Contents

1 Introduction 1

1.1 What this book is, and what it isn't ... 1

1.2 Assumed knowledge ... 1

1.2.1 Big Oh notation ... 1

1.2.2 Imperative programming language ... 3

1.2.3 Object oriented concepts ... 4

1.3 Pseudocode ... 4

1.4 Tips for working through the examples ... 6

1.5 Book outline ... 6

1.6 Testing ... 7

1.7 Where can I get the code? ... 7

1.8 Final messages ... 7

I Data Structures 8

2 Linked Lists 9

2.1 Singly Linked List ... 9

2.1.1 Insertion ... 10

2.1.2 Searching ... 10

2.1.3 Deletion ... 11

2.1.4 Traversing the list ... 12

2.1.5 Traversing the list in reverse order ... 13

2.2 Doubly Linked List ... 13

2.2.1 Insertion ... 15

2.2.2 Deletion ... 15

2.2.3 Reverse Traversal ... 16

2.3 Summary ... 17

3 Binary Search Tree 19

3.1 Insertion ... 20

3.2 Searching ... 21

3.3 Deletion ... 22

Preview from Notesale.co.uk
Page 1 of 106

B Translation Walkthrough 91

B.1 Summary 92

C Recursive Vs. Iterative Solutions 93

C.1 Activation Records 94

C.2 Some problems are recursive in nature 95

C.3 Summary 95

D Testing 97

D.1 What constitutes a unit test? 97

D.2 When should I write my tests? 98

D.3 How seriously should I view my test suite? 99

D.4 The three A's 99

D.5 The structuring of tests 99

D.6 Code Coverage 100

D.7 Summary 100

E Symbol Definitions 101

Chapter 1

Introduction

Preview from Notesale.co.uk
Page 4 of 106

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) algorithm
2)     Pre: head
3)
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 6= ∅
5)          curr ← tail
6)          while curr 6= head
7)          prev ← head
8)          while prev.Next 6= 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.

