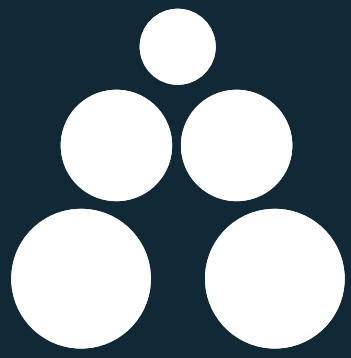
Data Structures and Algorithms

DSA



Atul Kumar

Annotated Reference with Examples

Granville Barnett

Luca Del Tongo

Contents

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.

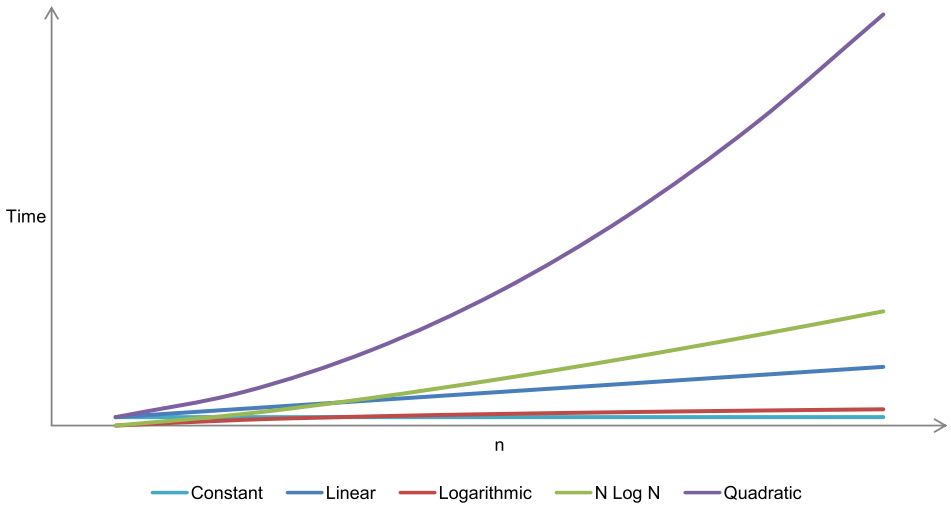


Figure 1.1: Algorithmic run time expansion

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.

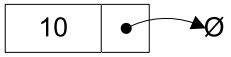


Figure 2.1: Singly linked list node

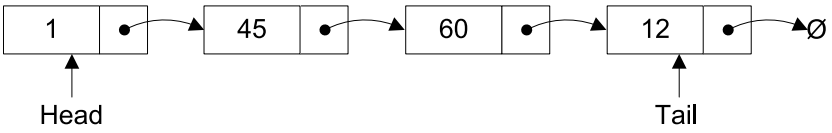


Figure 2.2: A singly linked list populated with integers

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.

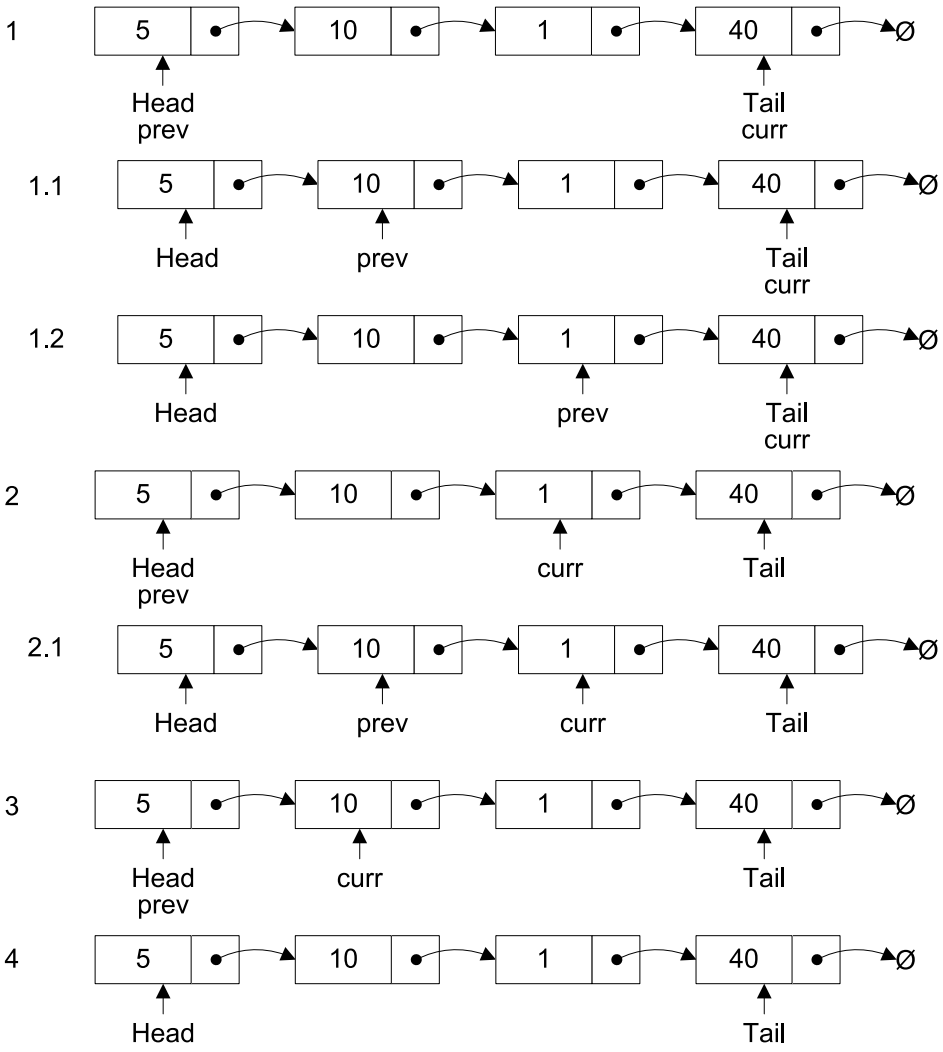


Figure 2.3: Reverse traveral of a singly linked list

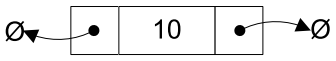


Figure 2.4: Doubly linked list node

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.

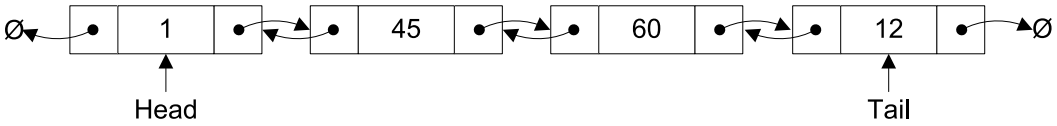


Figure 2.5: Doubly linked list populated with integers

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.

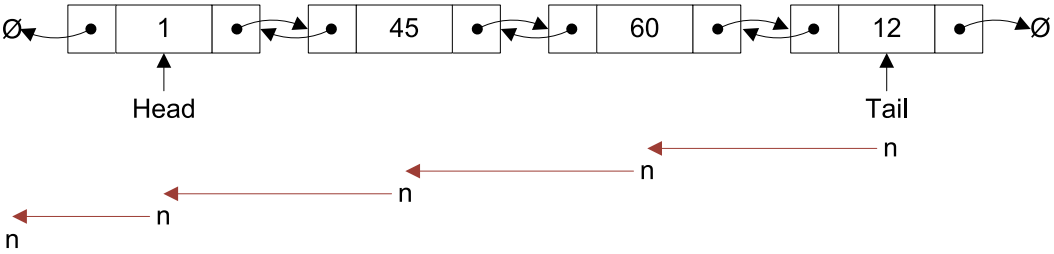


Figure 2.6: Doubly linked list reverse traversal

```
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.

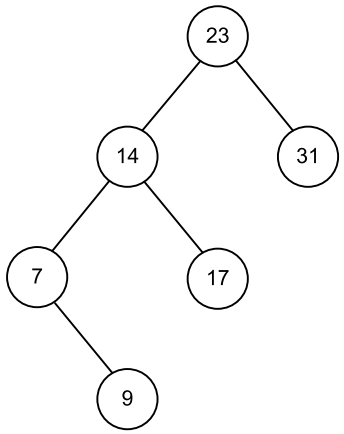


Figure 3.1: Simple unbalanced binary search tree

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.

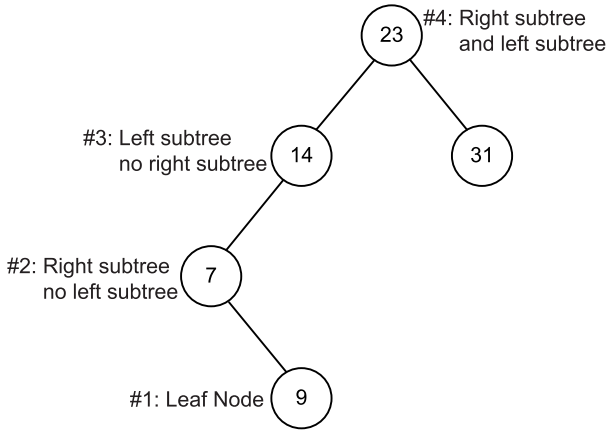


Figure 3.2: binary search tree deletion cases

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
```

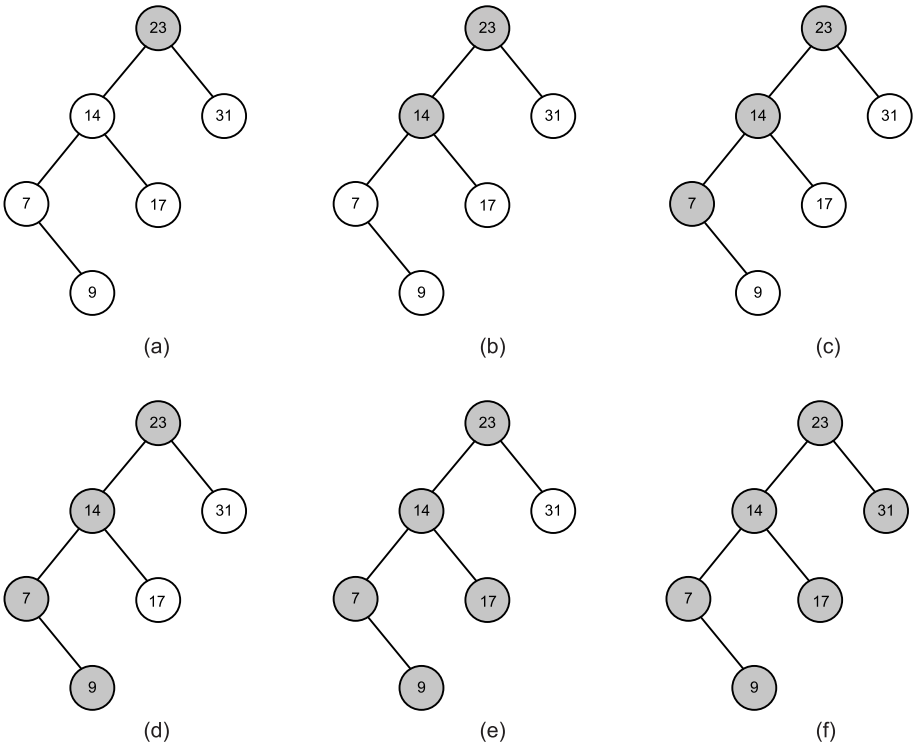


Figure 3.3: Preorder visit binary search tree example

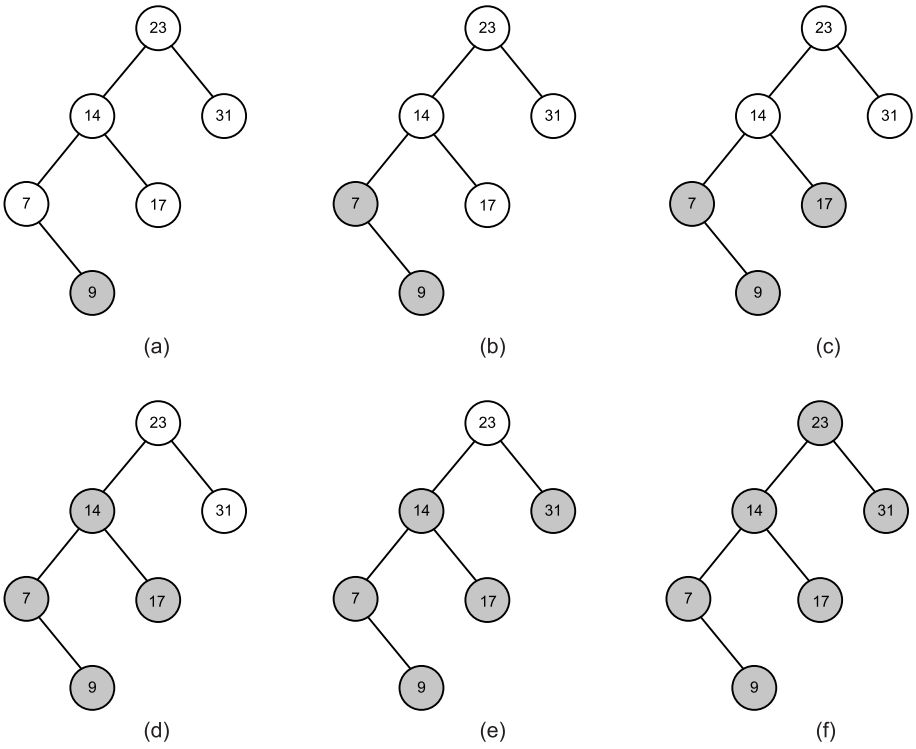


Figure 3.4: Postorder visit binary search tree example

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.

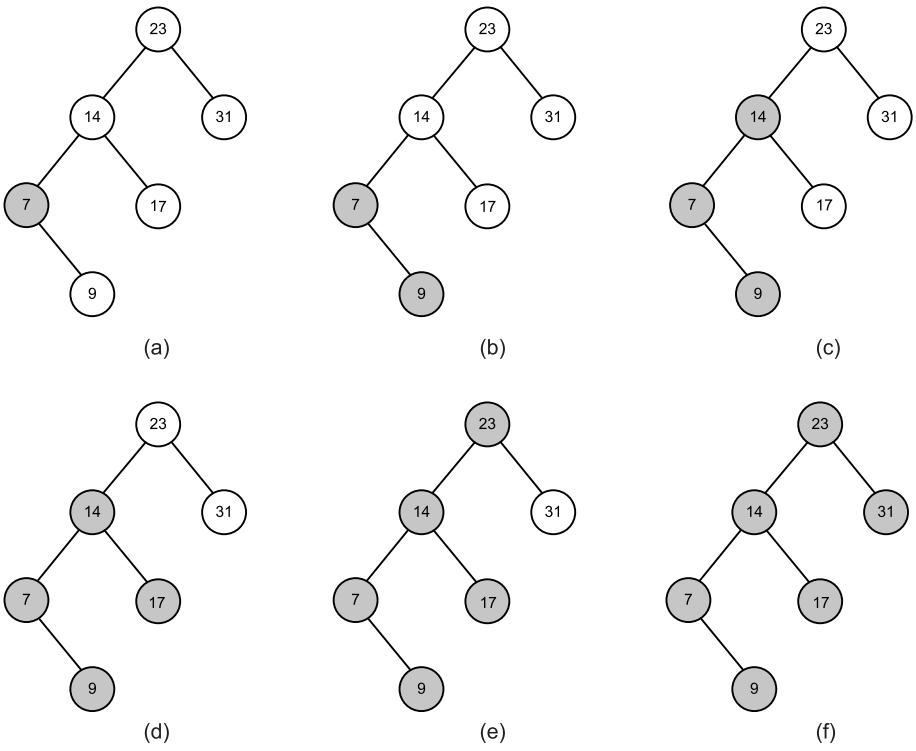


Figure 3.5: Inorder visit binary search tree example

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.

