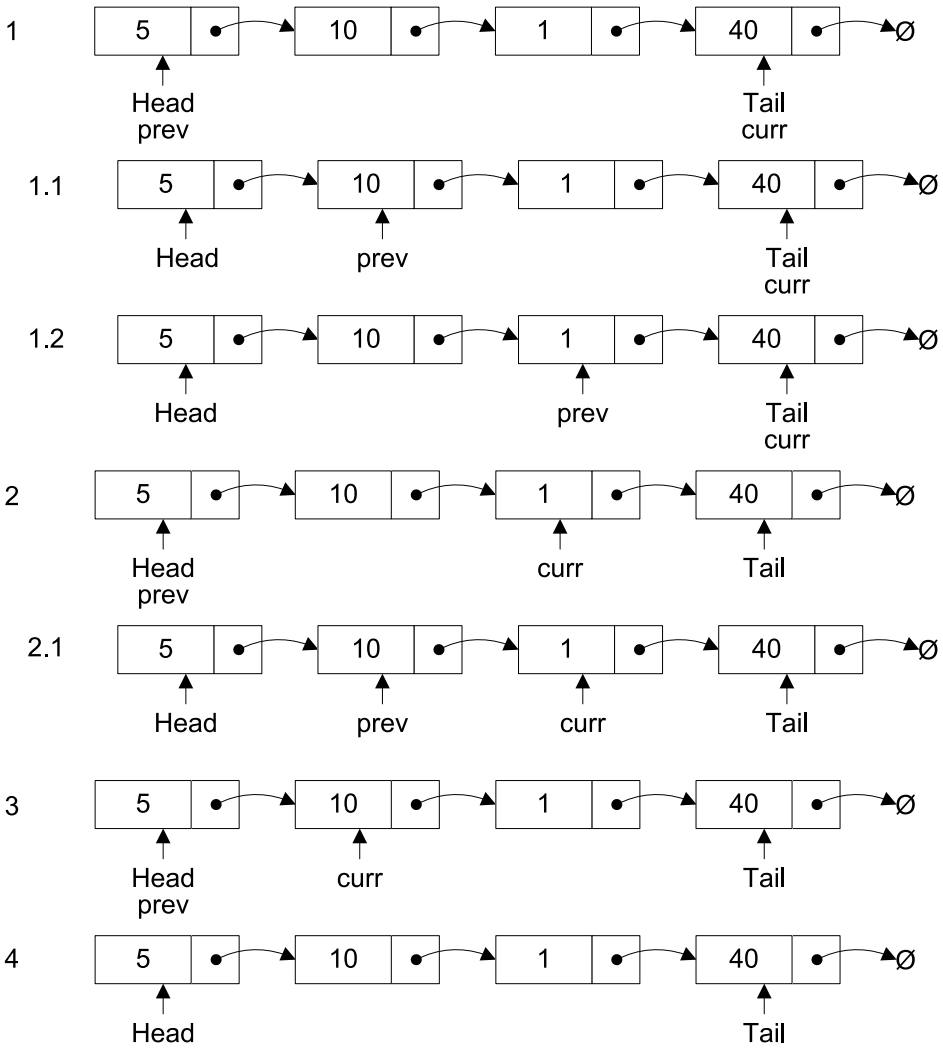


Figure 2.3: Reverse traveral of a singly linked list

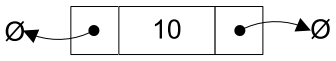


Figure 2.4: Doubly linked list node

The following algorithms for the doubly linked list are exactly the same as those listed previously for the singly linked list:

1. Searching (defined in §2.1.2)
2. Traversal (defined in §2.1.4)

2.2.1 Insertion

The only major difference between the algorithm in §2.1.1 is that we need to remember to bind the previous pointer of n to the previous tail node if n was not the first node to be inserted into the list.

```
1) algorithm Add(value)
2)     Pre: value is the value to add to the list
3)     Post: value has been placed at the tail of the list
4)     n ← node(value)
5)     if head = ∅
6)         head ← n
7)         tail ← n
8)     else
9)         n.Previous ← tail
10)        tail.Next ← n
11)        tail ← n
12)    end if
13) end Add
```

Figure 2.5 shows the doubly linked list after adding the sequence of integers defined in §2.1.1.

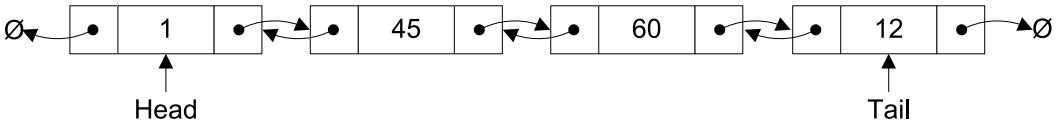


Figure 2.5: Doubly linked list populated with integers

2.2.2 Deletion

As you may of guessed the cases that we use for deletion in a doubly linked list are exactly the same as those defined in §2.1.3. Like insertion we have the added task of binding an additional reference (*Previous*) to the correct value.

```
1) algorithm Remove(head, value)
2)     Pre: head is the head node in the list
3)          value is the value to remove from the list
4)     Post: value is removed from the list, true; otherwise false
5)     if head = ∅
6)         return false
7)     end if
8)     if value = head.Value
9)         if head = tail
10)            head ← ∅
11)            tail ← ∅
12)        else
13)            head ← head.Next
14)            head.Previous ← ∅
15)        end if
16)        return true
17)    end if
18)    n ← head.Next
19)    while n ≠ ∅ and value ≠ n.Value
20)        n ← n.Next
21)    end while
22)    if n = tail
23)        tail ← tail.Previous
24)        tail.Next ← ∅
25)        return true
26)    else if n ≠ ∅
27)        n.Previous.Next ← n.Next
28)        n.Next.Previous ← n.Previous
29)        return true
30)    end if
31)    return false
32) end Remove
```

2.2.3 Reverse Traversal

Singly linked lists have a forward only design, which is why the reverse traversal algorithm defined in §2.1.5 required some creative invention. Doubly linked lists make reverse traversal as simple as forward traversal (defined in §2.1.4) except that we start at the tail node and update the pointers in the opposite direction. Figure 2.6 shows the reverse traversal algorithm in action.