Preview from Notesale.co.uk
Page 19 of 106

```
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 (nonrecursive) 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.

Preview from Notesale.co.uk
Page 26 of 106

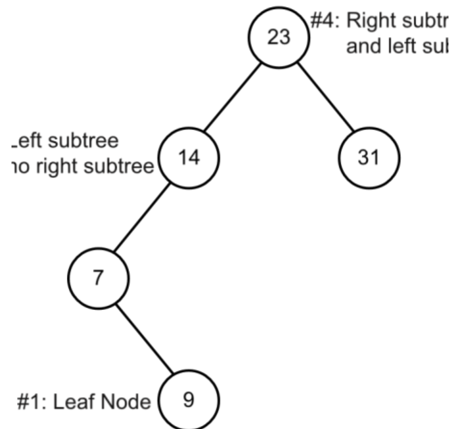


Figure 3.2: binary search tree deletion cases

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 6= ∅
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 =6 ∅ **and** *nodeToRemove*.Right = ∅

Preview from Notesale.co.uk
Page 28 of 106

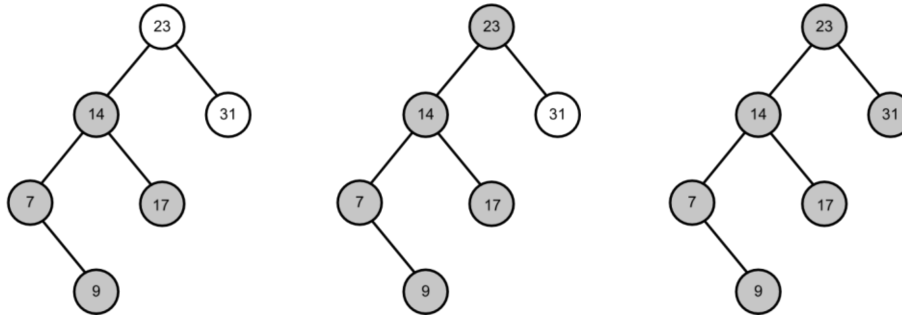


Figure 3.5: Inorder visit binary search tree example

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* 6= ∅
5) Inorder(*root*.Left)
6) **yield** *root*.Value