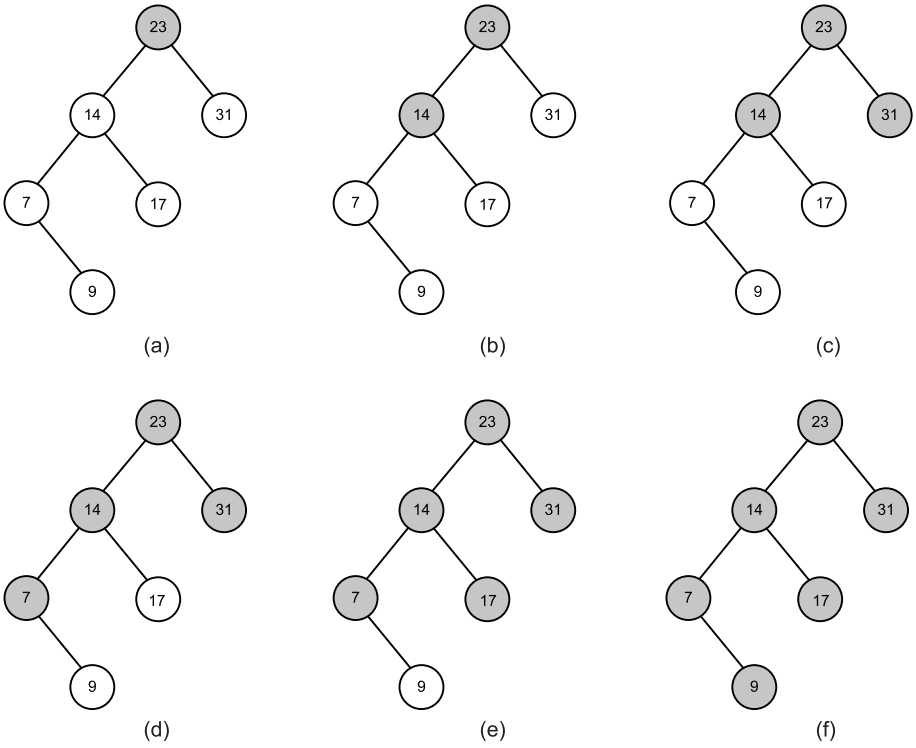


Figure 3.6: Breadth First visit binary search tree example

```
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

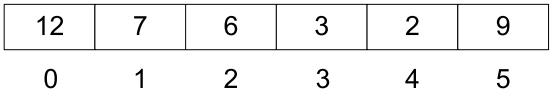


Figure 4.1: Array representation of a simple tree data structure

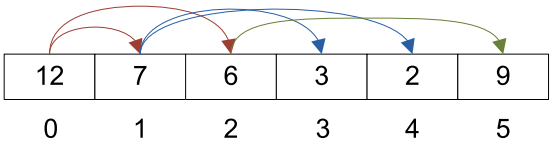


Figure 4.2: Direct children of the nodes in an array representation of a tree data structure

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.

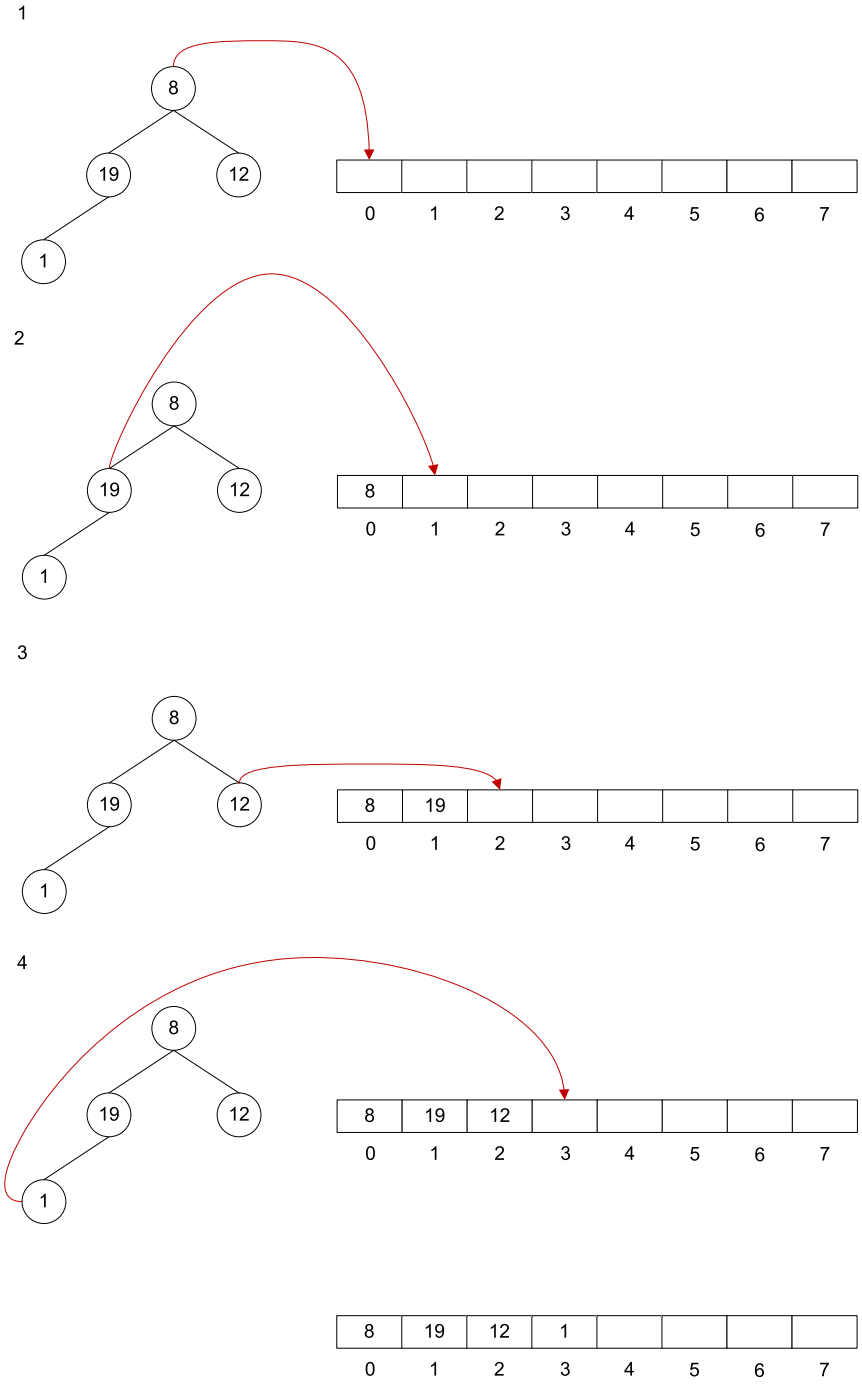


Figure 4.3: Converting a tree data structure to its array counterpart

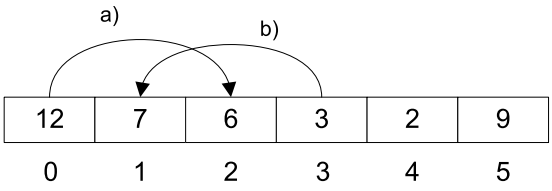


Figure 4.4: Calculating node properties

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.

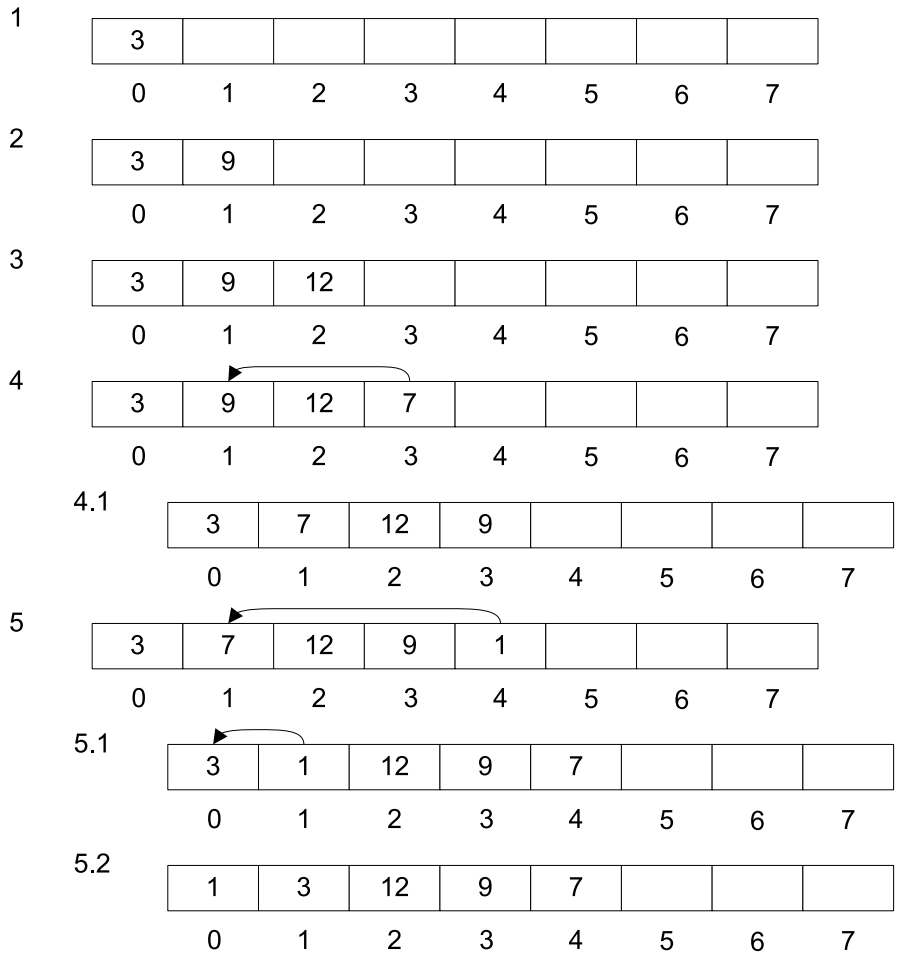


Figure 4.5: Inserting values into a min-heap

```
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.

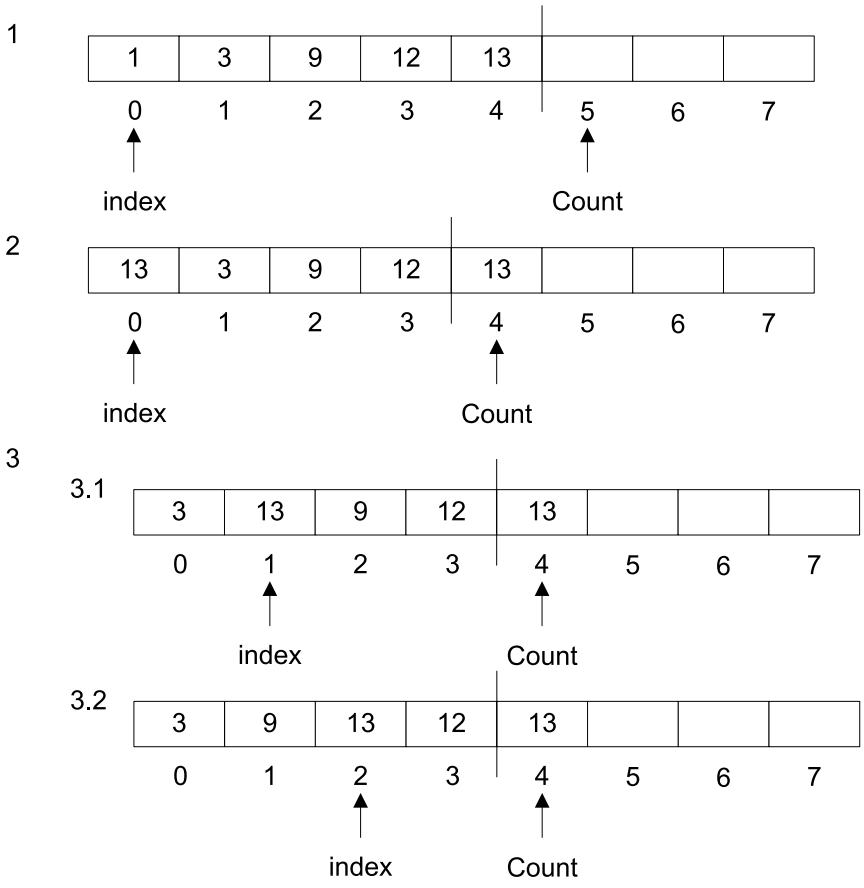


Figure 4.6: Deleting an item from a heap

```
1) algorithm Contains(value)
2)     Pre: value is the value to search the heap for
3)          Count is the number of items in the heap
4)          heap is the array used to store the heap items
5)     Post: value is located in the heap, in which case true; otherwise false
6)     i ← 0
7)     while i < Count and heap[i] ≠ value
8)         i ← i + 1
9)     end while
10)    if i < Count
11)        return true
12)    else
13)        return false
14)    end if
15) end Contains
```

The problem with the previous algorithm is that we don't take advantage of the properties in which all values of a heap hold, that is the property of the heap strategy being used. For instance if we had a heap that didn't contain the value 4 we would have to exhaust the whole backing heap array before we could determine that it wasn't present in the heap. Factoring in what we know about the heap we can optimise the search algorithm by including logic which makes use of the properties presented by a certain heap strategy.

Optimising to deterministically state that a value is in the heap is not that straightforward, however the problem is a very interesting one. As an example consider a min-heap that doesn't contain the value 5. We can only rule that the value is not in the heap if $5 >$ the parent of the current node being inspected and $<$ the current node being inspected $\forall$ nodes at the current level we are traversing. If this is the case then 5 cannot be in the heap and so we can provide an answer without traversing the rest of the heap. If this property is not satisfied for any level of nodes that we are inspecting then the algorithm will indeed fall back to inspecting all the nodes in the heap. The optimisation that we present can be very common and so we feel that the extra logic within the loop is justified to prevent the expensive worse case run time.

The following algorithm is specifically designed for a min-heap. To tailor the algorithm for a max-heap the two comparison operations in the **else if** condition within the inner **while** loop should be flipped.

```
1) algorithm Contains(value)
2)     Pre: value is the value to search the heap for
3)          Count is the number of items in the heap
4)          heap is the array used to store the heap items
5)     Post: value is located in the heap, in which case true; otherwise false
6)     start ← 0
7)     nodes ← 1
8)     while start < Count
9)         start ← nodes − 1
10)        end ← nodes + start
11)        count ← 0
12)        while start < Count and start < end
13)            if value = heap[start]
14)                return true
15)            else if value > Parent(heap[start]) and value < heap[start]
16)                count ← count + 1
17)            end if
18)            start ← start + 1
19)        end while
20)        if count = nodes
21)            return false
22)        end if
23)        nodes ← nodes ∗ 2
24)    end while
25)    return false
26) end Contains
```

The new *Contains* algorithm determines if the *value* is not in the heap by checking whether $count = nodes$. In such an event where this is true then we can confirm that $\forall$ nodes n at level $i : value >$ Parent(n), $value < n$ thus there is no possible way that *value* is in the heap. As an example consider Figure 4.7. If we are searching for the value 10 within the min-heap displayed it is obvious that we don't need to search the whole heap to determine 9 is not present. We can verify this after traversing the nodes in the second level of the heap as the previous expression defined holds true.