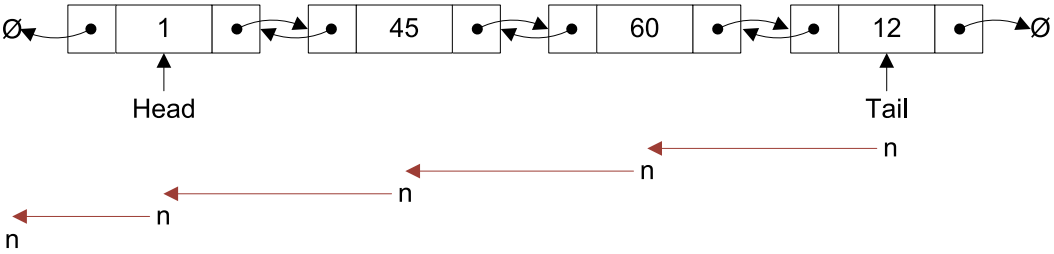


Figure 2.6: Doubly linked list reverse traversal

```
1) algorithm ReverseTraversal(tail)
2)     Pre: tail is the tail node of the list to traverse
3)     Post: the list has been traversed in reverse order
4)     n ← tail
5)     while n ≠ ∅
6)         yield n.Value
7)         n ← n.Previous
8)     end while
9) end ReverseTraversal
```

2.3 Summary

Linked lists are good to use when you have an unknown number of items to store. Using a data structure like an array would require you to specify the size up front; exceeding that size involves invoking a resizing algorithm which has a linear run time. You should also use linked lists when you will only remove nodes at either the head or tail of the list to maintain a constant run time. This requires maintaining pointers to the nodes at the head and tail of the list but the memory overhead will pay for itself if this is an operation you will be performing many times.

What linked lists are not very good for is random insertion, accessing nodes by index, and searching. At the expense of a little memory (in most cases 4 bytes would suffice), and a few more read/writes you could maintain a *count* variable that tracks how many items are contained in the list so that accessing such a primitive property is a constant operation - you just need to update *count* during the insertion and deletion algorithms.

Singly linked lists should be used when you are only performing basic insertions. In general doubly linked lists are more accommodating for non-trivial operations on a linked list.

We recommend the use of a doubly linked list when you require forwards and backwards traversal. For the most cases this requirement is present. For example, consider a token stream that you want to parse in a recursive descent fashion. Sometimes you will have to backtrack in order to create the correct parse tree. In this scenario a doubly linked list is best as its design makes bi-directional traversal much simpler and quicker than that of a singly linked

list.

Chapter 3

Binary Search Tree

Binary search trees (BSTs) are very simple to understand. We start with a root node with value x, where the left subtree of x contains nodes with values $< x$ and the right subtree contains nodes whose values are $\geq x$. Each node follows the same rules with respect to nodes in their left and right subtrees.

BSTs are of interest because they have operations which are favourably fast: insertion, look up, and deletion can all be done in $O(log\ n)$ time. It is important to note that the $O(log\ n)$ times for these operations can only be attained if the BST is reasonably balanced; for a tree data structure with self balancing properties see AVL tree defined in §7).

In the following examples you can assume, unless used as a parameter alias that *root* is a reference to the root node of the tree.

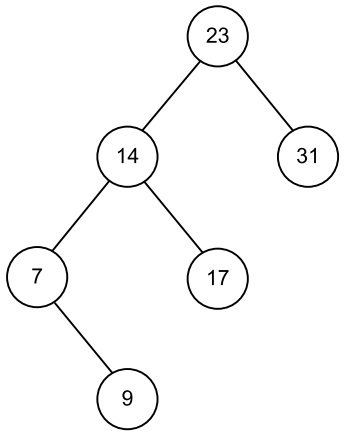


Figure 3.1: Simple unbalanced binary search tree

3.1 Insertion

As mentioned previously insertion is an $O(log\ n)$ operation provided that the tree is moderately balanced.

```
1) algorithm Insert(value)
2)     Pre: value has passed custom type checks for type T
3)     Post: value has been placed in the correct location in the tree
4)     if root = ∅
5)         root ← node(value)
6)     else
7)         InsertNode(root, value)
8)     end if
9) end Insert
```

```
1) algorithm InsertNode(current, value)
2)     Pre: current is the node to start from
3)     Post: value has been placed in the correct location in the tree
4)     if value < current.Value
5)         if current.Left = ∅
6)             current.Left ← node(value)
7)         else
8)             InsertNode(current.Left, value)
9)         end if
10)    else
11)        if current.Right = ∅
12)            current.Right ← node(value)
13)        else
14)            InsertNode(current.Right, value)
15)        end if
16)    end if
17) end InsertNode
```

The insertion algorithm is split for a good reason. The first algorithm (non-recursive) checks a very core base case - whether or not the tree is empty. If the tree is empty then we simply create our root node and finish. In all other cases we invoke the recursive *InsertNode* algorithm which simply guides us to the first appropriate place in the tree to put *value*. Note that at each stage we perform a binary chop: we either choose to recurse into the left subtree or the right by comparing the new value with that of the current node. For any totally ordered type, no value can simultaneously satisfy the conditions to place it in both subtrees.

3.2 Searching

Searching a BST is even simpler than insertion. The pseudocode is self-explanatory but we will look briefly at the premise of the algorithm nonetheless.