7) Inorder(*root*.Right)
8) **end if**
9) **end** Inorder

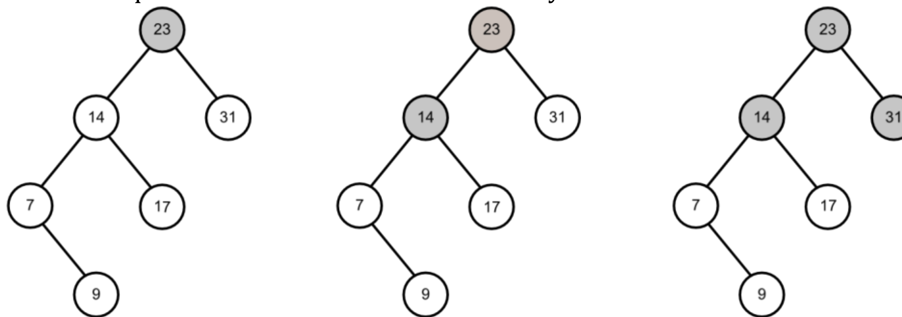
Preview from Notesale.co.uk
Page 34 of 106

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.



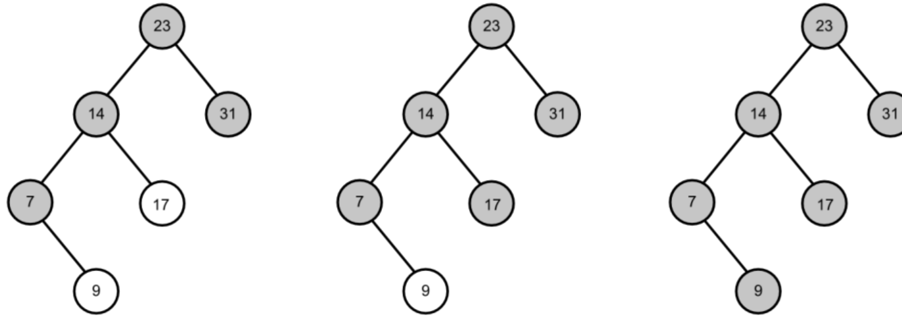


Figure 3.6: Breadth First visit binary search tree example

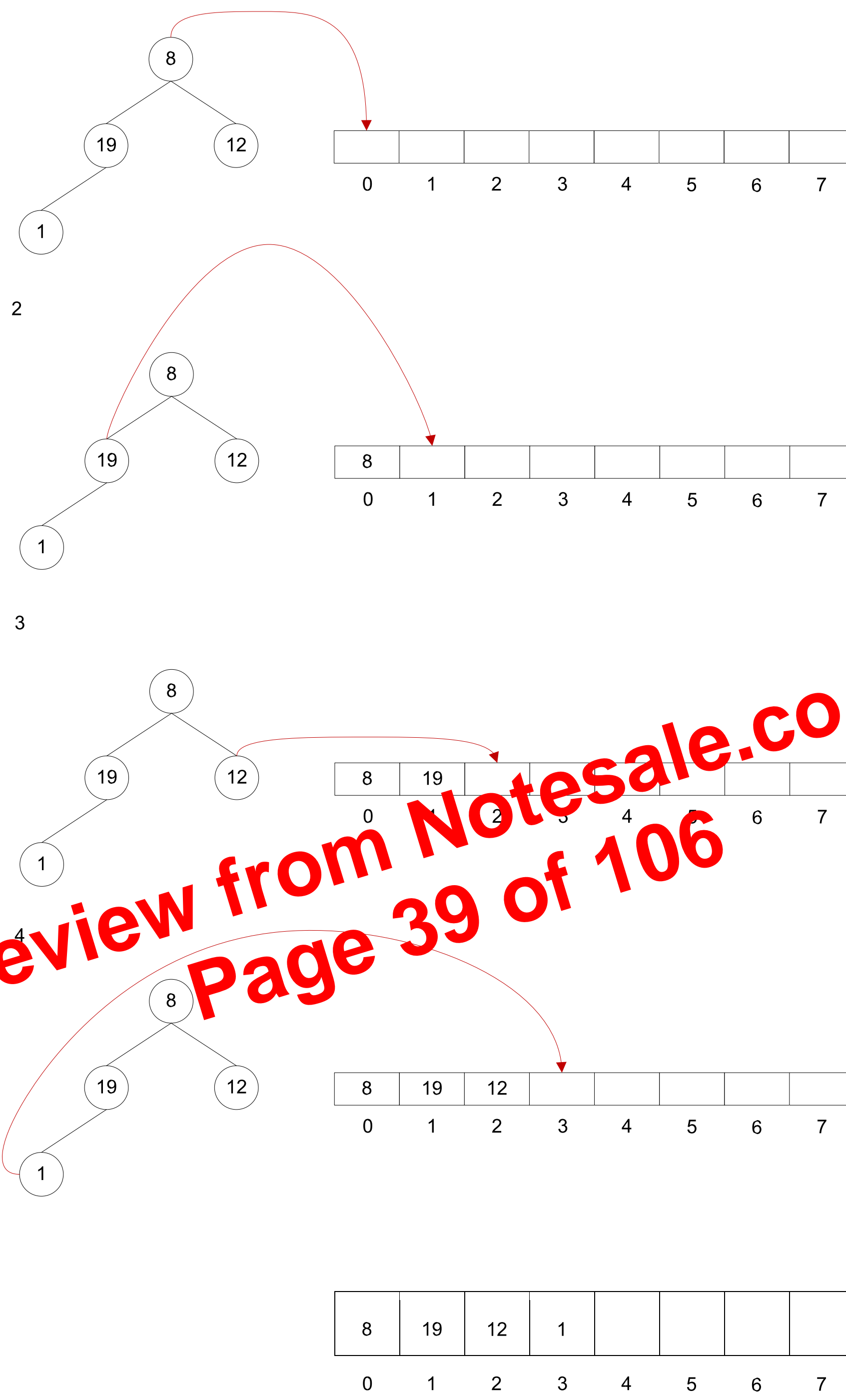
```
1)  algorithm BreadthFirst(root)
2)  Pre: root is the root node of the BST
3)  Post: the nodes in the BST have been visited in breadth first order
4)  q ← queue
5)  while root =6 ∅
6)  yield root.Value
7)  if root.Left 6= ∅
8)  q.Enqueue(root.Left)
9)  end if
10) if root.Right =6 ∅
11) q.Enqueue(root.Right)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

