1.2.3 Best Case Analysis

- The best case analysis of an algorithm gives a lower bound on the resources required by the algorithm.
- The best case running time of an algorithm determines the minimum amount of time taken by the algorithm on any input.
- The best case running time ensures that the algorithm runs the fastest.

In general worst case, average case and best case analyses are done to determine the running time of algorithms but they can be useful for determining the requirement of memory or other resources as well.

1.3 AMORTIZED ANALYSIS

- In an amortized analysis of running time of an algorithm, the time required to perform a sequence of data structure operations is averaged over all the operations performed.
- Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation within the sequence might be expensive.
- Amortized analysis differs from average case analysis in the sense that amortized analysis does not involve probability. Amortized analysis guarantees the average performance of each operation in the worst case.

1.3.1 Aggregate Analysis

- In an aggregate analysis, for all *n*, a sequence of *n* operations takes worst case time *T* (*n*) in total.
- Therefore, in the worst case, the average cost or amortized cost, per operation is T(n)/n. Here the amortized cost applies to each operation, even when there are several types of operations in the sequence.

1.3.2 The Potential Method

- The potential method represents the prepaid work as 'potential' which can be released to pay for future operations.
- This potential is associated with the data structure as a whole rather than with specific objects within the data structure.
- Let *n* operations are performed, starting with an initial data structure *D*₀.

i.	Enque	eue (Q	, 10)									
	1	2	3	4	1	5		6		7		
			12	2 8	3	4		9		10		
7 = = 7												
	1	2	3	4	1	5		6		7		
			12	2 8	3	4		9		10		
	↑		1	L L								
tail $[Q] = 1$ head $[Q] = 3$												
ii. Enqueue (<i>Q</i> , 5)												
	1	2	3	4		5		6		7		
	5		12	8		4	(9	1	.0		
	1≠7	-	-						-			
	1	2	3	4		5	6	6	2	7		
	5		12	8		4	Ģ	9	1	0		
		1	1									
	tail [Q	2] = 2	head	d [Q]	= 3							

Algorithm

Dequeu	e (<i>Q</i>)
Line 1	$x \leftarrow Q$ [head [Q]]
Line 2	if head $[Q] = = \text{length } [Q]$
Line 3	head $[Q] \leftarrow 1$
Line 4	else head $[Q] \leftarrow$ head $[Q] + 1$
Line 5	return x

Analysis

• The Dequeue procedure takes *O* (1) time.

Explanation

- The input to the Dequeue procedure is a queue *Q*. This procedure is used to delete elements from the queue.
- Line 1 indicates that *x* is the element at location *Q*[head [*Q*]].
- Line 2 checks the *if* condition. If this condition is true then the execution of Line 3 takes place else the control goes to Line 4.
- The *if* condition is true when the attribute head [*Q*] is equal to the length of *Q*. It means head [*Q*] points to the last location of *Q*.



Algorithm

Right-Rotate (T, x)						
Line 1	$y \leftarrow left [x]$					
Line 2	$left [x] \leftarrow right [y]$					
Line 3	if right $[y] \neq$ nil $[T]$					
Line 4	$P[right[y]] \leftarrow x$					
Line 5	$P[y] \leftarrow P[x]$					
Line 6	if $P[x] = = nil[T]$					
Line 7	root $[T] \leftarrow y$					
Line 8	elseif $x = = right [P[x]]$					
Line 9	right [P[x]] ← y					
Line 10	else left $[P[x]] \leftarrow y$					
Line 11	right $[y] \leftarrow x$					
Line 12	$P[x] \leftarrow y$					

Assumptions

Following are the assumptions made from the above algorithm:

- i. *y* is left child of x
- ii. y is not nil [T]
- iii. Parent of root is nil [*T*]

