RSSB
Rajasthan Staff Selection Board

Basic Computer Instructor Examination, 2022

PAPER-II

COMPUTER SCIENCE

Comprehensive theory in lucid language *with* practice questions

Also useful for Senior Computer Instructor Examination

MADE EASY
Publications

MADE EASY Publications

www.madeeasypublications.org

MADE EASY Publications Pvt. Ltd

Corporate Office: 44-A/4, Kalu Sarai (Near Hauz Khas Metro Station), New Delhi-110016
E-mail: infomep@madeeasy.in
Contact: 011-45124660, 08860378007

Visit us at: www.madeeasypublications.org

Rajasthan Staff Selection Board (RSSB) :
Basic Computer Instructor Examination, 2022 (Paper-II)

First Edition: 2022

Preface

The compilation of this book is motivated by the desire to provide a concise book which can benefit students who are preparing for Basic Computer Instructor Examination conducted by **Rajasthan Staff Selection Board (RSSB).**

B. Singh (Ex. IES)

This textbook provides all the requirements of the students, i.e. comprehensive coverage of Computer Science topics with objective practice questions articulated in a lucid language. This book is also useful for Senior Computer Instructor as well as an array of similar competitive examinations. All the topics are given the emphasis they deserve so that mere reading of the book helps aspirants immensely.

Our team has made their best efforts to remove all possible errors of any kind. Nonetheless, we would highly appreciate and acknowledge if you find and share with us any printing and conceptual errors.

It is impossible to thank all the individuals who helped us, but we would like to sincerely thank all the authors, editors and reviewers for putting in their efforts to publish this book.

With Best Wishes

B. Singh

CMD, MADE EASY Group

CONTENTS

Computer Science

Data Structure and Algorithms

06

6.1 Definition of Algorithms

An algorithm is a bunch of self-contained succession of guidelines or activities that contain limited space or grouping such that it will give us an outcome to a particular issue in a limited measure of time.

A good algorithm ought to be advanced in phrases of time and space. Thus, various sorts of issues require various kinds of algorithmic-strategies to be illuminated in the most improved way.

Types of Algorithms

1. **Brute Force Algorithm:** A brute force algorithm essentially attempts all the chances until an acceptable result is found. This is the most fundamental and least complex type of algorithm. Such types of algorithms are moreover used to locate the ideal or best solution as it checks all the potential solutions.

   Also, it is used for finding an agreeable solution (not the best), basically stopping when an answer to the issue is found. It is a clear way to deal with an issue that is the first approach that strikes our mind after observing the issue.
2. **Recursive Algorithm:** This type of algorithm depends on recursion. In recursion, an issue is comprehended by breaking it into subproblems of a similar kind and calling itself over and over until the issue is unravelled with the assistance of a base condition.

   It solves the base case legitimately and afterwards recurs with a more straightforward or simpler input every time. It is used to take care of the issues which can be broken into less complex or more modest issues of the same sort.
3. **Dynamic Programming Algorithm:** This type of algorithm is also called the memoization technique. This is because, in this, the thought is to store the recently determined outcome to try not to figure it over and over.

   In Dynamic Programming, partition the unpredictable issue into more modest covering subproblems and putting away the outcome for sometime later. In simple language, we can say that it recollects the previous outcome and uses it to discover new outcomes.
4. **Divide and Conquer Algorithm:** In the Divide and Conquer algorithm, the thought is to tackle the issue in two areas, the first section partitions the issue into subproblems of a similar sort. The second section is to tackle the more modest issue autonomously and afterwards add the joined outcome to create the last response to the issue.
5. **Greedy Algorithm:** Now coming towards another type that is a greedy algorithm, so in this, the solution is created portion by portion. The finding to select the following role is accomplished on the purpose that it provides the sudden help and it never deems the options that had assumed lately.
6. **Backtracking Algorithm:** In this type of algorithm, the issue is worked out steadily, for example, it is an algorithmic-procedure for taking care of issues recursively by attempting to construct an answer steadily, each piece, in turn, eliminating those solutions that neglect to fulfil the conditions of the situation at any point of time.
7. **Randomized Algorithm:** In this type of algorithm, a random number is taken for deciding at least once during the computations.