Preview from Notesale.co.uk
Page 35 of 106

3.8 Summary

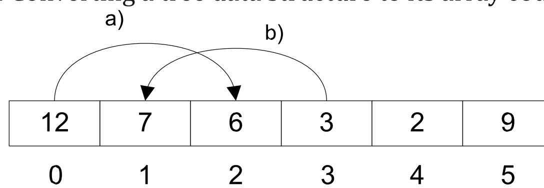
A binary search tree is a good solution when you need to represent types that are ordered according to some custom rules inherent to that type. With logarithmic insertion, lookup, and deletion it is very effecient. Traversal remains linear, but there are many ways in which you can visit the nodes of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



Preview from Notesale.co.uk
Page 39 of 106

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



```
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 ∀ 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.

Preview from Notesale.co.uk
Page 45 of 106

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.

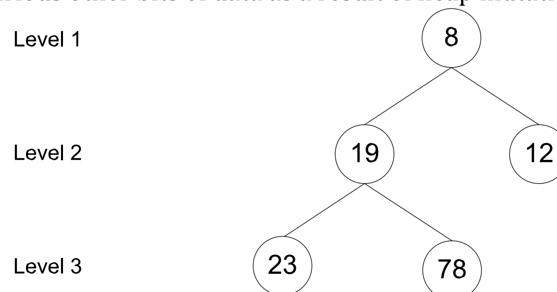


Figure 4.7: Determining 10 is not in the heap after inspecting the nodes of Level 2

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.

Preview from Notesale.co.uk
Page 51 of 106

Chapter 7

AVL Tree

In the early 60's G.M. Adelson-Velsky and E.M. Landis invented the first selfbalancing 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.

Preview from Notesale.co.uk
Page 58 of 106

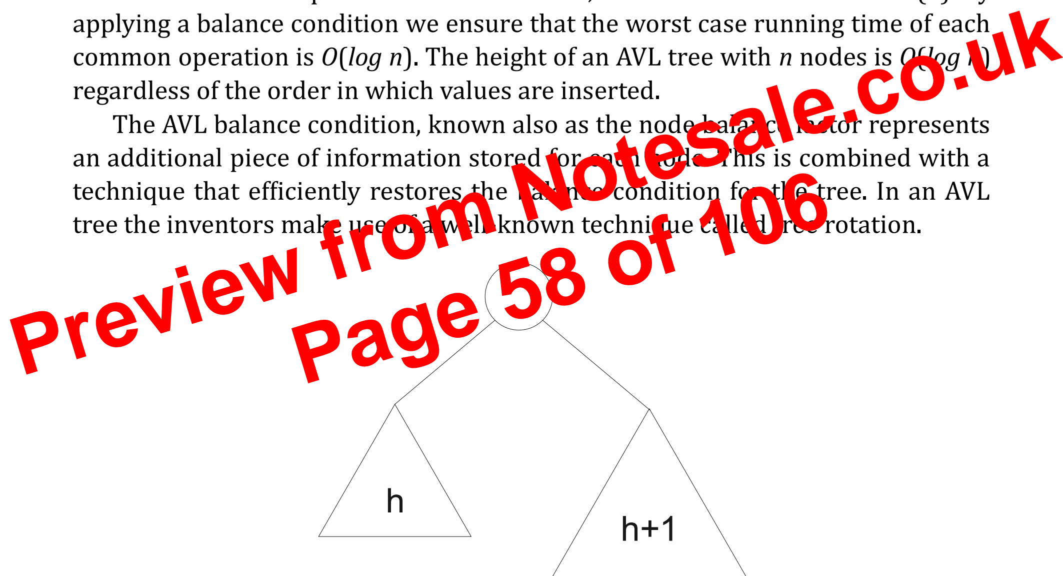


Figure 7.1: The left and right subtrees of an AVL tree differ in height by at most 1

1) **algorithm**
2) **Pre:**
3) **Post:**
4)

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).

Preview from Notesale.co.uk
Page 64 of 106

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* ← (*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* ← (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 6= ∅
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)  for i ← 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
```

Preview from Notesale.co.uk
Page 75 of 106

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.

B	e	n		a	t	e		h	a	y
0	1	2	3	4	5	6	7	8	9	10

Figure 11.2: String with three words

B	e	n		a	t	e			h	a	y
0	1	2	3	4	5	6	7	8	9	10	11

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.

Preview from Notesale.co.uk
Page 87 of 106

1) **algorithm** WordCount(*value*)
2) **Pre:** *value* $\neq \emptyset$
3) **Post:** the number of words contained within *value* is determined
4) *inWord* $\leftarrow$ *true*
5) *wordCount* $\leftarrow$ 0
6) *index* $\leftarrow$ 0
7) // skip initial white space
8) **while** *value*[*index*] = whitespace **and** *index* < *value*.Length −1
9) *index* $\leftarrow$ *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

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

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.

Preview from Notesale.co.uk
Page 95 of 106

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.

Preview from Notesale.co.uk
Page 101 of 106

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.

Preview from Notesale.co.uk
Page 103 of 106

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