We have talked previously about insertion, we go either left or right with the right subtree containing values that are $\geq x$ where x is the value of the node we are inserting. When searching the rules are made a little more atomic and at any one time we have four cases to consider:

1. the $root = \emptyset$ in which case *value* is not in the BST; or
2. $root$.Value = *value* in which case *value* is in the BST; or
3. *value* < $root$.Value, we must inspect the left subtree of *root* for *value*; or
4. *value* > $root$.Value, we must inspect the right subtree of *root* for *value*.

```
1) algorithm Contains(root, value)
2)     Pre: root is the root node of the tree, value is what we would like to locate
3)     Post: value is either located or not
4)     if root = ∅
5)         return false
6)     end if
7)     if root.Value = value
8)         return true
9)     else if value < root.Value
10)        return Contains(root.Left, value)
11)    else
12)        return Contains(root.Right, value)
13)    end if
14) end Contains
```

3.3 Deletion

Removing a node from a BST is fairly straightforward, with four cases to consider:

1. the value to remove is a leaf node; or
2. the value to remove has a right subtree, but no left subtree; or
3. the value to remove has a left subtree, but no right subtree; or
4. the value to remove has both a left and right subtree in which case we promote the largest value in the left subtree.

There is also an implicit fifth case whereby the node to be removed is the only node in the tree. This case is already covered by the first, but should be noted as a possibility nonetheless.

Of course in a BST a value may occur more than once. In such a case the first occurrence of that value in the BST will be removed.

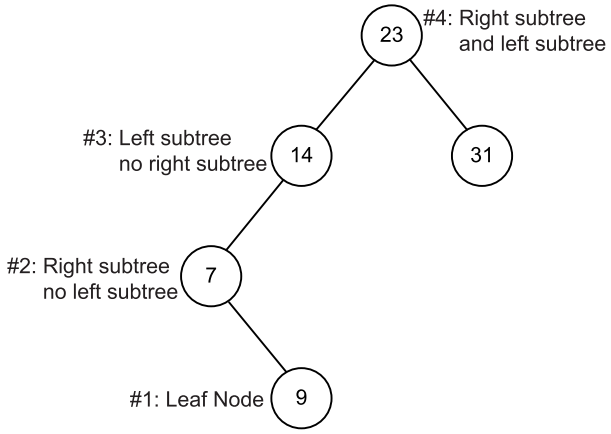


Figure 3.2: binary search tree deletion cases

The *Remove* algorithm given below relies on two further helper algorithms named $FindParent$, and $FindNode$ which are described in §3.4 and §3.5 respectively.

```
1) algorithm Remove(value)
2)    Pre:  value is the value of the node to remove, root is the root node of the BST
3)          Count is the number of items in the BST
3)    Post: node with value is removed if found in which case yields true, otherwise false
4)    nodeToRemove ← FindNode(value)
5)    if nodeToRemove = ∅
6)       return false // value not in BST
7)    end if
8)    parent ← FindParent(value)
9)    if Count = 1
10)      root ← ∅ // we are removing the only node in the BST
11)   else if nodeToRemove.Left = ∅ and nodeToRemove.Right = null
12)      // case #1
13)      if nodeToRemove.Value < parent.Value
14)         parent.Left ← ∅
15)      else
16)         parent.Right ← ∅
17)      end if
18)   else if nodeToRemove.Left = ∅ and nodeToRemove.Right ≠ ∅
19)      // case # 2
20)      if nodeToRemove.Value < parent.Value
21)         parent.Left ← nodeToRemove.Right
22)      else
23)         parent.Right ← nodeToRemove.Right
24)      end if
25)   else if nodeToRemove.Left ≠ ∅ and nodeToRemove.Right = ∅
26)      // case #3
27)      if nodeToRemove.Value < parent.Value
28)         parent.Left ← nodeToRemove.Left
29)      else
30)         parent.Right ← nodeToRemove.Left
31)      end if
32)   else
33)      // case #4
34)      largestValue ← nodeToRemove.Left
35)      while largestValue.Right ≠ ∅
36)         // find the largest value in the left subtree of nodeToRemove
37)         largestValue ← largestValue.Right
38)      end while
39)      // set the parents' Right pointer of largestValue to ∅
40)      FindParent(largestValue.Value).Right ← ∅
41)      nodeToRemove.Value ← largestValue.Value
42)   end if
43)   Count ← Count −1
44)   return true
45) end Remove
```

3.4 Finding the parent of a given node

The purpose of this algorithm is simple - to return a reference (or pointer) to the parent node of the one with the given value. We have found that such an algorithm is very useful, especially when performing extensive tree transformations.

```
1) algorithm FindParent(value, root)
2)     Pre:  value is the value of the node we want to find the parent of
3)           root is the root node of the BST and is ! = ∅
4)     Post: a reference to the parent node of value if found; otherwise ∅
5)     if value = root.Value
6)       return ∅
7)     end if
8)     if value < root.Value
9)       if root.Left = ∅
10)        return ∅
11)      else if root.Left.Value = value
12)        return root
13)      else
14)        return FindParent(value, root.Left)
15)      end if
16)    else
17)      if root.Right = ∅
18)        return ∅
19)      else if root.Right.Value = value
20)        return root
21)      else
22)        return FindParent(value, root.Right)
23)      end if
24)    end if
25) end FindParent
```

A special case in the above algorithm is when the specified value does not exist in the BST, in which case we return ∅. Callers to this algorithm must take account of this possibility unless they are already certain that a node with the specified value exists.

3.5 Attaining a reference to a node

This algorithm is very similar to §3.4, but instead of returning a reference to the parent of the node with the specified value, it returns a reference to the node itself. Again, ∅ is returned if the value isn't found.

```
1) algorithm FindNode(root, value)
2)     Pre: value is the value of the node we want to find the parent of
3)          root is the root node of the BST
4)     Post: a reference to the node of value if found; otherwise ∅
5)     if root = ∅
6)       return ∅
7)     end if
8)     if root.Value = value
9)       return root
10)    else if value < root.Value
11)      return FindNode(root.Left, value)
12)    else
13)      return FindNode(root.Right, value)
14)    end if
15) end FindNode
```

Astute readers will have noticed that the *FindNode* algorithm is exactly the same as the *Contains* algorithm (defined in §3.2) with the modification that we are returning a reference to a node not $true$ or $false$. Given *FindNode*, the easiest way of implementing *Contains* is to call *FindNode* and compare the return value with $\emptyset$.

3.6 Finding the smallest and largest values in the binary search tree

To find the smallest value in a BST you simply traverse the nodes in the left subtree of the BST always going left upon each encounter with a node, terminating when you find a node with no left subtree. The opposite is the case when finding the largest value in the BST. Both algorithms are incredibly simple, and are listed simply for completeness.