Characteristics of an algorithm

There are some characteristics that every algorithm should follow and here is the list of some of them which we will see one by one.

1. **Input specified:** The input is the information to be changed during the calculation to create the output. An algorithm ought to have at least 0 all around characterized inputs. Input exactness necessitates that you understand what sort of information, how much and what structure the information should be.
2. **Output specified:** The output is the information coming about because of the calculation. An algorithm ought to have at least 1 all around characterized outputs, and should coordinate the ideal output. Output exactness likewise necessitates that you understand what sort of information, how much and what structure the output should be.
3. **Clear and Unambiguous:** Algorithms must determine each step and each of its steps should be clear in all behaviours and must direct to only one meaning. That's why the algorithm should be clear and unambiguous. Details of each step must be likewise be explained (counting how to deal with errors). It ought to contain everything quantitative and not subjective.
4. **Feasible:** The algorithm should be effective which implies that all those means that are needed to get to output must be feasible with the accessible resources. It should not contain any pointless and excess advances which could make an algorithm ineffectual.
5. **Independent:** An algorithm should have step by step directions, which should be independent of any programming code. It should be with the end goal that it very well may be a sudden spike in demand for any of the programming dialects.
6. **Finiteness:** The algorithm must quit, eventually. Stopping may imply that you get the normal output. Algorithms must end after a limited number of steps. An algorithm should not be boundless and consistently end after a finite number of steps. There is no reason for building up an algorithm that is limitless as it will be pointless for us.

Important Algorithms for problem solving

1. **Searching Algorithms:** A search algorithm is an algorithm which solves a search problem. Search algorithms work to retrieve information stored within some data structure, or calculated in the search space of a problem domain, with either discrete or continuous values. Examples :- Linear Search, Binary Search, Ternary Search.
2. **Sorting Algorithms:** A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure. Examples :- Quick sort, insertion sort, merge sort, selection sort, bubble sort, radix sort, counting sort, etc.
3. **Divide and conquer based algorithms:** This technique can be divided into the following three parts:
   - **Divide:** This involves dividing the problem into smaller sub-problems.
   - **Conquer:** Solve sub-problems by calling recursively until solved.
   - **Combine:** Combine the sub-problems to get the final solution of the whole problem.

   Examples: Binary search, merge sort, quick sort, ternary search, etc
4. **Greedy Algorithms:** Greedy is an algorithmic paradigm that builds up a solution piece by piece, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy. Examples :- Kruskal's algorithm, Prim's algorithm, Dijkstra's algorithm, Fractional knapsack problem, Activity selection problem, Huffman coding, Job sequencing problem, etc.
5. **Dynamic Programming based algorithms:** Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The

idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial. **Examples:** Bellman Ford algorithm, Fibonacci series, Matrix Chain multiplication problem, Longest common subsequence problem, 0/1 knapsack problem, etc. following are the two main properties of a problem that suggest that the given problem can be solved using Dynamic programming:

- **Overlapping Subproblems:** In dynamic programming, computed solutions to subproblems are stored in a table so that these don't have to be recomputed thus reducing the time substantiallly.
- **Optimal Substructure:** A given problems has Optimal Substructure Property if optimal solution of the given problem can be obtained by using optimal solutions of its subproblems.

6.2 Abstract Data Types

The Data Type is basically a type of data that can be used in different computer program. It signifies the type like integer, float etc, the space like integer will take 4-bytes, character will take 1-byte of space etc.

The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword "Abstract" is used as we can use these datatypes, we can perform different operations. But how those operations are working that is totally hidden from the user. The ADT is made of with primitive datatypes, but operation logics are hidden.

Some examples of ADT are Stack, Queue, List etc.