4.4 Traversal

As mentioned in §4.3 traversal of a heap is usually done like that of any other array data structure which our heap implementation is based upon. As a result you traverse the array starting at the initial array index (0 in most languages) and then visit each value within the array until you have reached the upper bound of the heap. You will note that in the search algorithm that we use *Count* as this upper bound rather than the actual physical bound of the allocated array. *Count* is used to partition the conceptual heap from the actual array implementation of the heap: we only care about the items in the heap, not the whole array—the latter may contain various other bits of data as a result of heap mutation.

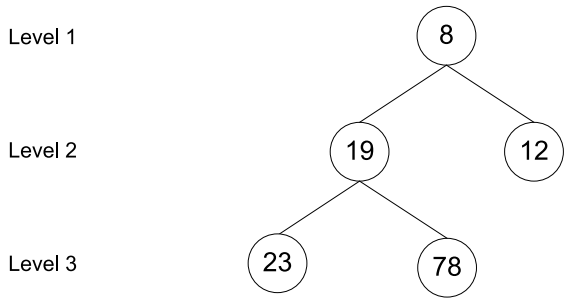


Figure 4.7: Determining 10 is not in the heap after inspecting the nodes of Level 2

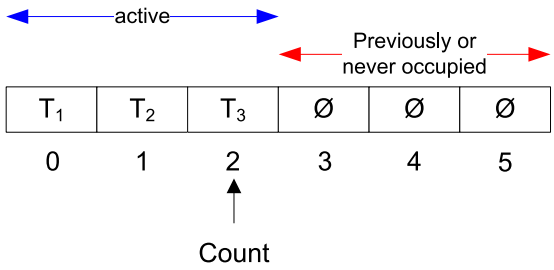


Figure 4.8: Living and dead space in the heap backing array

If you have followed the advice we gave in the deletion algorithm then a heap that has been mutated several times will contain some form of default value for items no longer in the heap. Potentially you will have at most $LengthOf(heapArray) - Count$ garbage values in the backing heap array data structure. The garbage values of course vary from platform to platform. To make things simple the garbage value of a reference type will be simple $\emptyset$ and 0 for a value type.

Figure 4.8 shows a heap that you can assume has been mutated many times. For this example we can further assume that at some point the items in indexes $3 - 5$ actually contained references to live objects of type T. In Figure 4.8 subscript is used to disambiguate separate objects of T.

From what you have read thus far you will most likely have picked up that traversing the heap in any other order would be of little benefit. The heap property only holds for the subtree of each node and so traversing a heap in any other fashion requires some creative intervention. Heaps are not usually traversed in any other way than the one prescribed previously.

4.5 Summary

Heaps are most commonly used to implement priority queues (see §6.2 for a sample implementation) and to facilitate heap sort. As discussed in both the insertion §4.1 and deletion §4.2 sections a heap maintains heap order according to the selected ordering strategy. These strategies are referred to as min-heap,

and max heap. The former strategy enforces that the value of a parent node is less than that of each of its children, the latter enforces that the value of the parent is greater than that of each of its children.

When you come across a heap and you are not told what strategy it enforces you should assume that it uses the min-heap strategy. If the heap can be configured otherwise, e.g. to use max-heap then this will often require you to state this explicitly. The heap abides progressively to a strategy during the invocation of the insertion, and deletion algorithms. The cost of such a policy is that upon each insertion and deletion we invoke algorithms that have logarithmic run time complexities. While the cost of maintaining the strategy might not seem overly expensive it does still come at a price. We will also have to factor in the cost of dynamic array expansion at some stage. This will occur if the number of items within the heap outgrows the space allocated in the heap's backing array. It may be in your best interest to research a good initial starting size for your heap array. This will assist in minimising the impact of dynamic array resizing.

Chapter 5

Sets

A set contains a number of values, in no particular order. The values within the set are distinct from one another.

Generally set implementations tend to check that a value is not in the set before adding it, avoiding the issue of repeated values from ever occurring.

This section does not cover set theory in depth; rather it demonstrates briefly the ways in which the values of sets can be defined, and common operations that may be performed upon them.

The notation $A = \{4, 7, 9, 12, 0\}$ defines a set A whose values are listed within the curly braces.

Given the set A defined previously we can say that 4 is a member of A denoted by $4 \in A$, and that 99 is not a member of A denoted by $99 \notin A$.

Often defining a set by manually stating its members is tiresome, and more importantly the set may contain a large number of values. A more concise way of defining a set and its members is by providing a series of properties that the values of the set must satisfy. For example, from the definition $A = \{x | x > 0, x \ \% \ 2 = 0\}$ the set A contains only positive integers that are even. x is an alias to the current value we are inspecting and to the right hand side of $|$ are the properties that x must satisfy to be in the set A. In this example, x must be > 0, and the remainder of the arithmetic expression $x/2$ must be 0. You will be able to note from the previous definition of the set A that the set can contain an infinite number of values, and that the values of the set A will be all even integers that are a member of the natural numbers set $\mathbb{N}$, where $\mathbb{N} = \{1, 2, 3, ...\}$.

Finally in this brief introduction to sets we will cover set intersection and union, both of which are very common operations (amongst many others) performed on sets. The union set can be defined as follows $A \cup B = \{x \mid x \in A \ or \ x \in B\}$, and intersection $A \cap B = \{x \mid x \in A \ and \ x \in B\}$. Figure 5.1 demonstrates set intersection and union graphically.

Given the set definitions $A = \{1, 2, 3\}$, and $B = \{6, 2, 9\}$ the union of the two sets is $A \cup B = \{1, 2, 3, 6, 9\}$, and the intersection of the two sets is $A \cap B = \{2\}$.

Both set union and intersection are sometimes provided within the framework associated with mainstream languages. This is the case in .NET 3.5[1] where such algorithms exist as extension methods defined in the type *System.Linq.Enumerable*[2], as a result DSA does not provide implementations of

[1] `http://www.microsoft.com/NET/`

[2] `http://msdn.microsoft.com/en-us/library/system.linq.enumerable_members.aspx`

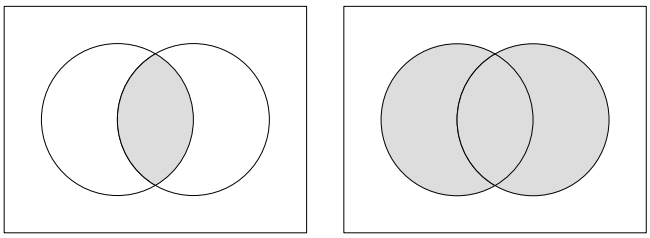


Figure 5.1: a) $A \cap B$; b) $A \cup B$

these algorithms. Most of the algorithms defined in *System.Linq.Enumerable* deal mainly with sequences rather than sets exclusively.

Set union can be implemented as a simple traversal of both sets adding each item of the two sets to a new union set.

```
1) algorithm Union(set1, set2)
2)     Pre:  set1, and set2 ≠ ∅
3)           union is a set
3)     Post: A union of set1, and set2 has been created
4)     foreach item in set1
5)         union.Add(item)
6)     end foreach
7)     foreach item in set2
8)         union.Add(item)
9)     end foreach
10)    return union
11) end Union
```

The run time of our *Union* algorithm is $O(m + n)$ where m is the number of items in the first set and n is the number of items in the second set. This runtime applies only to sets that exhibit $O(1)$ insertions.

Set intersection is also trivial to implement. The only major thing worth pointing out about our algorithm is that we traverse the set containing the fewest items. We can do this because if we have exhausted all the items in the smaller of the two sets then there are no more items that are members of both sets, thus we have no more items to add to the intersection set.

```
1) algorithm Intersection(set1, set2)
2)     Pre:  set1, and set2 ≠ ∅
3)           intersection, and smallerSet are sets
3)     Post: An intersection of set1, and set2 has been created
4)     if set1.Count < set2.Count
5)       smallerSet ← set1
6)     else
7)       smallerSet ← set2
8)     end if
9)     foreach item in smallerSet
10)      if set1.Contains(item) and set2.Contains(item)
11)        intersection.Add(item)
12)      end if
13)    end foreach
14)    return intersection
15) end Intersection
```

The run time of our *Intersection* algorithm is $O(n)$ where n is the number of items in the smaller of the two sets. Just like our *Union* algorithm a linear runtime can only be attained when operating on a set with $O(1)$ insertion.

5.1 Unordered

Sets in the general sense do not enforce the explicit ordering of their members. For example the members of $B = \{6, 2, 9\}$ conform to no ordering scheme because it is not required.

Most libraries provide implementations of unordered sets and so DSA does not; we simply mention it here to disambiguate between an unordered set and ordered set.

We will only look at insertion for an unordered set and cover briefly why a hash table is an efficient data structure to use for its implementation.

5.1.1 Insertion

An unordered set can be efficiently implemented using a hash table as its backing data structure. As mentioned previously we only add an item to a set if that item is not already in the set, so the backing data structure we use must have a quick look up and insertion run time complexity.

A hash map generally provides the following:

1. $O(1)$ for insertion
2. approaching $O(1)$ for look up

The above depends on how good the hashing algorithm of the hash table is, but most hash tables employ incredibly efficient general purpose hashing algorithms and so the run time complexities for the hash table in your library of choice should be very similar in terms of efficiency.

5.2 Ordered

An ordered set is similar to an unordered set in the sense that its members are distinct, but an ordered set enforces some predefined comparison on each of its members to produce a set whose members are ordered appropriately.

In DSA 0.5 and earlier we used a binary search tree (defined in §3) as the internal backing data structure for our ordered set. From versions 0.6 onwards we replaced the binary search tree with an AVL tree primarily because AVL is balanced.

The ordered set has its order realised by performing an inorder traversal upon its backing tree data structure which yields the correct ordered sequence of set members.

Because an ordered set in DSA is simply a wrapper for an AVL tree that additionally ensures that the tree contains unique items you should read §7 to learn more about the run time complexities associated with its operations.

5.3 Summary

Sets provide a way of having a collection of unique objects, either ordered or unordered.

When implementing a set (either ordered or unordered) it is key to select the correct backing data structure. As we discussed in §5.1.1 because we check first if the item is already contained within the set before adding it we need this check to be as quick as possible. For unordered sets we can rely on the use of a hash table and use the key of an item to determine whether or not it is already contained within the set. Using a hash table this check results in a near constant run time complexity. Ordered sets cost a little more for this check, however the logarithmic growth that we incur by using a binary search tree as its backing data structure is acceptable.

Another key property of sets implemented using the approach we describe is that both have favourably fast look-up times. Just like the check before insertion, for a hash table this run time complexity should be near constant. Ordered sets as described in 3 perform a binary chop at each stage when searching for the existence of an item yielding a logarithmic run time.

We can use sets to facilitate many algorithms that would otherwise be a little less clear in their implementation. For example in §11.4 we use an unordered set to assist in the construction of an algorithm that determines the number of repeated words within a string.

Chapter 6

Queues

Queues are an essential data structure that are found in vast amounts of software from user mode to kernel mode applications that are core to the system. Fundamentally they honour a first in first out (FIFO) strategy, that is the item first put into the queue will be the first served, the second item added to the queue will be the second to be served and so on.

A traditional queue only allows you to access the item at the front of the queue; when you add an item to the queue that item is placed at the back of the queue.

Historically queues always have the following three core methods:

Enqueue: places an item at the back of the queue;

Dequeue: retrieves the item at the front of the queue, and removes it from the queue;

Peek: [1] retrieves the item at the front of the queue without removing it from the queue

As an example to demonstrate the behaviour of a queue we will walk through a scenario whereby we invoke each of the previously mentioned methods observing the mutations upon the queue data structure. The following list describes the operations performed upon the queue in Figure 6.1:

1. Enqueue(10)
2. Enqueue(12)
3. Enqueue(9)
4. Enqueue(8)
5. Enqueue(3)
6. Dequeue()
7. Peek()

[1] This operation is sometimes referred to as Front

8. Enqueue(33)
9. Peek()
10. Dequeue()

6.1 A standard queue

A queue is implicitly like that described prior to this section. In DSA we don't provide a standard queue because queues are so popular and such a core data structure that you will find pretty much every mainstream library provides a queue data structure that you can use with your language of choice. In this section we will discuss how you can, if required, implement an efficient queue data structure.

The main property of a queue is that we have access to the item at the front of the queue. The queue data structure can be efficiently implemented using a singly linked list (defined in §2.1). A singly linked list provides $O(1)$ insertion and deletion run time complexities. The reason we have an $O(1)$ run time complexity for deletion is because we only ever remove items from the front of queues (with the Dequeue operation). Since we always have a pointer to the item at the head of a singly linked list, removal is simply a case of returning the value of the old head node, and then modifying the head pointer to be the next node of the old head node. The run time complexity for searching a queue remains the same as that of a singly linked list: $O(n)$.

6.2 Priority Queue

Unlike a standard queue where items are ordered in terms of who arrived first, a priority queue determines the order of its items by using a form of custom comparer to see which item has the highest priority. Other than the items in a priority queue being ordered by priority it remains the same as a normal queue: you can only access the item at the front of the queue.

A sensible implementation of a priority queue is to use a heap data structure (defined in §4). Using a heap we can look at the first item in the queue by simply returning the item at index 0 within the heap array. A heap provides us with the ability to construct a priority queue where the items with the highest priority are either those with the smallest value, or those with the largest.

6.3 Double Ended Queue

Unlike the queues we have talked about previously in this chapter a double ended queue allows you to access the items at both the front, and back of the queue. A double ended queue is commonly known as a deque which is the name we will here on in refer to it as.

A deque applies no prioritization strategy to its items like a priority queue does, items are added in order to either the front of back of the deque. The former properties of the deque are denoted by the programmer utilising the data structures exposed interface.

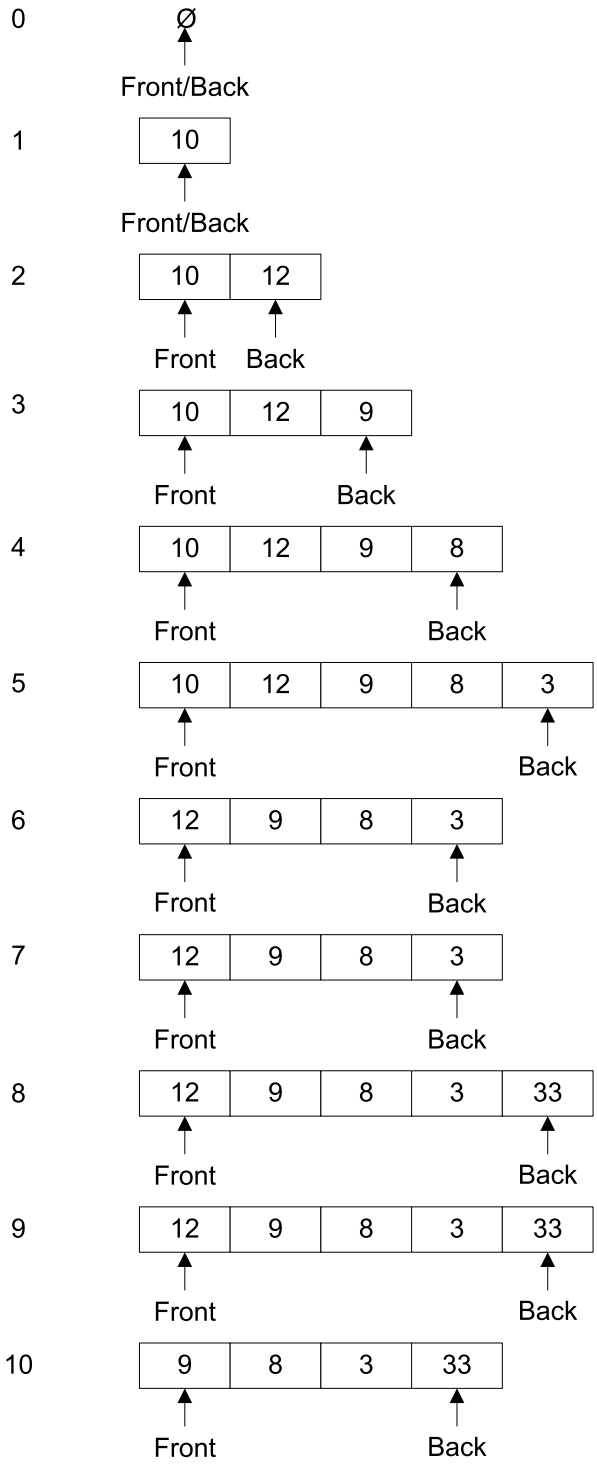


Figure 6.1: Queue mutations

Deque's provide front and back specific versions of common queue operations, e.g. you may want to enqueue an item to the front of the queue rather than the back in which case you would use a method with a name along the lines of *EnqueueFront*. The following list identifies operations that are commonly supported by deque's:

- EnqueueFront
- EnqueueBack
- DequeueFront
- DequeueBack
- PeekFront
- PeekBack

Figure 6.2 shows a deque after the invocation of the following methods (in-order):

1. EnqueueBack(12)
2. EnqueueFront(1)
3. EnqueueBack(23)
4. EnqueueFront(908)
5. DequeueFront()
6. DequeueBack()

The operations have a one-to-one translation in terms of behaviour with those of a normal queue, or priority queue. In some cases the set of algorithms that add an item to the back of the deque may be named as they are with normal queues, e.g. *EnqueueBack* may simply be called *Enqueue* an so on. Some frameworks also specify explicit behaviour's that data structures must adhere to. This is certainly the case in .NET where most collections implement an interface which requires the data structure to expose a standard *Add* method. In such a scenario you can safely assume that the *Add* method will simply enqueue an item to the back of the deque.