The base case in both $FindMin$, and $FindMax$ algorithms is when the Left ($FindMin$), or Right ($FindMax$) node references are $\emptyset$ in which case we have reached the last node.

```
1) algorithm FindMin(root)
2)     Pre: root is the root node of the BST
3)          root ≠ ∅
4)     Post: the smallest value in the BST is located
5)     if root.Left = ∅
6)       return root.Value
7)     end if
8)     FindMin(root.Left)
9) end FindMin
```

```
1) algorithm FindMax(root)
2)     Pre: root is the root node of the BST
3)          root ≠ ∅
4)     Post: the largest value in the BST is located
5)     if root.Right = ∅
6)         return root.Value
7)     end if
8)     FindMax(root.Right)
9) end FindMax
```

3.7 Tree Traversals

There are various strategies which can be employed to traverse the items in a tree; the choice of strategy depends on which node visitation order you require. In this section we will touch on the traversals that DSA provides on all data structures that derive from *BinarySearchTree*.

3.7.1 Preorder

When using the preorder algorithm, you visit the root first, then traverse the left subtree and finally traverse the right subtree. An example of preorder traversal is shown in Figure 3.3.

```
1) algorithm Preorder(root)
2)     Pre: root is the root node of the BST
3)     Post: the nodes in the BST have been visited in preorder
4)     if root ≠ ∅
5)         yield root.Value
6)         Preorder(root.Left)
7)         Preorder(root.Right)
8)     end if
9) end Preorder
```

3.7.2 Postorder

This algorithm is very similar to that described in §3.7.1, however the value of the node is yielded after traversing both subtrees. An example of postorder traversal is shown in Figure 3.4.

```
1) algorithm Postorder(root)
2)     Pre: root is the root node of the BST
3)     Post: the nodes in the BST have been visited in postorder
4)     if root ≠ ∅
5)         Postorder(root.Left)
6)         Postorder(root.Right)
7)         yield root.Value
8)     end if
9) end Postorder
```

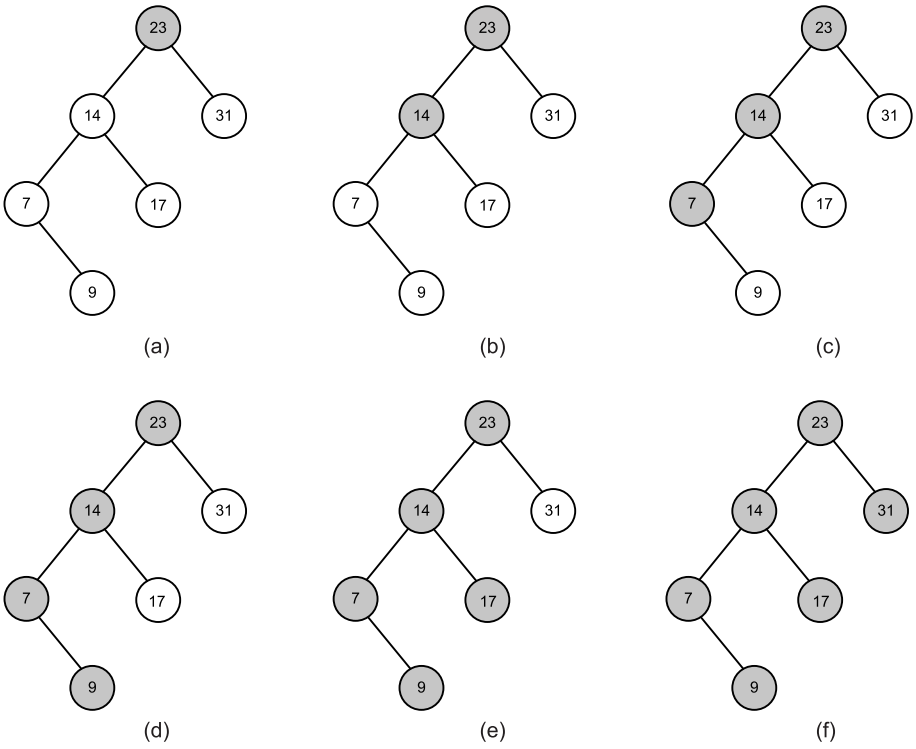


Figure 3.3: Preorder visit binary search tree example

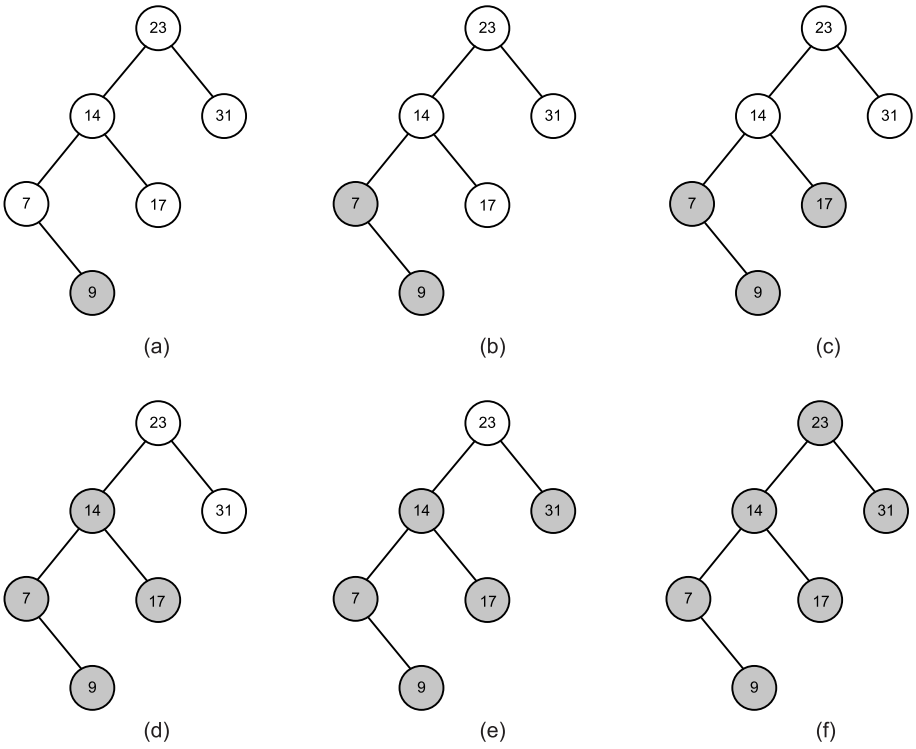


Figure 3.4: Postorder visit binary search tree example

3.7.3 Inorder

Another variation of the algorithms defined in §3.7.1 and §3.7.2 is that of inorder traversal where the value of the current node is yielded in between traversing the left subtree and the right subtree. An example of inorder traversal is shown in Figure 3.5.