Let us see some operations of those mentioned ADT

Stack:

- isFull(), This is used to check whether stack is full or not
- isEmpty(), This is used to check whether stack is empty or not
- push(x), This is used to push x into the stack
- pop(), This is used to delete one element from top of the stack
- peek(), This is used to get the top most element of the stack
- size(), this function is used to get number of elements present into the stack

Queue:

- isFull(), This is used to check whether queue is full or not
- isEmpty(), This is used to check whether queue is empty or not
- insert(x), This is used to add x into the queue at the rear end
- delete(), This is used to delete one element from the front end of the queue
- size(), this function is used to get number of elements present into the queue

List:

- size(), this function is used to get number of elements present into the list
- insert(x), this function is used to insert one element into the list
- remove(x), this function is used to remove given element from the list
- get(i), this function is used to get element at position i
- replace(x, y), this function is used to replace x with y value

Different types of data structures

Arrays

An array is a sequential collection of elements of same data type and stores data elements in a continuous memory location. The elements of an array are accessed by using an index. The index of an array of size N can range from 0 to N-1. For example, if your array size is 5, then your index will range from 0 to 4 (5-1). Each element of an array can be accessed by using arr[index].

Declaring an array is language-specific.

For example, in C/C++, to declare an array, you must specify, the following:

- **Size of the array:** This defines the number of elements that can be stored in the array.
- **Type of array:** This defines the type of each element i.e. number, character, or any other data type.

  For example:

  int arr[10];

  The above declares an array named “arr” which stores values of type integer and the total size of the array is 10 integer elements.

  This is a static array and the other kind is dynamic array, where type is just enough for declaration. In dynamic arrays, size increases as more elements are added to the array.

  Dynamic declaration of a 1-Dimensional array:

  int * ptr; // This is a pointer of type integer.

  ptr=(int*) malloc(10*sizeof(int)); // malloc allocates memory equaling 10 integer elements and passes the starting address of this block of memory to ptr.

2-Dimensional Arrays

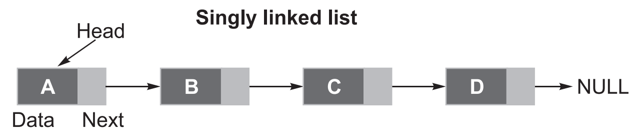
A two-dimensional array is similar to a one-dimensional array, but it can be visualised as a grid (or table) with rows and columns.

For example, a nine-by-nine grid could be referenced with numbers for each row and letters for each column. A nine-by-nine, two-dimensional array could be declared with a statement such as:

- arr[9][9]; // arr contains 9 rows and 9 columns ranging from 0 to 8.
- Another way to declare a 2-dimensional array with initial values is as follows :-
- int[][] myArray = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16} };

Linked List Data structure

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the data and the address of the next node.

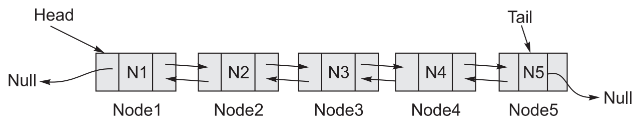


The above is the most simplest form of linked list called a singly linked list. Head is the starting node of the linked list. Each node contains a data element and a pointer which points to the next node. Linked list are useful when data is to be stored in non-contiguous memory locations.

Types of Linked List: There are three common types of Linked List.

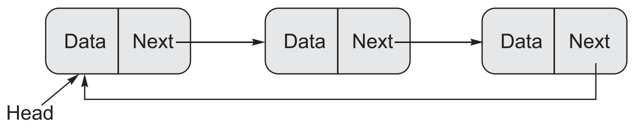
1. Singly Linked List - Already covered above.
2. Doubly Linked List
3. Circular Linked List
4. Circular Doubly Linked List

Doubly Linked List



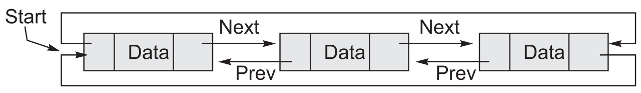
In a doubly linked list, each node contains two pointers instead of one. The previous pointer points to the previous node and the next pointer points to the next node. This allows us to travel in forward as well as backward direction in a doubly linked list. The previous pointer of the first node and the next pointer of the last node should be assigned NULL value.

Circular Linked List



A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.