- Line 3 makes pointer *y* to point to the right child of the parent of parent of *z*. It means that *y* points to the right child of the grand-parent of *z*, or in other words *y* points to *z*'s uncle.
- Line 4 checks the *if* condition. This condition is true if the color of node *y* is red. If this condition is true then the execution of Lines 5, 6, 7 and 8 takes place. If this condition is false then the procedure checks the *elseif* condition of Line 9.
- Line 5 paints the parent of *z* black.
- Line 6 paints node *y* black.
- Line 7 paints grandparent of *z* red.
- Line 8 makes the earlier grandparent of *z* as the new *z*. So the pointer *z* moves up two levels in the tree.
- Line 9 checks the *elseif* condition. This condition is true if *z* is the right child of the parent of *z*. If this condition is false then the control goes to Line 12.
- Line 10 makes the earlier parent of *z* as the new *z*. So the pointer *z* moves one level up in the tree.
- Line 11 calls the procedure Left-Rotate (*T*, *z*).
- Line 12 paints the parent of *z* black.
- Line 13 paints the grandparent of *z* red.
- Line 14 calls the procedure Right-Rotate (*T*, *P* [*P*[*z*]]).
- Line 15 makes pointer *y* to point to the left child of the parent of *z*. It means that *y* points to the left child of the grandparent of *z* or in other words *y* points to *z*'s uncle.
- Line 16 checks the *if* condition. This condition is true if the color of node *y* is red. If this condition is true then the execution of Lines 17, 18, 19 and 20 takes place. If this condition is false then the procedure checks the *elseif* condition of Line 21.
- Line 17 paints the parent of *z* black.
- Line 18 paints the node y (z's uncle) black.
- Line 19 paints grandparent of *z* red.
- Line 20 makes the earlier grandparent of *z* as the new *z*. So the pointer *z* moves up two levels in the tree.
- Line 21 checks the *elseif* condition. This condition is true if *z* is the left child of the parent of *z*. If this condition is false then the control goes to Line 24.
- Line 22 makes the earlier parent of *z* as the new *z*. So the pointer *z* moves one level up in the tree.
- Line 23 calls the procedure Right-Rotate (*T*, *z*).
- Line 24 paints the parent of *z* black.

Contents

vii

1. Algorithm	1
 1.1 Asymptotic Notation 1 1.2 Worst Case, Average Case And Best Case Analysis 3 1.2.1 Worst Case Analysis 3 1.2.2 Average Case Analysis 3 1.2.3 Best Case Analysis 4 1.3 Amortized Analysis 4 1.3.1 Aggregate Analysis 4 1.3.2 The Potential Method 4 	
2. Data Structure	6
 2.1 Linear Arrays 7 2.2 Queues 7 2.3 Linked Lists 11 2.4 Graphs 11 2.5 Red Black (RB) Trees 12 2.6 B-Trees 54 2.7 Fibonacci Heaps 84 2.8 Data Structures for Disjoint Sets 111 	
3. Techniques for the Design of Algorithms	112
 3.1 Dynamic Programming Approach 112 3.1.1 Matrix Chain Multiplication 113 3.1.2 Longest Common Subsequence (LCS) 131 3.1.3 Knapsack Problem 135 3.2 Greedy Approach 139 3.2.1 Activity Selection Problem 140 3.2.2 Huffman Codes 142 3.2.3 Single-Source Shortest Paths 149 3.2.3.1 Bellman-Ford algorithm 150 3.2.4 Minimum Spanning Trees 164 3.2.4.1 Kruskal's algorithm 165 3.2.4.2 Prim's algorithm 171 	

Preface

+ x Design and Analysis of Algorithms

- 3.3 Divide and Conquer Approach 178 3.3.1 Quick Sort 178
- 3.4 Decrease and Conquer Approach 183
 - 3.4.1 Insertion Sort 183
 - 3.4.2 Breadth First Search (BFS) 189
 - 3.4.3 Depth First Search (DFS) 199
- 3.5 Data Structure Based Approach 210 3.5.1 Heap Sort 211
- 3.6 Backtracking Approach 234
 - 3.6.1 *n*-Queens Problem 235
 - 3.6.2 Graph Coloring Problem 235
 - 3.6.3 Hamiltonian Circuits Problem 237

4. Miscellaneous Topics

239

- 4.1 Counting Sort 239
- 4.2 Radix Sort 244
- 4.3 Bucket Sort 245
- 4.4 String Matching 247
- 4.5 Methods to Solve Recurrence Relations 250
 - 4.5.1 The Recursion Tree Method 250
 - 4.5.2 The Master Method 251

Bibliography

Index

253

255