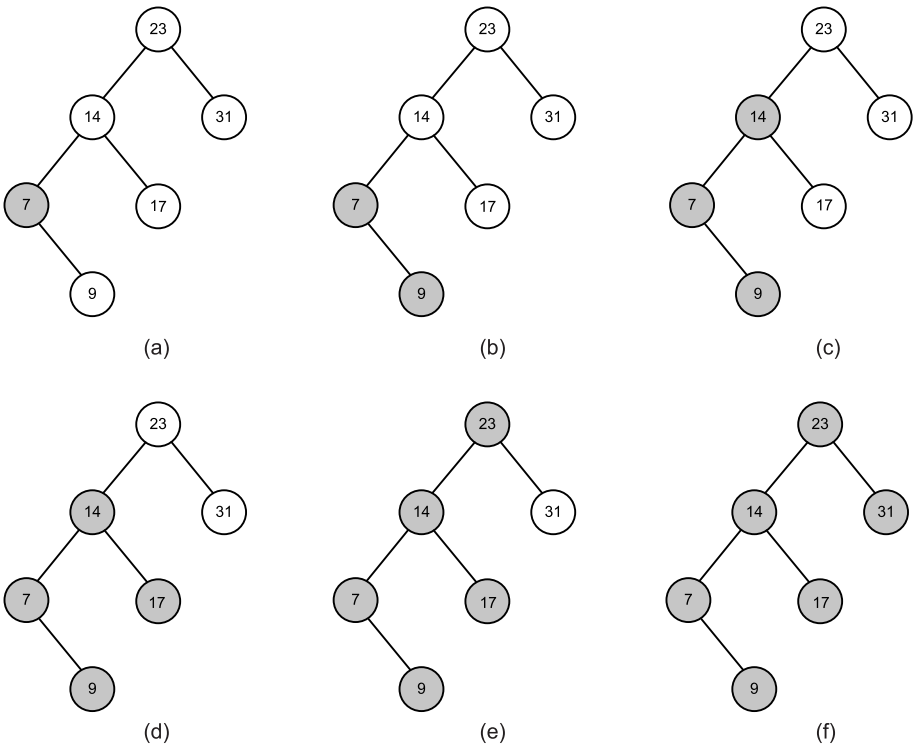


Figure 3.5: Inorder visit binary search tree example

1) **algorithm** Inorder(*root*)
2) **Pre:** *root* is the root node of the BST
3) **Post:** the nodes in the BST have been visited in inorder
4) **if** *root* $\neq \emptyset$
5) Inorder(*root*.Left)
6) **yield** *root*.Value
7) Inorder(*root*.Right)
8) **end if**
9) **end** Inorder

One of the beauties of inorder traversal is that values are yielded in their comparison order. In other words, when traversing a populated BST with the inorder strategy, the yielded sequence would have property $x_i \leq x_{i+1} \forall i$.

3.7.4 Breadth First

Traversing a tree in breadth first order yields the values of all nodes of a particular depth in the tree before any deeper ones. In other words, given a depth d we would visit the values of all nodes at d in a left to right fashion, then we would proceed to $d + 1$ and so on until we hade no more nodes to visit. An example of breadth first traversal is shown in Figure 3.6.

Traditionally breadth first traversal is implemented using a list (vector, resizeable array, etc) to store the values of the nodes visited in breadth first order and then a queue to store those nodes that have yet to be visited.