With respect to algorithmic run time complexities a deque is the same as a normal queue. That is enqueueing an item to the back of a the queue is $O(1)$, additionally enqueuing an item to the front of the queue is also an $O(1)$ operation.

A deque is a wrapper data structure that uses either an array, or a doubly linked list. Using an array as the backing data structure would require the programmer to be explicit about the size of the array up front, this would provide an obvious advantage if the programmer could deterministically state the maximum number of items the deque would contain at any one time. Unfortunately in most cases this doesn't hold, as a result the backing array will inherently incur the expense of invoking a resizing algorithm which would most likely be an $O(n)$ operation. Such an approach would also leave the library developer

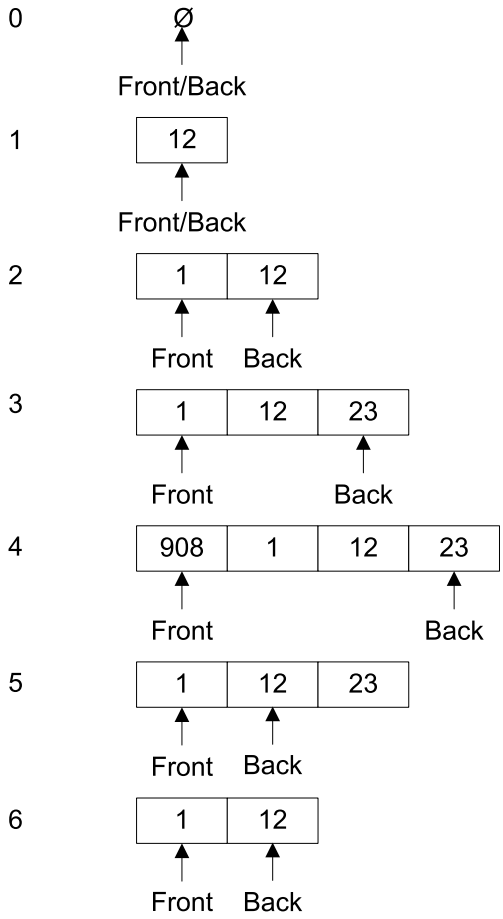


Figure 6.2: Deque data structure after several mutations

to look at array minimization techniques as well, it could be that after several invocations of the resizing algorithm and various mutations on the deque later that we have an array taking up a considerable amount of memory yet we are only using a few small percentage of that memory. An algorithm described would also be $O(n)$ yet its invocation would be harder to gauge strategically.

To bypass all the aforementioned issues a deque typically uses a doubly linked list as its baking data structure. While a node that has two pointers consumes more memory than its array item counterpart it makes redundant the need for expensive resizing algorithms as the data structure increases in size dynamically. With a language that targets a garbage collected virtual machine memory reclamation is an opaque process as the nodes that are no longer referenced become unreachable and are thus marked for collection upon the next invocation of the garbage collection algorithm. With C++ or any other language that uses explicit memory allocation and deallocation it will be up to the programmer to decide when the memory that stores the object can be freed.

6.4 Summary

With normal queues we have seen that those who arrive first are dealt with first; that is they are dealt with in a first-in-first-out (FIFO) order. Queues can be ever so useful; for example the Windows CPU scheduler uses a different queue for each priority of process to determine which should be the next process to utilise the CPU for a specified time quantum. Normal queues have constant insertion and deletion run times. Searching a queue is fairly unusual—typically you are only interested in the item at the front of the queue. Despite that, searching is usually exposed on queues and typically the run time is linear.

In this chapter we have also seen priority queues where those at the front of the queue have the highest priority and those near the back have the lowest. One implementation of a priority queue is to use a heap data structure as its backing store, so the run times for insertion, deletion, and searching are the same as those for a heap (defined in §4).

Queues are a very natural data structure, and while they are fairly primitive they can make many problems a lot simpler. For example the breadth first search defined in §3.7.4 makes extensive use of queues.

Chapter 7

AVL Tree

In the early 60's G.M. Adelson-Velsky and E.M. Landis invented the first self-balancing binary search tree data structure, calling it AVL Tree.

An AVL tree is a binary search tree (BST, defined in §3) with a self-balancing condition stating that the difference between the height of the left and right subtrees cannot be no more than one, see Figure 7.1. This condition, restored after each tree modification, forces the general shape of an AVL tree. Before continuing, let us focus on why balance is so important. Consider a binary search tree obtained by starting with an empty tree and inserting some values in the following order 1,2,3,4,5.

The BST in Figure 7.2 represents the worst case scenario in which the running time of all common operations such as search, insertion and deletion are $O(n)$. By applying a balance condition we ensure that the worst case running time of each common operation is $O(log\ n)$. The height of an AVL tree with n nodes is $O(log\ n)$ regardless of the order in which values are inserted.

The AVL balance condition, known also as the node balance factor represents an additional piece of information stored for each node. This is combined with a technique that efficiently restores the balance condition for the tree. In an AVL tree the inventors make use of a well-known technique called tree rotation.

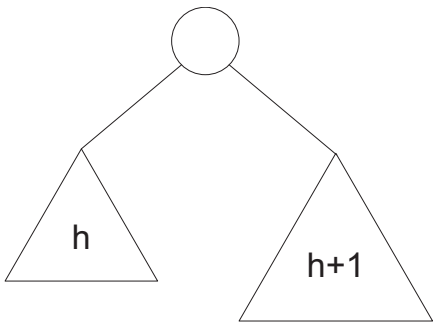


Figure 7.1: The left and right subtrees of an AVL tree differ in height by at most 1

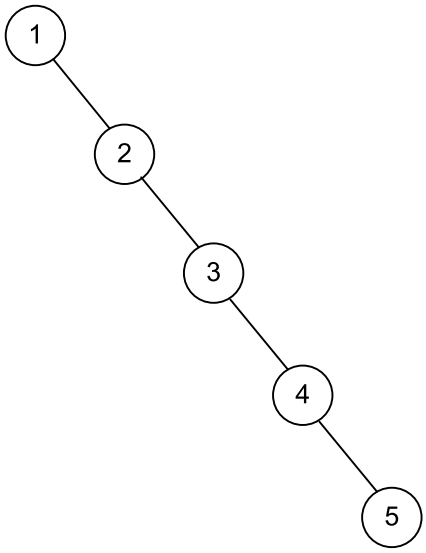


Figure 7.2: Unbalanced binary search tree

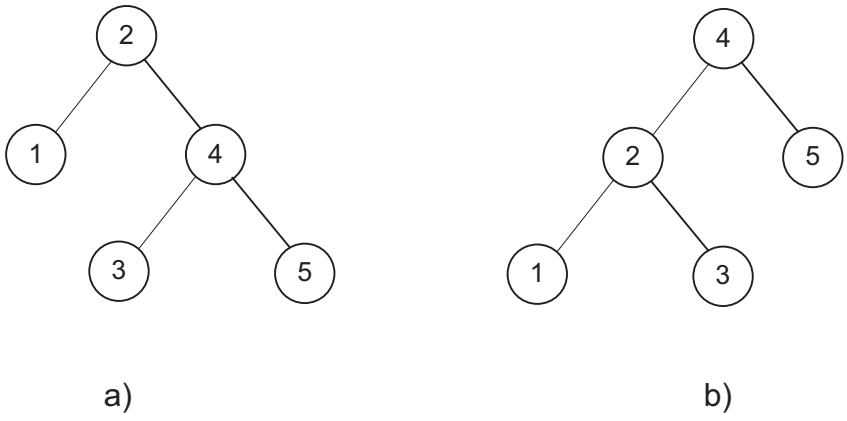


Figure 7.3: Avl trees, insertion order: -a)1,2,3,4,5 -b)1,5,4,3,2

7.1 Tree Rotations

A tree rotation is a constant time operation on a binary search tree that changes the shape of a tree while preserving standard BST properties. There are left and right rotations both of them decrease the height of a BST by moving smaller subtrees down and larger subtrees up.

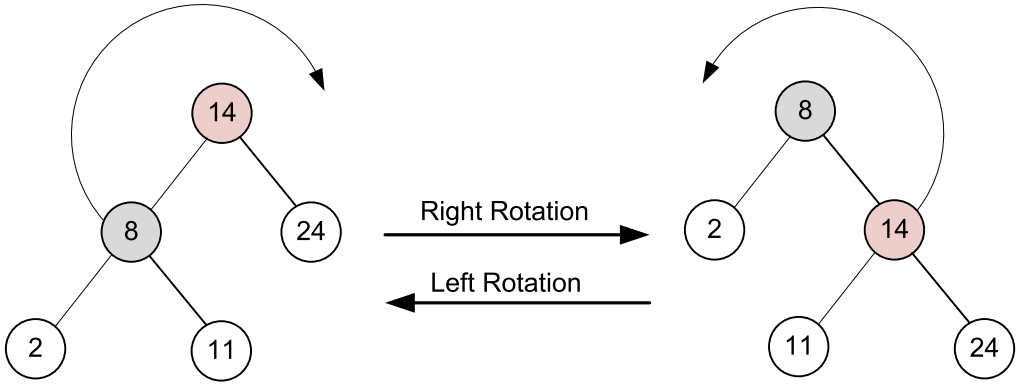


Figure 7.4: Tree left and right rotations

```
1) algorithm LeftRotation(node)
2)     Pre:  node.Right ! = ∅
3)     Post: node.Right is the new root of the subtree,
4)           node has become node.Right's left child and,
5)           BST properties are preserved
6)     RightNode ← node.Right
7)     node.Right ← RightNode.Left
8)     RightNode.Left ← node
9) end LeftRotation
```

```
1) algorithm RightRotation(node)
2)     Pre:  node.Left ! = ∅
3)     Post: node.Left is the new root of the subtree,
4)           node has become node.Left's right child and,
5)           BST properties are preserved
6)     LeftNode ← node.Left
7)     node.Left ← LeftNode.Right
8)     LeftNode.Right ← node
9) end RightRotation
```

The right and left rotation algorithms are symmetric. Only pointers are changed by a rotation resulting in an $O(1)$ runtime complexity; the other fields present in the nodes are not changed.

7.2 Tree Rebalancing

The algorithm that we present in this section verifies that the left and right subtrees differ at most in height by 1. If this property is not present then we perform the correct rotation.

Notice that we use two new algorithms that represent double rotations. These algorithms are named *LeftAndRightRotation*, and *RightAndLeftRotation*. The algorithms are self documenting in their names, e.g. *LeftAndRightRotation* first performs a left rotation and then subsequently a right rotation.

```
1) algorithm CheckBalance(current)
2)     Pre: current is the node to start from balancing
3)     Post: current height has been updated while tree balance is if needed
4)           restored through rotations
5)     if current.Left = ∅ and current.Right = ∅
6)       current.Height = -1;
7)     else
8)       current.Height = Max(Height(current.Left),Height(current.Right)) + 1
9)     end if
10)    if Height(current.Left) - Height(current.Right) > 1
11)      if Height(current.Left.Left) - Height(current.Left.Right) > 0
12)        RightRotation(current)
13)      else
14)        LeftAndRightRotation(current)
15)      end if
16)    else if Height(current.Left) - Height(current.Right) < −1
17)          if Height(current.Right.Left) - Height(current.Right.Right) < 0
18)            LeftRotation(current)
19)          else
20)            RightAndLeftRotation(current)
21)          end if
22)    end if
23) end CheckBalance
```

7.3 Insertion

AVL insertion operates first by inserting the given value the same way as BST insertion and then by applying rebalancing techniques if necessary. The latter is only performed if the AVL property no longer holds, that is the left and right subtrees height differ by more than 1. Each time we insert a node into an AVL tree:

1. We go down the tree to find the correct point at which to insert the node, in the same manner as for BST insertion; then

2. we travel up the tree from the inserted node and check that the node balancing property has not been violated; if the property hasn't been violated then we need not rebalance the tree, the opposite is true if the balancing property has been violated.

```
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 while
4)             preserving tree balance
5)     if value < current.Value
6)         if current.Left = ∅
7)             current.Left ← node(value)
8)         else
9)             InsertNode(current.Left, value)
10)        end if
11)    else
12)        if current.Right = ∅
13)            current.Right ← node(value)
14)        else
15)            InsertNode(current.Right, value)
16)        end if
17)    end if
18)    CheckBalance(current)
19) end InsertNode
```

7.4 Deletion

Our balancing algorithm is like the one presented for our BST (defined in §3.3). The major difference is that we have to ensure that the tree still adheres to the AVL balance property after the removal of the node. If the tree doesn't need to be rebalanced and the value we are removing is contained within the tree then no further step are required. However, when the value is in the tree and its removal upsets the AVL balance property then we must perform the correct rotation(s).

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

```
51)       end while
52)       // set the parents' Right pointer of largestValue to ∅
53)       FindParent(largestValue.Value).Right ← ∅
54)       nodeToRemove.Value ← largestValue.Value
55)    end if
56)    while path.Count > 0
57)       CheckBalance(path.Pop()) // we trackback to the root node check balance
58)    end while
59)    count ← count − 1
60)    return true
61) end Remove
```

7.5 Summary

The AVL tree is a sophisticated self balancing tree. It can be thought of as the smarter, younger brother of the binary search tree. Unlike its older brother the AVL tree avoids worst case linear complexity runtimes for its operations. The AVL tree guarantees via the enforcement of balancing algorithms that the left and right subtrees differ in height by at most 1 which yields at most a logarithmic runtime complexity.

Part II

Algorithms

Chapter 8

Sorting

All the sorting algorithms in this chapter use data structures of a specific type to demonstrate sorting, e.g. a 32 bit integer is often used as its associated operations (e.g. $<$, $>$, etc) are clear in their behaviour.