Circular Doubly Linked List



In a circular doubly linked list, the next pointer of the last member points to the first member and the previous pointer of the first member points to the last member in addition to the features present in a doubly linked list.

Singly linked list operation	Real time complexity	Assumed time complexity
Access i-th element	O(√N * N)	O(N)
Traverse all elements	O(√N * N)	O(N)
Insert element E at current point	O(1)	O(1)
Delete current element	O(1)	O(1)
Insert element E at front	O(1)	O(1)
Insert element E at end	O(√N * N)	O(N)

6.3 Stack

Stackisanabstractdatatypewithabounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the top of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.

Properties of stack

1. Stack is an ordered list of similar data type.
2. Stack is a LIFO(Last in First out) structure or we can say FILO(First in Last out).
3. push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called Top.
4. Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- **Push Operation :** Insert an element at the top of the stack. O(1)
- **Pop Operation :** Deletes an element from the top of the stack. O(1)
- **Top Operation :** Returns the element present at the tpo of the stack. O(1)
- **Search Operation :** Searches a particular element in the stack. O(n)

The time complexities for push() and pop() functions are O(1) because we always have to insert or remove the data from the top of the stack, which is a one step process.

6.4 Queue

Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the REAR(also called tail), and the removal of existing element takes place from the other end called as FRONT(also called head).

Properties of a queue

- Queue is a FIFO data structure which is also called as First In First Out. This means that the element which is inserted first in the queue data structure should be the first one to be deleted.
- Like stack, queue is also an ordered list of elements of similar data types.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.

Operations performed on a queue

- **Enqueue:** This operation inserts a new element at the rear position in the queue. This takes O(1) time.
- **Dequeue:** This operation deletes an element from front position of the queue. This takes O(1) time.

Applications of a queue

- Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
- In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
- Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Queue can be implemented using an array, stack or Linked List. The easiest way of implementing a queue is by using an Array.

Different types of queues

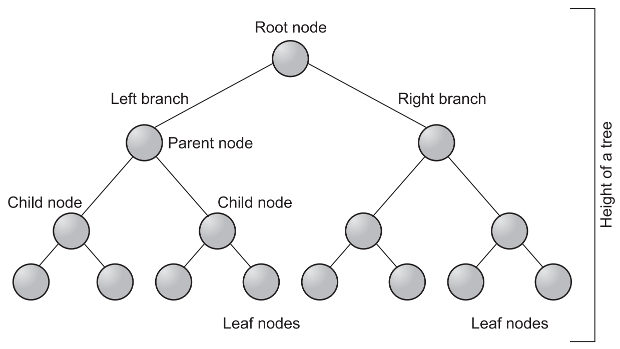
There are four different types of queues:

- **Simple Queue:** In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.
- **Circular Queue:** In a circular queue, the last element points to the first element making a circular link.
- **Priority Queue:** A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.
- **Double Ended Queue:** In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.

6.5 Binary Tree

A binary tree is a tree-type non-linear data structure with a maximum of two children for each parent. Every node in a binary tree has a left and right reference along with the data element. The node at the top of the hierarchy of a tree is called the root node. The nodes that hold other sub-nodes are the parent nodes.

A parent node has two child nodes: the left child and right child. Hashing, routing data for network traffic, data compression, preparing binary heaps, and binary search trees are some of the applications that use a binary tree.



- **Node:** It represents a termination point in a tree.
- **Root:** A tree's topmost node.
- **Parent:** Each node (apart from the root) in a tree that has at least one sub-node of its own is called a parent node.
- **Child:** A node that straightway came from a parent node when moving away from the root is the child node.
- **Leaf Node:** These are external nodes. They are the nodes that have no child.
- **Internal Node:** As the name suggests, these are inner nodes with at least one child.
- **Depth of a Tree:** The number of edges from the tree's node to the root is.
- Height of a Tree: It is the number of edges from the node to the deepest leaf. The tree height is also considered the root height.

Types of Binary trees

- **Full Binary Tree:** It is a special kind of a binary tree that has either zero children or two children. It means that all the nodes in that binary tree should either have two child nodes of its parent node or the parent node is itself the leaf node or the external node.