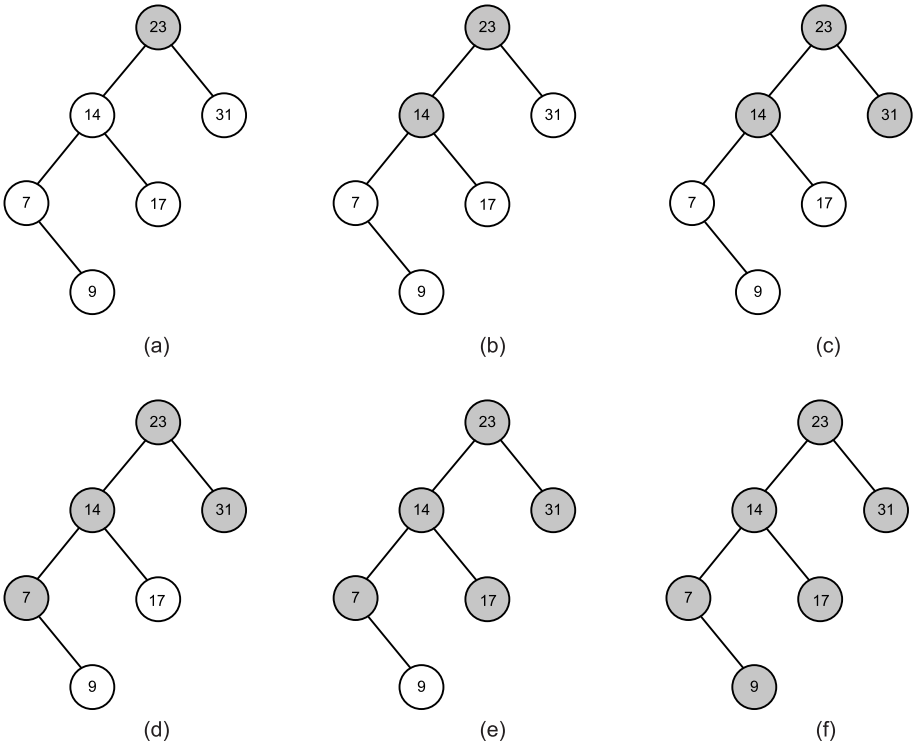


Figure 3.6: Breadth First visit binary search tree example

```
1) algorithm BreadthFirst(root)
2)     Pre: root is the root node of the BST
3)     Post: the nodes in the BST have been visited in breadth first order
4)     q ← queue
5)     while root ≠ ∅
6)         yield root.Value
7)         if root.Left ≠ ∅
8)             q.Enqueue(root.Left)
9)         end if
10)        if root.Right ≠ ∅
11)            q.Enqueue(root.Right)
12)        end if
13)        if !q.IsEmpty()
14)            root ← q.Dequeue()
15)        else
16)            root ← ∅
17)        end if
18)    end while
19) end BreadthFirst
```

3.8 Summary

A binary search tree is a good solution when you need to represent types that are ordered according to some custom rules inherent to that type. With logarithmic insertion, lookup, and deletion it is very effecient. Traversal remains linear, but there are many ways in which you can visit the nodes of a tree. Trees are recursive data structures, so typically you will find that many algorithms that operate on a tree are recursive.

The run times presented in this chapter are based on a pretty big assumption - that the binary search tree's left and right subtrees are reasonably balanced. We can only attain logarithmic run times for the algorithms presented earlier when this is true. A binary search tree does not enforce such a property, and the run times for these operations on a pathologically unbalanced tree become linear: such a tree is effectively just a linked list. Later in §7 we will examine an AVL tree that enforces self-balancing properties to help attain logarithmic run times.

Chapter 4

Heap

A heap can be thought of as a simple tree data structure, however a heap usually employs one of two strategies:

1. min heap; or
2. max heap

Each strategy determines the properties of the tree and its values. If you were to choose the min heap strategy then each parent node would have a value that is $\leq$ than its children. For example, the node at the root of the tree will have the smallest value in the tree. The opposite is true for the max heap strategy. In this book you should assume that a heap employs the min heap strategy unless otherwise stated.

Unlike other tree data structures like the one defined in §3 a heap is generally implemented as an array rather than a series of nodes which each have references to other nodes. The nodes are conceptually the same, however, having at most two children. Figure 4.1 shows how the tree (not a heap data structure) (12 7(3 2) 6(9)) would be represented as an array. The array in Figure 4.1 is a result of simply adding values in a top-to-bottom, left-to-right fashion. Figure 4.2 shows arrows to the direct left and right child of each value in the array.

This chapter is very much centred around the notion of representing a tree as an array and because this property is key to understanding this chapter Figure 4.3 shows a step by step process to represent a tree data structure as an array. In Figure 4.3 you can assume that the default capacity of our array is eight.

Using just an array is often not sufficient as we have to be up front about the size of the array to use for the heap. Often the run time behaviour of a program can be unpredictable when it comes to the size of its internal data structures, so we need to choose a more dynamic data structure that contains the following properties:

1. we can specify an initial size of the array for scenarios where we know the upper storage limit required; and
2. the data structure encapsulates resizing algorithms to grow the array as required at run time

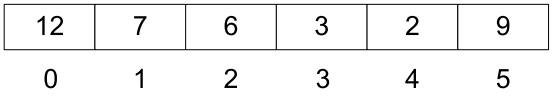


Figure 4.1: Array representation of a simple tree data structure

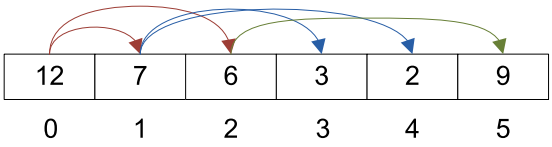


Figure 4.2: Direct children of the nodes in an array representation of a tree data structure

1. Vector
2. ArrayList
3. List

Figure 4.1 does not specify how we would handle adding null references to the heap. This varies from case to case; sometimes null values are prohibited entirely; in other cases we may treat them as being smaller than any non-null value, or indeed greater than any non-null value. You will have to resolve this ambiguity yourself having studied your requirements. For the sake of clarity we will avoid the issue by prohibiting null values.

Because we are using an array we need some way to calculate the index of a parent node, and the children of a node. The required expressions for this are defined as follows for a node at *index*:

1. $(index - 1)/2$ (parent index)
2. $2 * index + 1$ (left child)
3. $2 * index + 2$ (right child)

In Figure 4.4 a) represents the calculation of the right child of 12 $(2 * 0 + 2)$; and b) calculates the index of the parent of 3 $((3 - 1)/2)$.

4.1 Insertion

Designing an algorithm for heap insertion is simple, but we must ensure that heap order is preserved after each insertion. Generally this is a post-insertion operation. Inserting a value into the next free slot in an array is simple: we just need to keep track of the next free index in the array as a counter, and increment it after each insertion. Inserting our value into the heap is the first part of the algorithm; the second is validating heap order. In the case of min-heap ordering this requires us to swap the values of a parent and its child if the value of the child is $<$ the value of its parent. We must do this for each subtree containing the value we just inserted.