The algorithms discussed can easily be translated into generic sorting algorithms within your respective language of choice.

8.1 Bubble Sort

One of the most simple forms of sorting is that of comparing each item with every other item in some list, however as the description may imply this form of sorting is not particularly effecient $O(n^2)$. In it's most simple form bubble sort can be implemented as two loops.

```
1) algorithm BubbleSort(list)
2)     Pre: list ≠ ∅
3)     Post: list has been sorted into values of ascending order
4)     for i ← 0 to list.Count − 1
5)         for j ← 0 to list.Count − 1
6)             if list[i] < list[j]
7)                 Swap(list[i], list[j])
8)             end if
9)         end for
10)    end for
11)    return list
12) end BubbleSort
```

8.2 Merge Sort

Merge sort is an algorithm that has a fairly efficient space time complexity - $O(n \log n)$ and is fairly trivial to implement. The algorithm is based on splitting a list, into two similar sized lists ($left$, and $right$) and sorting each list and then merging the sorted lists back together.

Note: the function MergeOrdered simply takes two ordered lists and makes them one.

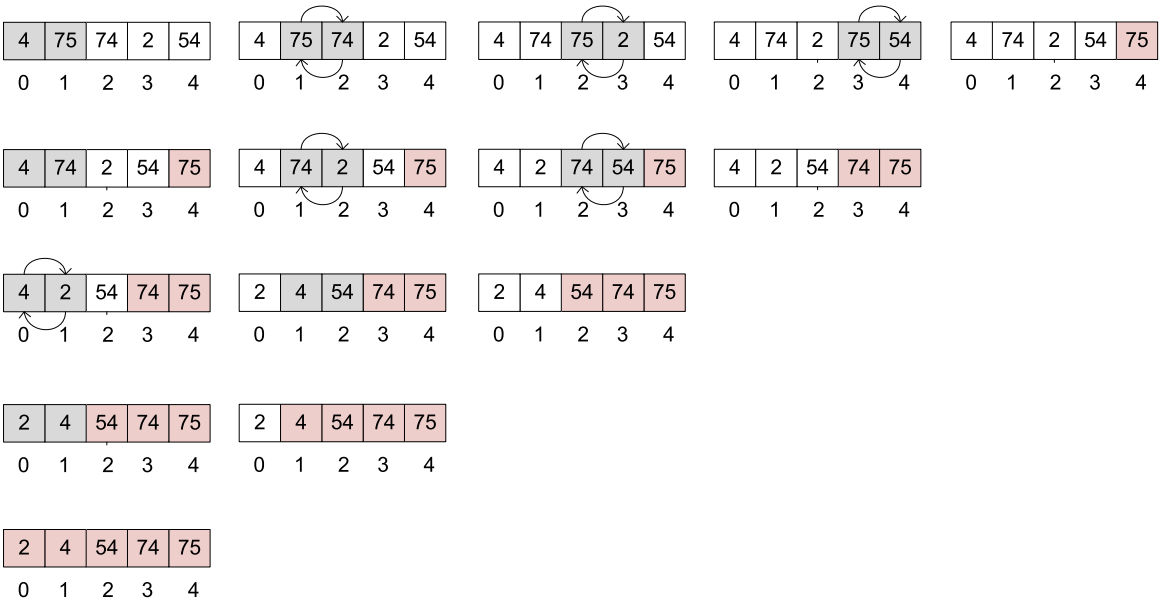


Figure 8.1: Bubble Sort Iterations

1) **algorithm** Mergesort(*list*)
2) **Pre:** $list \neq \emptyset$
3) **Post:** *list* has been sorted into values of ascending order
4) **if** *list*.Count = 1 // already sorted
5) **return** *list*
6) **end if**
7) $m \leftarrow list$.Count / 2
8) $left \leftarrow$ list(m)
9) $right \leftarrow$ list($list$.Count $- m$)
10) **for** $i \leftarrow 0$ to $left$.Count-1
11) $left[i] \leftarrow$ list$[i]$
12) **end for**
13) **for** $i \leftarrow 0$ to $right$.Count-1
14) $right[i] \leftarrow$ list$[i]$
15) **end for**
16) $left \leftarrow$ Mergesort($left$)
17) $right \leftarrow$ Mergesort($right$)
18) **return** MergeOrdered($left$, $right$)
19) **end** Mergesort

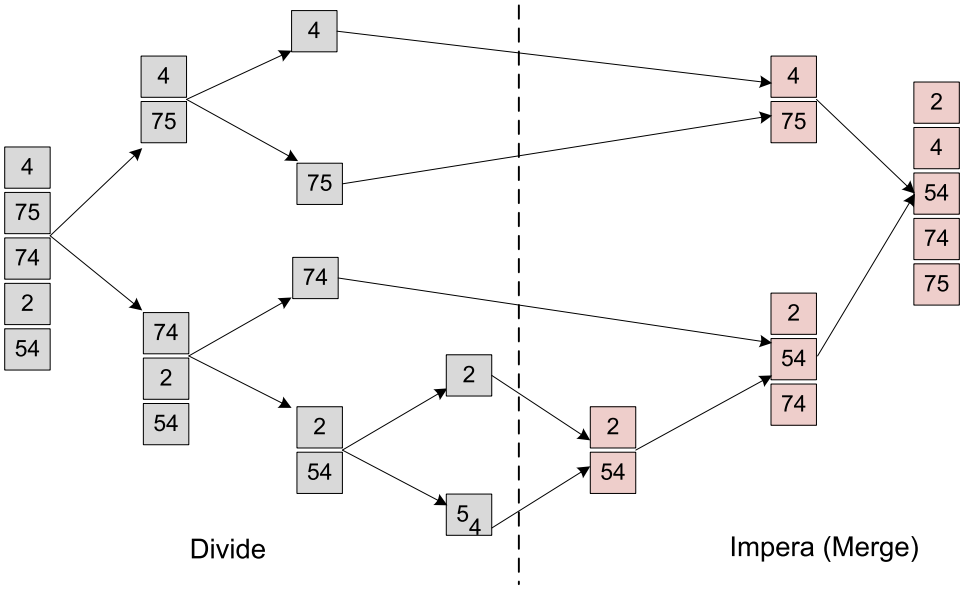


Figure 8.2: Merge Sort Divide et Impera Approach

8.3 Quick Sort

Quick sort is one of the most popular sorting algorithms based on divide et impera strategy, resulting in an $O(n\ log\ n)$ complexity. The algorithm starts by picking an item, called pivot, and moving all smaller items before it, while all greater elements after it. This is the main quick sort operation, called partition, recursively repeated on lesser and greater sub lists until their size is one or zero - in which case the list is implicitly sorted.

Choosing an appropriate pivot, as for example the median element is fundamental for avoiding the drastically reduced performance of $O(n^2)$.

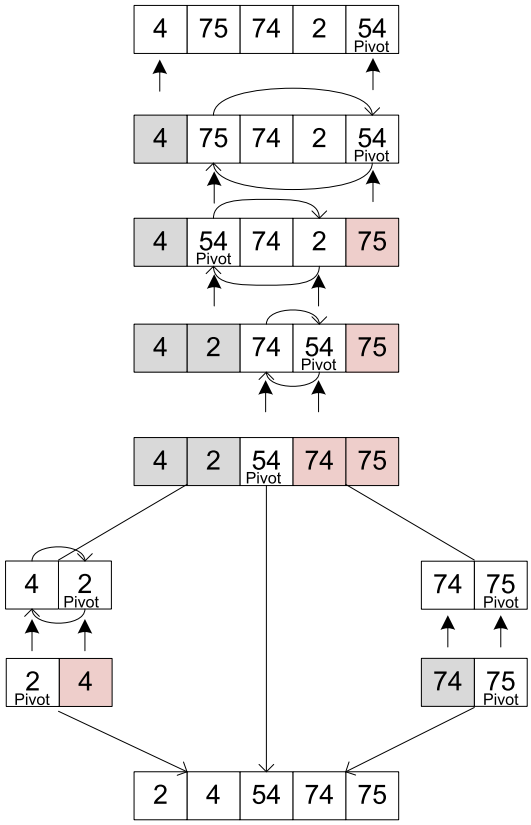


Figure 8.3: Quick Sort Example (pivot median strategy)

```
1) algorithm QuickSort(list)
2)     Pre: list ≠ ∅
3)     Post: list has been sorted into values of ascending order
4)     if list.Count = 1 // already sorted
5)        return list
6)     end if
7)     pivot ←MedianValue(list)
8)     for i ← 0 to list.Count−1
9)        if list[i] = pivot
10)          equal.Insert(list[i])
11)       end if
12)       if list[i] < pivot
13)          less.Insert(list[i])
14)       end if
15)       if list[i] > pivot
16)          greater.Insert(list[i])
17)       end if
18)    end for
19)    return Concatenate(QuickSort(less), equal, QuickSort(greater))
20) end Quicksort
```

8.4 Insertion Sort

Insertion sort is a somewhat interesting algorithm with an expensive runtime of $O(n^2)$. It can be best thought of as a sorting scheme similar to that of sorting a hand of playing cards, i.e. you take one card and then look at the rest with the intent of building up an ordered set of cards in your hand.

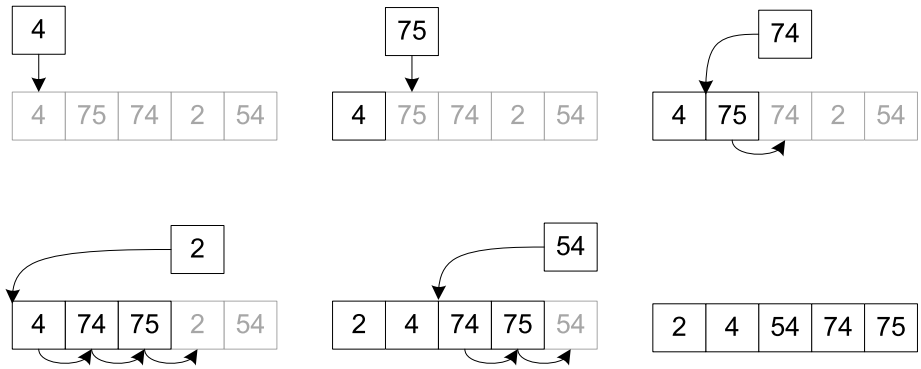


Figure 8.4: Insertion Sort Iterations

```
1) algorithm Insertionsort(list)
2)     Pre:  list ≠ ∅
3)     Post: list has been sorted into values of ascending order
4)     unsorted ← 1
5)     while unsorted < list.Count
6)         hold ← list[unsorted]
7)         i ← unsorted − 1
8)         while i ≥ 0 and hold < list[i]
9)             list[i + 1] ← list[i]
10)            i ← i − 1
11)        end while
12)        list[i + 1] ← hold
13)        unsorted ← unsorted + 1
14)    end while
15)    return list
16) end Insertionsort
```

8.5 Shell Sort

Put simply shell sort can be thought of as a more efficient variation of insertion sort as described in §8.4, it achieves this mainly by comparing items of varying distances apart resulting in a run time complexity of $O(n\ log^2\ n)$.

Shell sort is fairly straight forward but may seem somewhat confusing at first as it differs from other sorting algorithms in the way it selects items to compare. Figure 8.5 shows shell sort being ran on an array of integers, the red coloured square is the current value we are holding.

```
1) algorithm ShellSort(list)
2)     Pre: list ≠ ∅
3)     Post: list has been sorted into values of ascending order
4)     increment ← list.Count / 2
5)     while increment ≠ 0
6)         current ← increment
7)         while current < list.Count
8)             hold ← list[current]
9)             i ← current − increment
10)            while i ≥ 0 and hold < list[i]
11)                list[i + increment] ← list[i]
12)                i− = increment
13)            end while
14)            list[i + increment] ← hold
15)            current ← current + 1
16)        end while
17)        increment / = 2
18)    end while
19)    return list
20) end ShellSort
```

8.6 Radix Sort

Unlike the sorting algorithms described previously radix sort uses buckets to sort items, each bucket holds items with a particular property called a key. Normally a bucket is a queue, each time radix sort is performed these buckets are emptied starting the smallest key bucket to the largest. When looking at items within a list to sort we do so by isolating a specific key, e.g. in the example we are about to show we have a maximum of three keys for all items, that is the highest key we need to look at is hundreds. Because we are dealing with, in this example base 10 numbers we have at any one point 10 possible key values 0..9 each of which has their own bucket. Before we show you this first simple version of radix sort let us clarify what we mean by isolating keys. Given the number 102 if we look at the first key, the ones then we can see we have two of them, progressing to the next key - tens we can see that the number has zero of them, finally we can see that the number has a single hundred. The number used as an example has in total three keys:

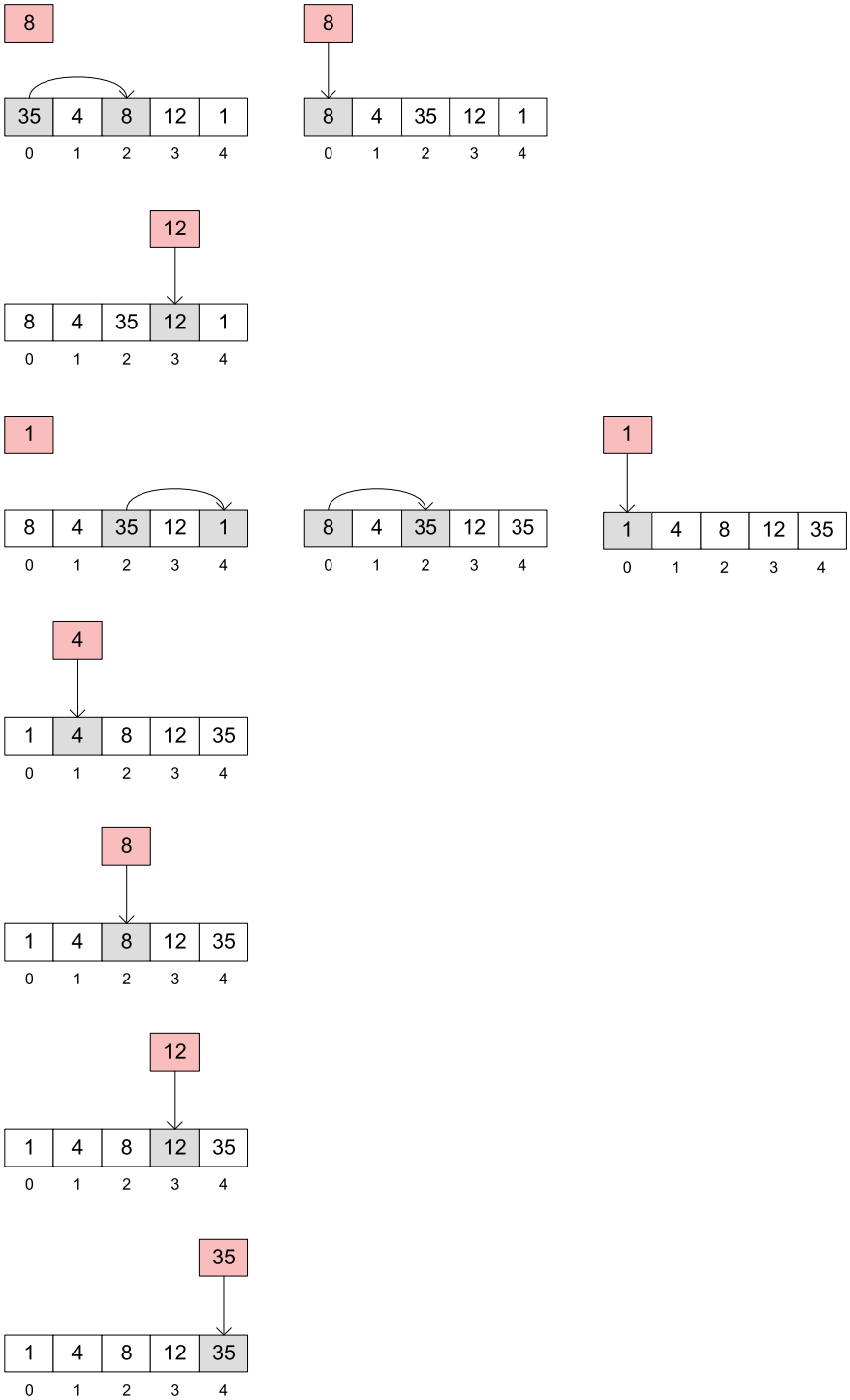


Figure 8.5: Shell sort

1. Ones
2. Tens
3. Hundreds

For further clarification what if we wanted to determine how many thousands the number 102 has? Clearly there are none, but often looking at a number as final like we often do it is not so obvious so when asked the question how many thousands does 102 have you should simply pad the number with a zero in that location, e.g. 0102 here it is more obvious that the key value at the thousands location is zero.