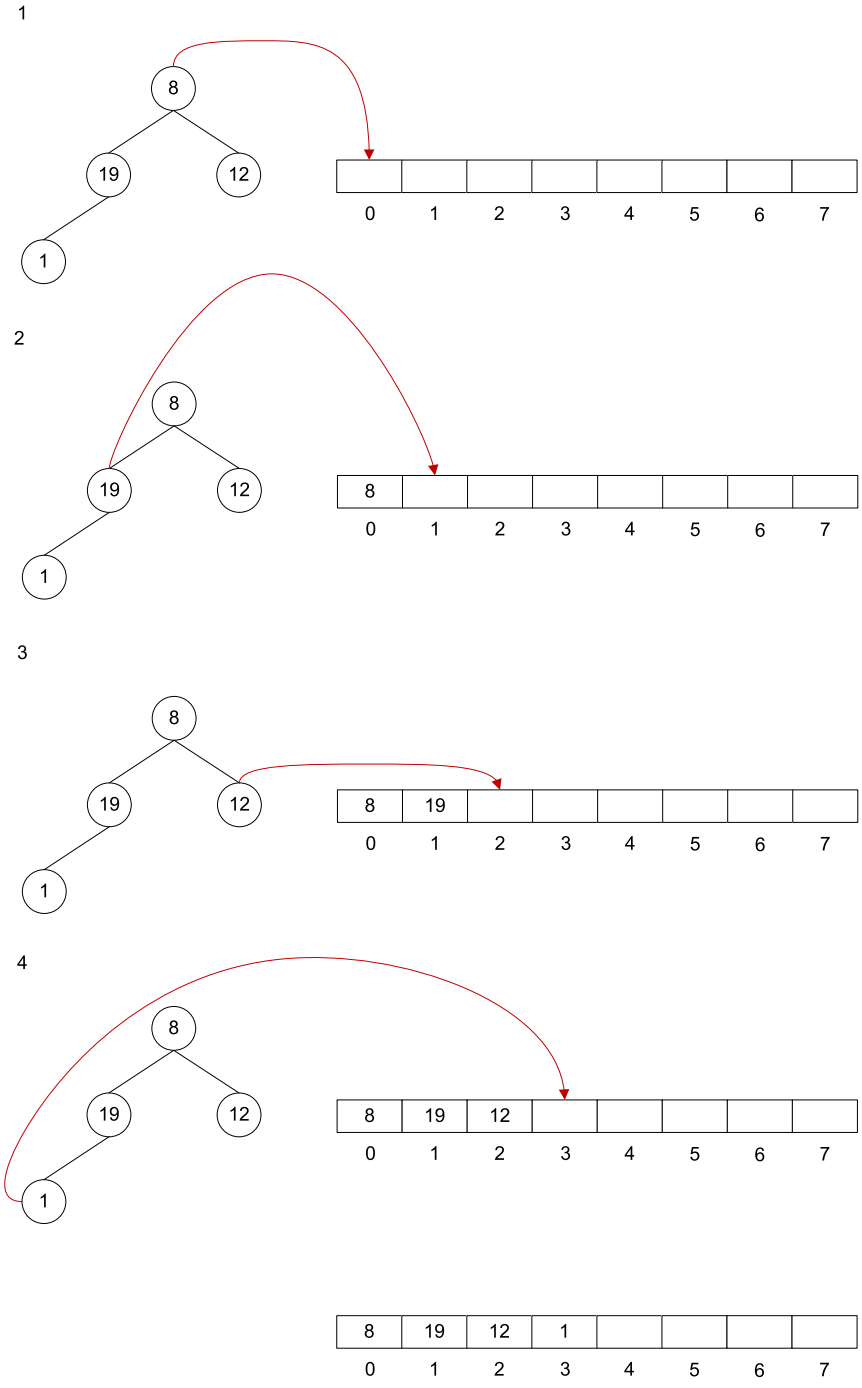


Figure 4.3: Converting a tree data structure to its array counterpart

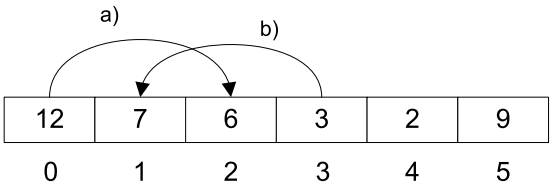


Figure 4.4: Calculating node properties

The run time efficiency for heap insertion is $O(log\ n)$. The run time is a by product of verifying heap order as the first part of the algorithm (the actual insertion into the array) is $O(1)$.

Figure 4.5 shows the steps of inserting the values 3, 9, 12, 7, and 1 into a min-heap.