The last thing to identify before we actually show you a simple implementation of radix sort that works on only positive integers, and requires you to specify the maximum key size in the list is that we need a way to isolate a specific key at any one time. The solution is actually very simple, but its not often you want to isolate a key in a number so we will spell it out clearly here. A key can be accessed from any integer with the following expression: $key \leftarrow (number \ / \ keyToAccess) \ \% \ 10$. As a simple example lets say that we want to access the tens key of the number 1290, the tens column is key 10 and so after substitution yields $key \leftarrow (1290 \ / \ 10) \ \% \ 10 \ = \ 9$. The next key to look at for a number can be attained by multiplying the last key by ten working left to right in a sequential manner. The value of *key* is used in the following algorithm to work out the index of an array of queues to enqueue the item into.

```
1) algorithm Radix(list, maxKeySize)
2)     Pre: list ≠ ∅
3)          maxKeySize ≥ 0 and represents the largest key size in the list
4)     Post: list has been sorted
5)     queues ← Queue[10]
6)     indexOfKey ← 1
7)     fori ← 0 to maxKeySize − 1
8)         foreach item in list
9)             queues[GetQueueIndex(item, indexOfKey)].Enqueue(item)
10)        end foreach
11)        list ← CollapseQueues(queues)
12)        ClearQueues(queues)
13)        indexOfKey ← indexOfKey ∗ 10
14)    end for
15)    return list
16) end Radix
```

Figure 8.6 shows the members of *queues* from the algorithm described above operating on the list whose members are $90, 12, 8, 791, 123$, and 61, the key we are interested in for each number is highlighted. Omitted queues in Figure 8.6 mean that they contain no items.

8.7 Summary

Throughout this chapter we have seen many different algorithms for sorting lists, some are very efficient (e.g. quick sort defined in §8.3), some are not (e.g.

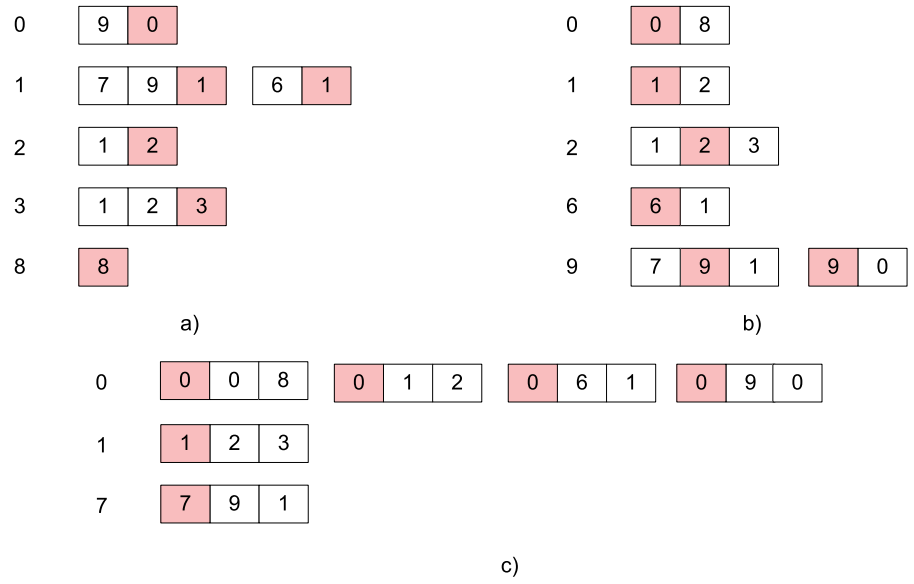


Figure 8.6: Radix sort base 10 algorithm

bubble sort defined in §8.1).

Selecting the correct sorting algorithm is usually denoted purely by efficiency, e.g. you would always choose merge sort over shell sort and so on. There are also other factors to look at though and these are based on the actual implementation. Some algorithms are very nicely expressed in a recursive fashion, however these algorithms ought to be pretty efficient, e.g. implementing a linear, quadratic, or slower algorithm using recursion would be a very bad idea.

If you want to learn more about why you should be very, very careful when implementing recursive algorithms see Appendix C.

Chapter 9

Numeric

Unless stated otherwise the alias n denotes a standard 32 bit integer.

9.1 Primality Test

A simple algorithm that determines whether or not a given integer is a prime number, e.g. 2, 5, 7, and 13 are **all** prime numbers, however 6 is not as it can be the result of the product of two numbers that are < 6.

In an attempt to slow down the inner loop the $\sqrt{n}$ is used as the upper bound.

```
1) algorithm IsPrime(n)
2)     Post: n is determined to be a prime or not
3)     for i ← 2 to n do
4)         for j ← 1 to sqrt(n) do
5)             if i * j = n
6)                 return false
7)             end if
8)         end for
9)     end for
10) end IsPrime
```

9.2 Base conversions

DSA contains a number of algorithms that convert a base 10 number to its equivalent binary, octal or hexadecimal form. For example 78_{10} has a binary representation of 1001110_2.

Table 9.1 shows the algorithm trace when the number to convert to binary is 742_{10}.

```
1) algorithm ToBinary(n)
2)     Pre:  n ≥ 0
3)     Post: n has been converted into its base 2 representation
4)     while n > 0
5)         list.Add(n % 2)
6)         n ← n/2
7)     end while
8)     return Reverse(list)
9) end ToBinary
```

n	$list$
742	{ 0 }
371	{ 0, 1 }
185	{ 0, 1, 1 }
92	{ 0, 1, 1, 0 }
46	{ 0, 1, 1, 0, 1 }
23	{ 0, 1, 1, 0, 1, 1 }
11	{ 0, 1, 1, 0, 1, 1, 1 }
5	{ 0, 1, 1, 0, 1, 1, 1, 1 }
2	{ 0, 1, 1, 0, 1, 1, 1, 1, 0 }
1	{ 0, 1, 1, 0, 1, 1, 1, 1, 0, 1 }

Table 9.1: Algorithm trace of ToBinary

9.3 Attaining the greatest common denominator of two numbers

A fairly routine problem in mathematics is that of finding the greatest common denominator of two integers, what we are essentially after is the greatest number which is a multiple of both, e.g. the greatest common denominator of 9, and 15 is 3. One of the most elegant solutions to this problem is based on Euclid's algorithm that has a run time complexity of $O(n^2)$.

```
1) algorithm GreatestCommonDenominator(m, n)
2)     Pre:  m and n are integers
3)     Post: the greatest common denominator of the two integers is calculated
4)     if n = 0
5)        return m
6)     end if
7)     return GreatestCommonDenominator(n, m % n)
8) end GreatestCommonDenominator
```

9.4 Computing the maximum value for a number of a specific base consisting of N digits

This algorithm computes the maximum value of a number for a given number of digits, e.g. using the base 10 system the maximum number we can have made up of 4 digits is the number 9999_{10}. Similarly the maximum number that consists of 4 digits for a base 2 number is 1111_2 which is 15_{10}.

The expression by which we can compute this maximum value for N digits is: $B^N - 1$. In the previous expression B is the number base, and N is the number of digits. As an example if we wanted to determine the maximum value for a hexadecimal number (base 16) consisting of 6 digits the expression would be as follows: $16^6 - 1$. The maximum value of the previous example would be represented as $FFFFFF_{16}$ which yields 16777215_{10}.

In the following algorithm *numberBase* should be considered restricted to the values of 2, 8, 9, and 16. For this reason in our actual implementation *numberBase* has an enumeration type. The *Base* enumeration type is defined as:

$$Base = \{Binary \leftarrow 2, Octal \leftarrow 8, Decimal \leftarrow 10, Hexadecimal \leftarrow 16\}$$

The reason we provide the definition of *Base* is to give you an idea how this algorithm can be modelled in a more readable manner rather than using various checks to determine the correct base to use. For our implementation we cast the value of *numberBase* to an integer, as such we extract the value associated with the relevant option in the *Base* enumeration. As an example if we were to cast the option *Octal* to an integer we would get the value 8. In the algorithm listed below the cast is implicit so we just use the actual argument *numberBase*.

1) **algorithm** MaxValue(*numberBase*, n)
2) **Pre:** *numberBase* is the number system to use, n is the number of digits
3) **Post:** the maximum value for *numberBase* consisting of n digits is computed
4) **return** Power(*numberBase*, n) -1
5) **end** MaxValue

9.5 Factorial of a number

Attaining the factorial of a number is a primitive mathematical operation. Many implementations of the factorial algorithm are recursive as the problem is recursive in nature, however here we present an iterative solution. The iterative solution is presented because it too is trivial to implement and doesn't suffer from the use of recursion (for more on recursion see §C).

The factorial of 0 and 1 is 0. The aforementioned acts as a base case that we will build upon. The factorial of 2 is 2∗ the factorial of 1, similarly the factorial of 3 is 3∗ the factorial of 2 and so on. We can indicate that we are after the factorial of a number using the form $N!$ where N is the number we wish to attain the factorial of. Our algorithm doesn't use such notation but it is handy to know.

```
1) algorithm Factorial(n)
2)     Pre:  n ≥ 0, n is the number to compute the factorial of
3)     Post: the factorial of n is computed
4)     if n < 2
5)         return 1
6)     end if
7)     factorial ← 1
8)     for i ← 2 to n
9)         factorial ← factorial * i
10)    end for
11)    return factorial
12) end Factorial
```

9.6 Summary

In this chapter we have presented several numeric algorithms, most of which are simply here because they were fun to design. Perhaps the message that the reader should gain from this chapter is that algorithms can be applied to several domains to make work in that respective domain attainable. Numeric algorithms in particular drive some of the most advanced systems on the planet computing such data as weather forecasts.

Chapter 10

Searching

10.1 Sequential Search

A simple algorithm that search for a specific item inside a list. It operates looping on each element $O(n)$ until a match occurs or the end is reached.

```
1) algorithm SequentialSearch(list, item)
2)     Pre:  list ≠ ∅
3)     Post: return index of item if found, otherwise −1
4)     index ← 0
5)     while index < list.Count and list[index] ≠ item
6)         index ← index + 1
7)     end while
8)     if index < list.Count and list[index] = item
9)        return index
10)    end if
11)    return −1
12) end SequentialSearch
```

10.2 Probability Search

Probability search is a statistical sequential searching algorithm. In addition to searching for an item, it takes into account its frequency by swapping it with it's predecessor in the list. The algorithm complexity still remains at $O(n)$ but in a non-uniform items search the more frequent items are in the first positions, reducing list scanning time.

Figure 10.1 shows the resulting state of a list after searching for two items, notice how the searched items have had their search probability increased after each search operation respectively.

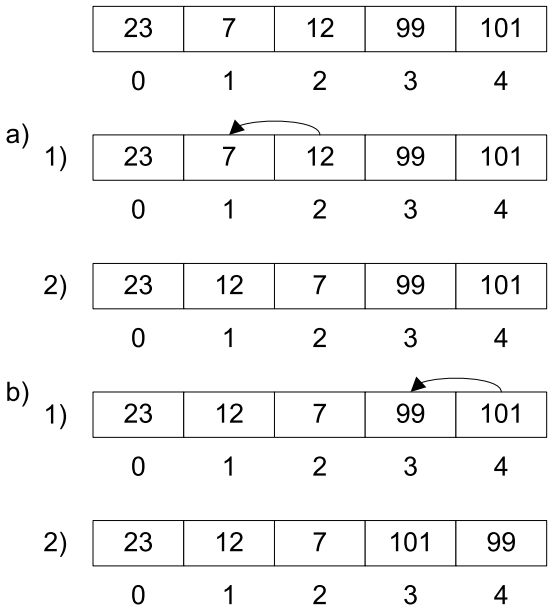


Figure 10.1: a) Search(12), b) Search(101)

```
1) algorithm ProbabilitySearch(list, item)
2)     Pre:  list ≠ ∅
3)     Post: a boolean indicating where the item is found or not;
                 in the former case swap founded item with its predecessor
4)     index ← 0
5)     while index < list.Count and list[index] ≠ item
6)         index ← index + 1
7)     end while
8)     if index ≥ list.Count or list[index] ≠ item
9)        return false
10)    end if
11)    if index > 0
12)       Swap(list[index], list[index − 1])
13)    end if
14)    return true
15) end ProbabilitySearch
```

10.3 Summary

In this chapter we have presented a few novel searching algorithms. We have presented more efficient searching algorithms earlier on, like for instance the logarithmic searching algorithm that AVL and BST tree's use (defined in §3.2). We decided not to cover a searching algorithm known as binary chop (another name for binary search, binary chop usually refers to its array counterpart) as

the reader has already seen such an algorithm in §3.

Searching algorithms and their efficiency largely depends on the underlying data structure being used to store the data. For instance it is quicker to determine whether an item is in a hash table than it is an array, similarly it is quicker to search a BST than it is a linked list. If you are going to search for data fairly often then we strongly advise that you sit down and research the data structures available to you. In most cases using a list or any other primarily linear data structure is down to lack of knowledge. Model your data and then research the data structures that best fit your scenario.

Chapter 11

Strings

Strings have their own chapter in this text purely because string operations and transformations are incredibly frequent within programs. The algorithms presented are based on problems the authors have come across previously, or were formulated to satisfy curiosity.

11.1 Reversing the order of words in a sentence

Defining algorithms for primitive string operations is simple, e.g. extracting a sub-string of a string, however some algorithms that require more inventiveness can be a little more tricky.

The algorithm presented here does not simply reverse the characters in a string, rather it reverses the order of words within a string. This algorithm works on the principal that words are all delimited by white space, and using a few markers to define where words start and end we can easily reverse them.

```
1) algorithm ReverseWords(value)
2)     Pre:  value ≠ ∅, sb is a string buffer
3)     Post: the words in value have been reversed
4)     last ← value.Length − 1
5)     start ← last
6)     while last ≥ 0
7)         // skip whitespace
8)         while start ≥ 0 and value[start] = whitespace
9)             start ← start − 1
10)        end while
11)        last ← start
12)        // march down to the index before the beginning of the word
13)        while start ≥ 0 and start ≠ whitespace
14)            start ← start − 1
15)        end while
16)        // append chars from start + 1 to length + 1 to string buffer sb
17)        for i ← start + 1 to last
18)            sb.Append(value[i])
19)        end for
20)        // if this isn't the last word in the string add some whitespace after the word in the buffer
21)        if start > 0
22)            sb.Append(' ')
23)        end if
24)        last ← start − 1
25)        start ← last
26)    end while
27)    // check if we have added one too many whitespace to sb
28)    if sb[sb.Length −1] = whitespace
29)        // cut the whitespace
30)        sb.Length ← sb.Length −1
31)    end if
32)    return sb
33) end ReverseWords
```

11.2 Detecting a palindrome

Although not a frequent algorithm that will be applied in real-life scenarios detecting a palindrome is a fun, and as it turns out pretty trivial algorithm to design.

The algorithm that we present has a $O(n)$ run time complexity. Our algorithm uses two pointers at opposite ends of string we are checking is a palindrome or not. These pointers march in towards each other always checking that each character they point to is the same with respect to value. Figure 11.1 shows the *IsPalindrome* algorithm in operation on the string "Was it Eliot's toilet I saw?" If you remove all punctuation, and white space from the aforementioned string you will find that it is a valid palindrome.

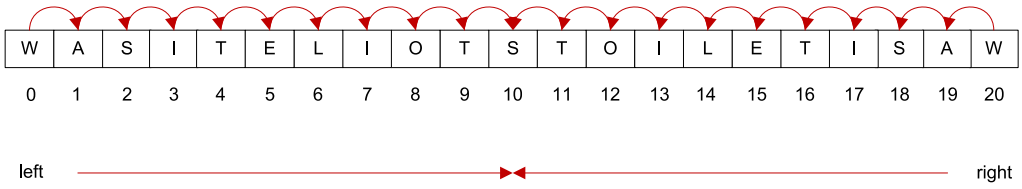


Figure 11.1: $left$ and $right$ pointers marching in towards one another

```
1) algorithm IsPalindrome(value)
2)     Pre:  value ≠ ∅
3)     Post:  value is determined to be a palindrome or not
4)     word ← value.Strip().ToUpperCase()
5)     left ← 0
6)     right ← word.Length −1
7)     while word[left] = word[right] and left < right
8)         left ← left + 1
9)         right ← right − 1
10)    end while
11)    return word[left] = word[right]
12) end IsPalindrome
```

In the *IsPalindrome* algorithm we call a method by the name of *Strip*. This algorithm discards punctuation in the string, including white space. As a result *word* contains a heavily compacted representation of the original string, each character of which is in its uppercase representation.

Palindromes discard white space, punctuation, and case making these changes allows us to design a simple algorithm while making our algorithm fairly robust with respect to the palindromes it will detect.

11.3 Counting the number of words in a string

Counting the number of words in a string can seem pretty trivial at first, however there are a few cases that we need to be aware of:

1. tracking when we are in a string
2. updating the word count at the correct place
3. skipping white space that delimits the words

As an example consider the string "Ben ate hay" Clearly this string contains three words, each of which distinguished via white space. All of the previously listed points can be managed by using three variables:

1. *index*
2. *wordCount*
3. *inWord*

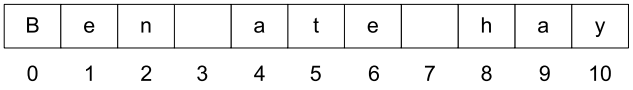


Figure 11.2: String with three words

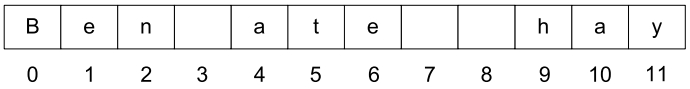


Figure 11.3: String with varying number of white space delimiting the words

Of the previously listed *index* keeps track of the current index we are at in the string, *wordCount* is an integer that keeps track of the number of words we have encountered, and finally $inWord$ is a Boolean flag that denotes whether or not at the present time we are within a word. If we are not currently hitting white space we are in a word, the opposite is true if at the present index we are hitting white space.

What denotes a word? In our algorithm each word is separated by one or more occurrences of white space. We don't take into account any particular splitting symbols you may use, e.g. in .NET *String.Split*[1] can take a char (or array of characters) that determines a delimiter to use to split the characters within the string into chunks of strings, resulting in an array of sub-strings.

In Figure 11.2 we present a string indexed as an array. Typically the pattern is the same for most words, delimited by a single occurrence of white space. Figure 11.3 shows the same string, with the same number of words but with varying white space splitting them.

[1] http://msdn.microsoft.com/en-us/library/system.string.split.aspx

```
1) algorithm WordCount(value)
2)     Pre:  value ≠ ∅
3)     Post: the number of words contained within value is determined
4)     inWord ← true
5)     wordCount ← 0
6)     index ← 0
7)     // skip initial white space
8)     while value[index] = whitespace and index < value.Length −1
9)         index ← index + 1
10)    end while
11)    // was the string just whitespace?
12)    if index = value.Length and value[index] = whitespace
13)        return 0
14)    end if
15)    while index < value.Length
16)        if value[index] = whitespace
17)            // skip all whitespace
18)            while value[index] = whitespace and index < value.Length −1
19)                index ← index + 1
20)            end while
21)            inWord ← false
22)            wordCount ← wordCount + 1
23)        else
24)            inWord ← true
25)        end if
26)        index ← index + 1
27)    end while
28)    // last word may have not been followed by whitespace
29)    if inWord
30)        wordCount ← wordCount + 1
31)    end if
32)    return wordCount
33) end WordCount
```

11.4 Determining the number of repeated words within a string

With the help of an unordered set, and an algorithm that can split the words within a string using a specified delimiter this algorithm is straightforward to implement. If we split all the words using a single occurrence of white space as our delimiter we get all the words within the string back as elements of an array. Then if we iterate through these words adding them to a set which contains only unique strings we can attain the number of unique words from the string. All that is left to do is subtract the unique word count from the total number of stings contained in the array returned from the split operation. The split operation that we refer to is the same as that mentioned in §11.3.

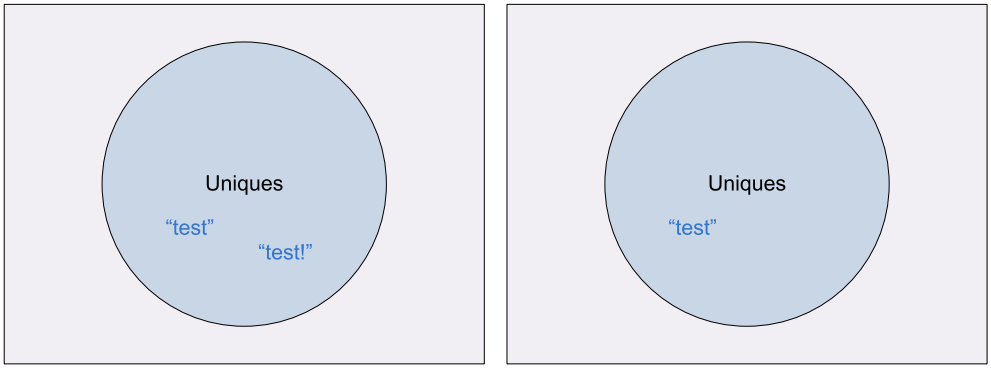


Figure 11.4: a) Undesired *uniques* set; b) desired *uniques* set

```
1) algorithm RepeatedWordCount(value)
2)     Pre:   value ≠ ∅
3)     Post:  the number of repeated words in value is returned
4)     words ← value.Split(' ')
5)     uniques ← Set
6)     foreach word in words
7)         uniques.Add(word.Strip())
8)     end foreach
9)     return words.Length −uniques.Count
10) end RepeatedWordCount
```

You will notice in the *RepeatedWordCount* algorithm that we use the *Strip* method we referred to earlier in §11.1. This simply removes any punctuation from a *word*. The reason we perform this operation on each *word* is so that we can build a more accurate unique string collection, e.g. “test”, and “test!” are the same word minus the punctuation. Figure 11.4 shows the undesired and desired sets for the *unique* set respectively.

11.5 Determining the first matching character between two strings

The algorithm to determine whether any character of a string matches any of the characters in another string is pretty trivial. Put simply, we can parse the strings considered using a double loop and check, discarding punctuation, the equality between any characters thus returning a non-negative index that represents the location of the first character in the match (Figure 11.5); otherwise we return -1 if no match occurs. This approach exhibit a run time complexity of $O(n^2)$.

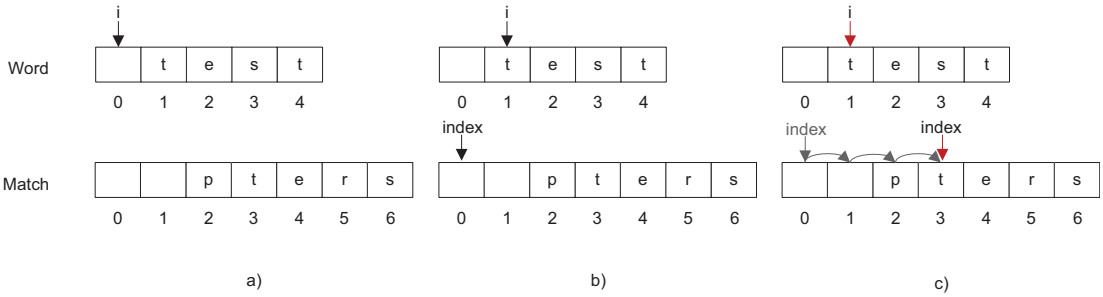


Figure 11.5: a) First Step; b) Second Step c) Match Occurred

```
1) algorithm Any(word,match)
2)     Pre:  word, match  ≠ ∅
3)     Post: index representing match location if occured, −1 otherwise
4)     for i ← 0 to word.Length − 1
5)         while word[i] = whitespace
6)                 i ← i + 1
7)         end while
8)         for index ← 0 to match.Length − 1
9)             while match[index] = whitespace
10)                    index ← index + 1
11)            end while
12)            if match[index] = word[i]
13)              return index
14)            end if
15)        end for
16)    end for
17)    return −1
18) end Any
```

11.6 Summary

We hope that the reader has seen how fun algorithms on string data types are. Strings are probably the most common data type (and data structure - remember we are dealing with an array) that you will work with so its important that you learn to be creative with them. We for one find strings fascinating. A simple Google search on string nuances between languages and encodings will provide you with a great number of problems. Now that we have spurred you along a little with our introductory algorithms you can devise some of your own.

Appendix A

Algorithm Walkthrough

Learning how to design good algorithms can be assisted greatly by using a structured approach to tracing its behaviour. In most cases tracing an algorithm only requires a single table. In most cases tracing is not enough, you will also want to use a diagram of the data structure your algorithm operates on. This diagram will be used to visualise the problem more effectively. Seeing things visually can help you understand the problem quicker, and better.

The trace table will store information about the variables used in your algorithm. The values within this table are constantly updated when the algorithm mutates them. Such an approach allows you to attain a history of the various values each variable has held. You may also be able to infer patterns from the values each variable has contained so that you can make your algorithm more efficient.

We have found this approach both simple, and powerful. By combining a visual representation of the problem as well as having a history of past values generated by the algorithm it can make understanding, and solving problems much easier.

In this chapter we will show you how to work through both iterative, and recursive algorithms using the technique outlined.

A.1 Iterative algorithms

We will trace the *IsPalindrome* algorithm (defined in §11.2) as our example iterative walkthrough. Before we even look at the variables the algorithm uses, first we will look at the actual data structure the algorithm operates on. It should be pretty obvious that we are operating on a string, but how is this represented? A string is essentially a block of contiguous memory that consists of some char data types, one after the other. Each character in the string can be accessed via an index much like you would do when accessing items within an array. The picture should be presenting itself - a string can be thought of as an array of characters.

For our example we will use *IsPalindrome* to operate on the string "Never odd or even" Now we know how the string data structure is represented, and the value of the string we will operate on let's go ahead and draw it as shown in Figure A.1.

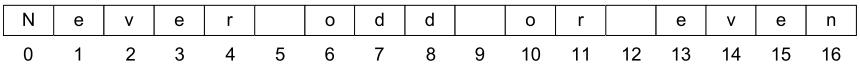


Figure A.1: Visualising the data structure we are operating on

value	*word*	*left*	*right*

Table A.1: A column for each variable we wish to track

The *IsPalindrome* algorithm uses the following list of variables in some form throughout its execution:

1. *value*
2. *word*
3. *left*
4. *right*

Having identified the values of the variables we need to keep track of we simply create a column for each in a table as shown in Table A.1.

Now, using the *IsPalindrome* algorithm execute each statement updating the variable values in the table appropriately. Table A.2 shows the final table values for each variable used in *IsPalindrome* respectively.

While this approach may look a little bloated in print, on paper it is much more compact. Where we have the strings in the table you should annotate these strings with array indexes to aid the algorithm walkthrough.

There is one other point that we should clarify at this time - whether to include variables that change only a few times, or not at all in the trace table. In Table A.2 we have included both the *value*, and *word* variables because it was convenient to do so. You may find that you want to promote these values to a larger diagram (like that in Figure A.1) and only use the trace table for variables whose values change during the algorithm. We recommend that you promote the core data structure being operated on to a larger diagram outside of the table so that you can interrogate it more easily.

value	*word*	*left*	*right*
“Never odd or even”	“NEVERODDOREVEN”	0	13
		1	12
		2	11
		3	10
		4	9
		5	8
		6	7
		7	6

Table A.2: Algorithm trace for *IsPalindrome*

We cannot stress enough how important such traces are when designing your algorithm. You can use these trace tables to verify algorithm correctness. At the cost of a simple table, and quick sketch of the data structure you are operating on you can devise correct algorithms quicker. Visualising the problem domain and keeping track of changing data makes problems a lot easier to solve. Moreover you always have a point of reference which you can look back on.

A.2 Recursive Algorithms

For the most part working through recursive algorithms is as simple as walking through an iterative algorithm. One of the things that we need to keep track of though is which method call returns to who. Most recursive algorithms are much simple to follow when you draw out the recursive calls rather than using a table based approach. In this section we will use a recursive implementation of an algorithm that computes a number from the Fiboncacci sequence.

```
1) algorithm Fibonacci(n)
2)     Pre:  n is the number in the fibonacci sequence to compute
3)     Post: the fibonacci sequence number n has been computed
4)     if n < 1
5)         return 0
6)     else if n < 2
7)         return 1
8)     end if
9)     return Fibonacci(n − 1) + Fibonacci(n − 2)
10) end Fibonacci
```

Before we jump into showing you a diagrammtic representation of the algorithm calls for the *Fibonacci* algorithm we will briefly talk about the cases of the algorithm. The algorithm has three cases in total:

1. $n < 1$
2. $n < 2$
3. $n \geq 2$

The first two items in the preceeding list are the base cases of the algorithm. Until we hit one of our base cases in our recursive method call tree we won't return anything. The third item from the list is our recursive case.

With each call to the recursive case we etch ever closer to one of our base cases. Figure A.2 shows a diagrammtic representation of the recursive call chain. In Figure A.2 the order in which the methods are called are labelled. Figure A.3 shows the call chain annotated with the return values of each method call as well as the order in which methods return to their callers. In Figure A.3 the return values are represented as annotations to the red arrows.