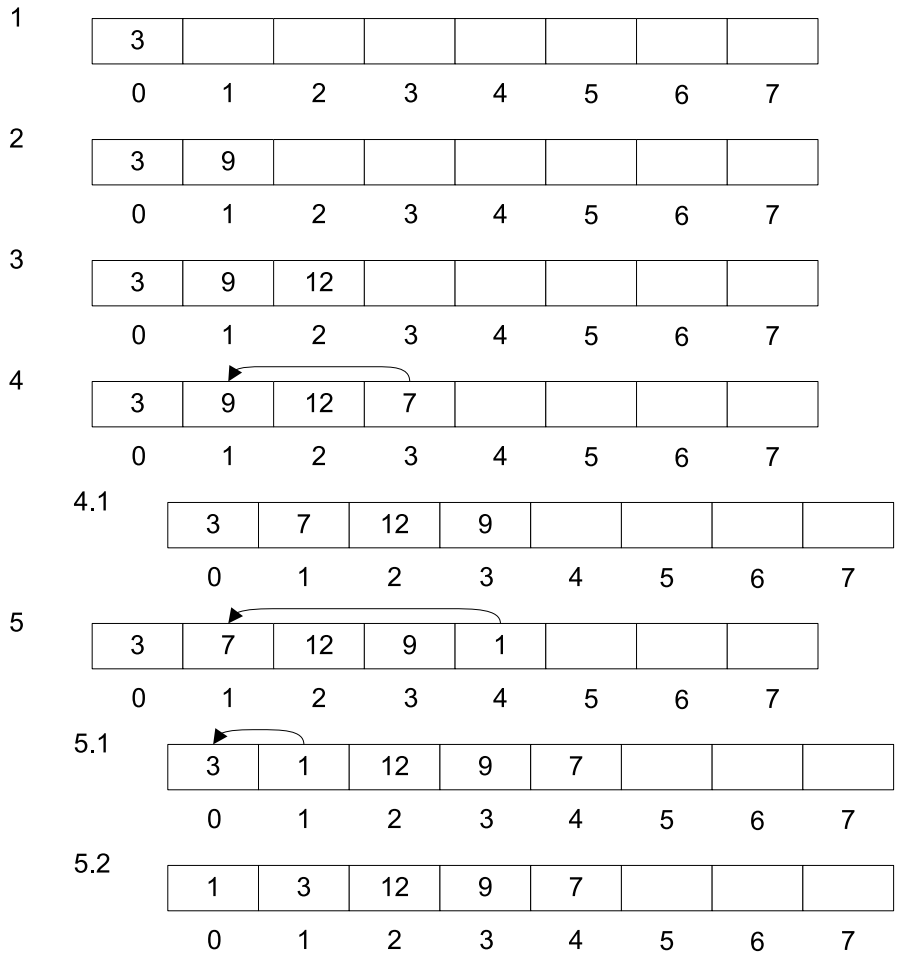


Figure 4.5: Inserting values into a min-heap

```
1) algorithm Add(value)
2)     Pre: value is the value to add to the heap
3)          Count is the number of items in the heap
4)     Post: the value has been added to the heap
5)     heap[Count] ← value
6)     Count ← Count +1
7)     MinHeapify()
8) end Add
```

```
1) algorithm MinHeapify()
2)     Pre: Count is the number of items in the heap
3)          heap is the array used to store the heap items
4)     Post: the heap has preserved min heap ordering
5)     i ← Count −1
6)     while i > 0 and heap[i] < heap[(i − 1)/2]
7)         Swap(heap[i], heap[(i − 1)/2]
8)         i ← (i − 1)/2
9)     end while
10) end MinHeapify
```

The design of the *MaxHeapify* algorithm is very similar to that of the *MinHeapify* algorithm, the only difference is that the $<$ operator in the second condition of entering the while loop is changed to $>$.

4.2 Deletion

Just as for insertion, deleting an item involves ensuring that heap ordering is preserved. The algorithm for deletion has three steps:

1. find the index of the value to delete
2. put the last value in the heap at the index location of the item to delete
3. verify heap ordering for each subtree which used to include the value

```
1) algorithm Remove(value)
2)     Pre: value is the value to remove from the heap
3)          left, and right are updated alias' for 2 * index + 1, and 2 * index + 2 respectively
4)          Count is the number of items in the heap
5)          heap is the array used to store the heap items
6)     Post: value is located in the heap and removed, true; otherwise false
7)     // step 1
8)     index ← FindIndex(heap, value)
9)     if index < 0
10)        return false
11)    end if
12)    Count ← Count −1
13)    // step 2
14)    heap[index] ← heap[Count]
15)    // step 3
16)    while left < Count and heap[index] > heap[left] or heap[index] > heap[right]
17)        // promote smallest key from subtree
18)        if heap[left] < heap[right]
19)            Swap(heap, left, index)
20)            index ← left
21)        else
22)            Swap(heap, right, index)
23)            index ← right
24)        end if
25)    end while
26)    return true
27) end Remove
```

Figure 4.6 shows the *Remove* algorithm visually, removing 1 from a heap containing the values 1, 3, 9, 12, and 13. In Figure 4.6 you can assume that we have specified that the backing array of the heap should have an initial capacity of eight.

Please note that in our deletion algorithm that we don't default the removed value in the *heap* array. If you are using a heap for reference types, i.e. objects that are allocated on a heap you will want to free that memory. This is important in both unmanaged, and managed languages. In the latter we will want to null that empty hole so that the garbage collector can reclaim that memory. If we were to not null that hole then the object could still be reached and thus won't be garbage collected.

4.3 Searching

Searching a heap is merely a matter of traversing the items in the heap array sequentially, so this operation has a run time complexity of $O(n)$. The search can be thought of as one that uses a breadth first traversal as defined in §3.7.4 to visit the nodes within the heap to check for the presence of a specified item.

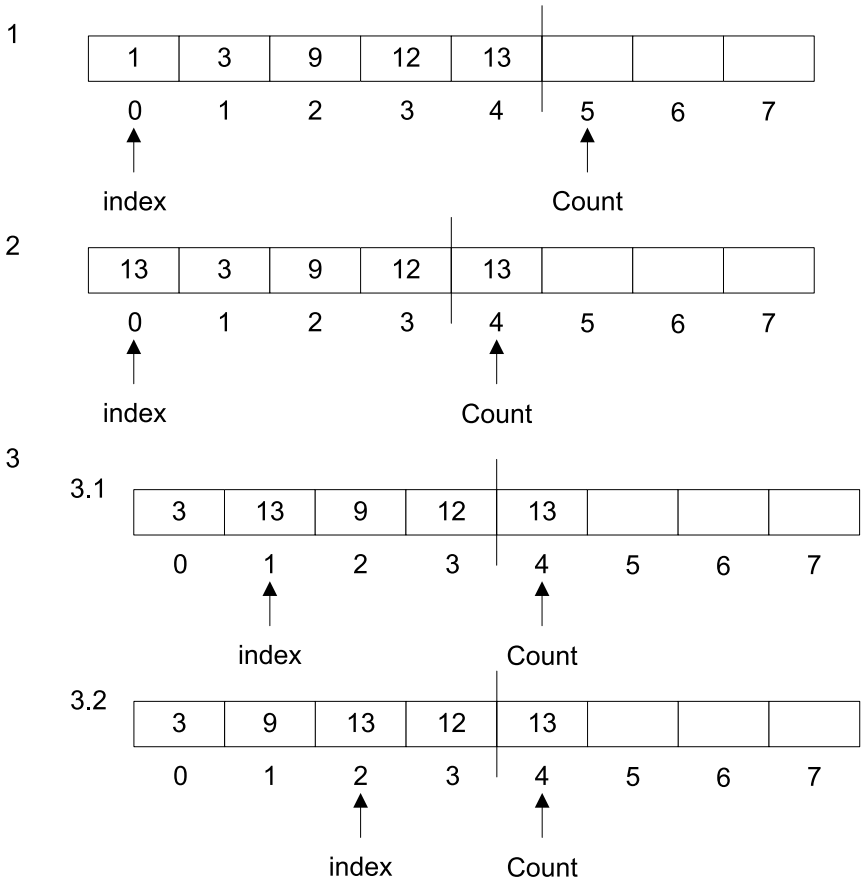


Figure 4.6: Deleting an item from a heap