It is important to note that each recursive call only ever returns to its caller upon hitting one of the two base cases. When you do eventually hit a base case that branch of recursive calls ceases. Upon hitting a base case you go back to

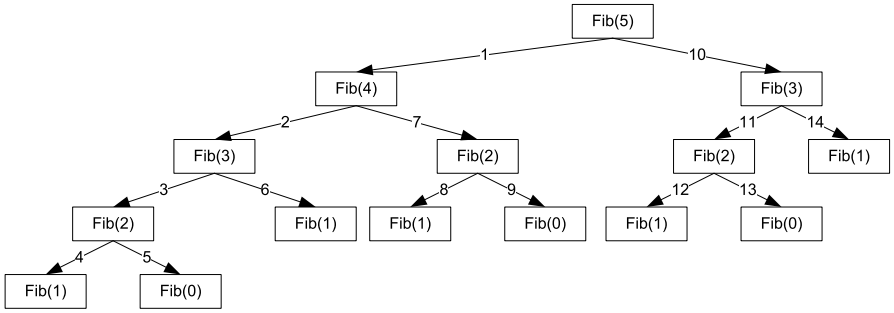


Figure A.2: Call chain for *Fibonacci* algorithm

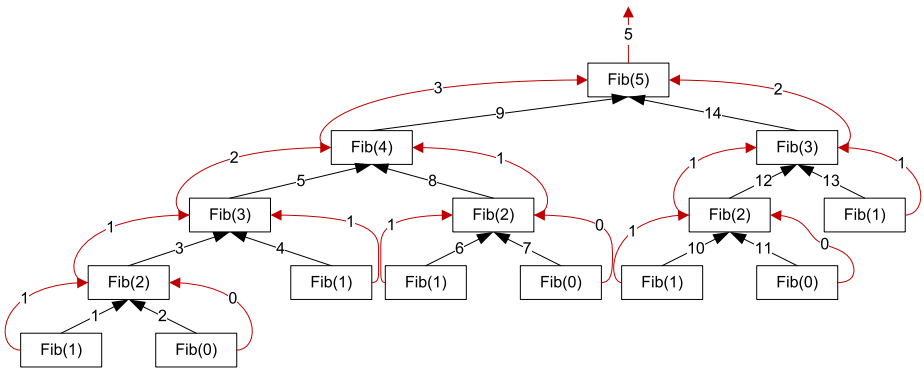


Figure A.3: Return chain for *Fibonacci* algorithm

the caller and continue execution of that method. Execution in the caller is contiued at the next statement, or expression after the recursive call was made.

In the *Fibonacci* algorithms' recursive case we make two recursive calls. When the first recursive call (Fibonacci($n-1$)) returns to the caller we then execute the the second recursive call (Fibonacci($n-2$)). After both recursive calls have returned to their caller, the caller can then subesequently return to its caller and so on.

Recursive algorithms are much easier to demonstrate diagrammatically as Figure A.2 demonstrates. When you come across a recursive algorithm draw method call diagrams to understand how the algorithm works at a high level.

A.3 Summary

Understanding algorithms can be hard at times, particularly from an implementation perspective. In order to understand an algorithm try and work through it using trace tables. In cases where the algorithm is also recursive sketch the recursive calls out so you can visualise the call/return chain.

In the vast majority of cases implementing an algorithm is simple provided that you know how the algorithm works. Mastering how an algorithm works from a high level is key for devising a well designed solution to the problem in hand.

Appendix B

Translation Walkthrough

The conversion from pseudo to an actual imperative language is usually very straight forward, to clarify an example is provided. In this example we will convert the algorithm in §9.1 to the C# language.

```
1) public static bool IsPrime(int number)
2) {
3)     if (number < 2)
4)     {
5)         return false;
6)     }
7)     int innerLoopBound = (int)Math.Floor(Math.Sqrt(number));
8)     for (int i = 1; i < number; i++)
9)     {
10)        for(int j = 1; j <= innerLoopBound; j++)
11)        {
12)            if (i * j == number)
13)            {
14)                return false;
15)            }
16)        }
17)    }
18)    return true;
19) }
```

For the most part the conversion is a straight forward process, however you may have to inject various calls to other utility algorithms to ascertain the correct result.

A consideration to take note of is that many algorithms have fairly strict preconditions, of which there may be several - in these scenarios you will need to inject the correct code to handle such situations to preserve the correctness of the algorithm. Most of the preconditions can be suitably handled by throwing the correct exception.

B.1 Summary

As you can see from the example used in this chapter we have tried to make the translation of our pseudo code algorithms to mainstream imperative languages as simple as possible.

Whenever you encounter a keyword within our pseudo code examples that you are unfamiliar with just browse to Appendix E which descirbes each keyword.

Appendix C

Recursive Vs. Iterative Solutions

One of the most succinct properties of modern programming languages like C++, C#, and Java (as well as many others) is that these languages allow you to define methods that reference themselves, such methods are said to be recursive. One of the biggest advantages recursive methods bring to the table is that they usually result in more readable, and compact solutions to problems.

A recursive method then is one that is defined in terms of itself. Generally a recursive algorithms has two main properties:

1. One or more base cases; and
2. A recursive case

For now we will briefly cover these two aspects of recursive algorithms. With each recursive call we should be making progress to our base case otherwise we are going to run into trouble. The trouble we speak of manifests itself typically as a stack overflow, we will describe why later.

Now that we have briefly described what a recursive algorithm is and why you might want to use such an approach for your algorithms we will now talk about iterative solutions. An iterative solution uses no recursion whatsoever. An iterative solution relies only on the use of loops (e.g. for, while, do-while, etc). The down side to iterative algorithms is that they tend not to be as clear as to their recursive counterparts with respect to their operation. The major advantage of iterative solutions is speed. Most production software you will find uses little or no recursive algorithms whatsoever. The latter property can sometimes be a companies prerequisite to checking in code, e.g. upon checking in a static analysis tool may verify that the code the developer is checking in contains no recursive algorithms. Normally it is systems level code that has this zero tolerance policy for recursive algorithms.

Using recursion should always be reserved for fast algorithms, you should avoid it for the following algorithm run time deficiencies:

1. $O(n^2)$
2. $O(n^3)$

3. $O(2^n)$

If you use recursion for algorithms with any of the above run time efficiency's you are inviting trouble. The growth rate of these algorithms is high and in most cases such algorithms will lean very heavily on techniques like divide and conquer. While constantly splitting problems into smaller problems is good practice, in these cases you are going to be spawning a lot of method calls. All this overhead (method calls don't come *that* cheap) will soon pile up and either cause your algorithm to run a lot slower than expected, or worse, you will run out of stack space. When you exceed the allotted stack space for a thread the process will be shutdown by the operating system. This is the case irrespective of the platform you use, e.g. .NET, or native C++ etc. You can ask for a bigger stack size, but you typically only want to do this if you have a very good reason to do so.

C.1 Activation Records

An activation record is created every time you invoke a method. Put simply an activation record is something that is put on the stack to support method invocation. Activation records take a small amount of time to create, and are pretty lightweight.

Normally an activation record for a method call is as follows (this is very general):

- The actual parameters of the method are pushed onto the stack
- The return address is pushed onto the stack
- The top-of-stack index is incremented by the total amount of memory required by the local variables within the method
- A jump is made to the method

In many recursive algorithms operating on large data structures, or algorithms that are inefficient you will run out of stack space quickly. Consider an algorithm that when invoked given a specific value it creates many recursive calls. In such a case a big chunk of the stack will be consumed. We will have to wait until the activation records start to be unwound after the nested methods in the call chain exit and return to their respective caller. When a method exits it's activation record is unwound. Unwinding an activation record results in several steps:

1. The top-of-stack index is decremented by the total amount of memory consumed by the method
2. The return address is popped off the stack
3. The top-of-stack index is decremented by the total amount of memory consumed by the actual parameters

While activation records are an efficient way to support method calls they can build up very quickly. Recursive algorithms can exhaust the stack size allocated to the thread fairly fast given the chance.

Just about now we should be dusting the cobwebs off the age old example of an iterative vs. recursive solution in the form of the Fibonacci algorithm. This is a famous example as it highlights both the beauty and pitfalls of a recursive algorithm. The iterative solution is not as pretty, nor self documenting but it does the job a lot quicker. If we were to give the Fibonacci algorithm an input of say 60 then we would have to wait a while to get the value back because it has an $O(g^n)$ run time. The iterative version on the other hand has a $O(n)$ run time. Don't let this put you off recursion. This example is mainly used to shock programmers into thinking about the ramifications of recursion rather than warning them off.

C.2 Some problems are recursive in nature

Something that you may come across is that some data structures and algorithms are actually recursive in nature. A perfect example of this is a tree data structure. A common tree node usually contains a value, along with two pointers to two other nodes of the same node type. As you can see tree is recursive in its makeup wit each node possibly pointing to two other nodes.

When using recursive algorithms on tree's it makes sense as you are simply adhering to the inherent design of the data structure you are operating on. Of course it is not all good news, after all we are still bound by the limitations we have mentioned previously in this chapter.

We can also look at sorting algorithms like merge sort, and quick sort. Both of these algorithms are recursive in their design and so it makes sense to model them recursively.

C.3 Summary

Recursion is a powerful tool, and one that all programmers should know of. Often software projects will take a trade between readability, and efficiency in which case recursion is great provided you don't go and use it to implement an algorithm with a quadratic run time or higher. Of course this is not a rule of thumb, this is just us throwing caution to the wind. Defensive coding will always prevail.

Many times recursion has a natural home in recursive data structures and algorithms which are recursive in nature. Using recursion in such scenarios is perfectly acceptable. Using recursion for something like linked list traversal is a little overkill. Its iterative counterpart is probably less lines of code than its recursive counterpart.

Because we can only talk about the implications of using recursion from an abstract point of view you should consult your compiler and run time environment for more details. It may be the case that your compiler recognises things like tail recursion and can optimise them. This isn't unheard of, in fact most commercial compilers will do this. The amount of optimisation compilers can

do though is somewhat limited by the fact that you are still using recursion. You, as the developer have to accept certain accountability's for performance.

Appendix D

Testing

Testing is an essential part of software development. Testing has often been discarded by many developers in the belief that the burden of proof of their software is on those within the company who hold test centric roles. This couldn't be further from the truth. As a developer you should at least provide a suite of unit tests that verify certain boundary conditions of your software.

A great thing about testing is that you build up progressively a safety net. If you add or tweak algorithms and then run your suite of tests you will be quickly alerted to any cases that you have broken with your recent changes. Such a suite of tests in any sizeable project is absolutely essential to maintaining a fairly high bar when it comes to quality. Of course in order to attain such a standard you need to think carefully about the tests that you construct.

Unit testing which will be the subject of the vast majority of this chapter are widely available on most platforms. Most modern languages like C++, C#, and Java offer an impressive catalogue of testing frameworks that you can use for unit testing.

The following list identifies testing frameworks which are popular:

JUnit: Targeted at Jav., `http://www.junit.org/`

NUnit: Can be used with languages that target Microsoft's Common Language Runtime. `http://www.nunit.org/index.php`

Boost Test Library: Targeted at C++. The test library that ships with the incredibly popular Boost libraries. `http://www.boost.org`. A direct link to the libraries documentation `http://www.boost.org/doc/libs/1_36_0/libs/test/doc/html/index.html`

CppUnit: Targeted at C++. `http://cppunit.sourceforge.net/`

Don't worry if you think that the list is very sparse, there are far more on offer than those that we have listed. The ones listed are the testing frameworks that we believe are the most popular for C++, C#, and Java.

D.1 What constitutes a unit test?

A unit test should focus on a single atomic property of the subject being tested. Do not try and test many things at once, this will result in a suite of somewhat

unstructured tests. As an example if you were wanting to write a test that verified that a particular value V is returned from a specific input I then your test should do the smallest amount of work possible to verify that V is correct given I. A unit test should be simple and self describing.

As well as a unit test being relatively atomic you should also make sure that your unit tests execute quickly. If you can imagine in the future when you may have a test suite consisting of thousands of tests you want those tests to execute as quickly as possible. Failure to attain such a goal will most likely result in the suite of tests not being ran that often by the developers on your team. This can occur for a number of reasons but the main one would be that it becomes incredibly tedious waiting several minutes to run tests on a developers local machine.

Building up a test suite can help greatly in a team scenario, particularly when using a continuous build server. In such a scenario you can have the suite of tests devised by the developers and testers ran as part of the build process.

Employing such strategies can help you catch niggling little error cases early rather than via your customer base. There is nothing more embarrassing for a developer than to have a very trivial bug in their code reported to them from a customer.

D.2 When should I write my tests?

A source of great debate would be an understatement to personify such a question as this. In recent years a test driven approach to development has become very popular. Such an approach is known as test driven development, or more commonly the acronym TDD.

One of the founding principles of TDD is to write the unit test first, watch it fail and then make it pass. The premise being that you only ever write enough code at any one time to satisfy the state based assertions made in a unit test. We have found this approach to provide a more structured intent to the implementation of algorithms. At any one stage you only have a single goal, to make the failing test pass. Because TDD makes you write the tests up front you never find yourself in a situation where you forget, or can't be bothered to write tests for your code. This is often the case when you write your tests after you have coded up your implementation. We, as the authors of this book ourselves use TDD as our preferred method.

As we have already mentioned that TDD is our favoured approach to testing it would be somewhat of an injustice to not list, and describe the mantra that is often associate with it:

Red: Signifies that the test has failed.

Green: The failing test now passes.

Refactor: Can we restructure our program so it makes more sense, and easier to maintain?

The first point of the above list always occurs at least once (more if you count the build error) in TDD initially. Your task at this stage is solely to make the test pass, that is to make the respective test green. The last item is based around

the restructuring of your program to make it as readable and maintainable as possible. The last point is very important as TDD is a progressive methodology to building a solution. If you adhere to progressive revisions of your algorithm restructuring when appropriate you will find that using TDD you can implement very cleanly structured types and so on.

D.3 How seriously should I view my test suite?

Your tests are a major part of your project ecosystem and so they should be treated with the same amount of respect as your production code. This ranges from correct, and clean code formatting, to the testing code being stored within a source control repository.

Employing a methodology like TDD, or testing after implementing you will find that you spend a great amount of time writing tests and thus they should be treated no differently to your production code. All tests should be clearly named, and fully documented as to their intent.

D.4 The three A's

Now that you have a sense of the importance of your test suite you will inevitably want to know how to actually structure each block of imperatives within a single unit test. A popular approach - the three A's is described in the following list:

Assemble: Create the objects you require in order to perform the state based assertions.

Act: Invoke the respective operations on the objects you have assembled to mutate the state to that desired for your assertions.

Assert: Specify what you expect to hold after the previous two steps.

The following example shows a simple test method that employs the three A's:

```
public void MyTest()
{
    // assemble
    Type t = new Type();
    // act
    t.MethodA();
    // assert
    Assert.IsTrue(t.BoolExpr)
}
```

D.5 The structuring of tests

Structuring tests can be viewed upon as being the same as structuring production code, e.g. all unit tests for a *Person* type may be contained within

a *PersonTest* type. Typically all tests are abstracted from production code. That is that the tests are disjoint from the production code, you may have two dynamic link libraries (dll); the first containing the production code, the second containing your test code.

We can also use things like inheritance etc when defining classes of tests. The point being that the test code is very much like your production code and you should apply the same amount of thought to its structure as you would do the production code.

D.6 Code Coverage

Something that you can get as a product of unit testing are code coverage statistics. Code coverage is merely an indicator as to the portions of production code that your units tests cover. Using TDD it is likely that your code coverage will be very high, although it will vary depending on how easy it is to use TDD within your project.

D.7 Summary

Testing is key to the creation of a moderately stable product. Moreover unit testing can be used to create a safety blanket when adding and removing features providing an early warning for breaking changes within your production code.

Appendix E

Symbol Definitions

Throughout the pseudocode listings you will find several symbols used, describes the meaning of each of those symbols.

Symbol	Description
$\leftarrow$	Assignment.
$=$	Equality.
$\leq$	Less than or equal to.
$<$	Less than.*
$\geq$	Greater than or equal to.
$>$	Greater than.*
$\neq$	Inequality.
$\emptyset$	Null.
and	Logical and.
or	Logical or.
whitespace	Single occurrence of whitespace.
yield	Like **return** but builds a sequence.

Table E.1: Pseudo symbol definitions

* This symbol has a direct translation with the vast majority of imperative counterparts.