  In other words, a full binary tree is a unique binary tree where every node except the external node has two children. When it holds a single child, such a binary tree will not be a full binary tree. Here, the quantity of leaf nodes is equal to the number of internal nodes plus one. The equation is like L=I+1, where L is the number of leaf nodes, and I is the number of internal nodes.
- **Complete Binary Tree:** A complete binary tree is another specific type of binary tree where all the tree levels are filled entirely with nodes, except the lowest level of the tree. Also, in the last or the lowest level of this binary tree, every node should possibly reside on the left side.
- **Perfect Binary Tree:** A binary tree is said to be 'perfect' if all the internal nodes have strictly two children, and every external or leaf node is at the same level or same depth within a tree. A perfect binary tree having height 'h' has 2h – 1 node.

- **Balanced Binary Tree:** A binary tree is said to be 'balanced' if the tree height is O(logN), where 'N' is the number of nodes. In a balanced binary tree, the height of the left and the right subtrees of each node should vary by at most one. An AVL Tree and a Red-Black Tree are some common examples of data structure that can generate a balanced binary search tree.
- **Degenerate Binary Tree:** The degenerate binary tree is a tree in which all the internal nodes have only one children. A degenerate binary tree can be either left-skewed tree (when all nodes have only left child) or a right-skewed tree (when all nodes have only right child).

Properties of a Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \ldots 2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to h+1.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

Tree Traversals

Tree traversal means traversing or visiting each node of a tree. Linear data structures like Stack, Queue, linked list have only one way for traversing, whereas the tree has various ways to traverse or visit each node. The following are the three different ways of traversal:

1. Inorder traversal
2. Preorder traversal
3. Postorder traversal

1. **Indorder Traversal:** An inorder traversal is a traversal technique that follows the policy, i.e., Left Root Right. Here, Left Root Right means that the left subtree of the root node is traversed first, then the root node, and then the right subtree of the root node is traversed. Here, inorder name itself suggests that the root node comes in between the left and the right subtrees.
2. **Preorder Traversal:** A preorder traversal is a traversal technique that follows the policy, i.e., Root Left Right. Here, Root Left Right means root node of the tree is traversed first, then the left subtree and finally the right subtree is traversed. Here, the Preorder name itself suggests that the root node would be traversed first.
3. **Postorder Traversal:** A Postorder traversal is a traversal technique that follows the policy, i.e., Left Right Root. Here, Left Right Root means the left subtree of the root node is traversed first, then the right subtree, and finally, the root node is traversed. Here, the Postorder name itself suggests that the root node of the tree would be traversed at the last.

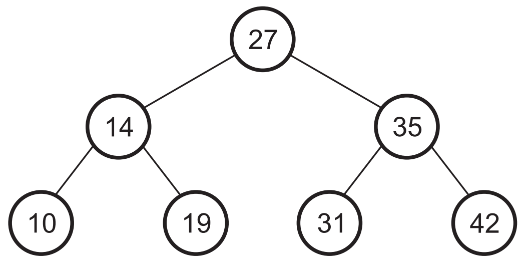
Binary Search Trees

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties –

- The value of the key of the left sub-tree is less than the value of its parent (root) node's key.
- The value of the key of the right sub-tree is greater than or equal to the value of its parent (root) node's key.

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST:



AVL Tree

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

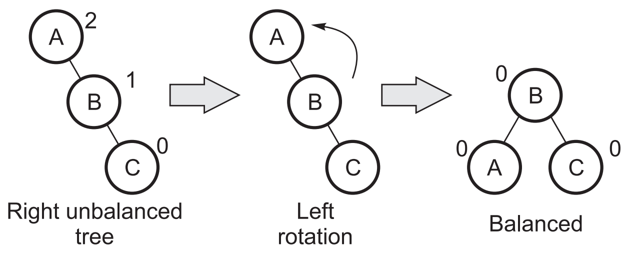
AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (i.e. worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.

We perform rotation in AVL tree only in case if Balance Factor is other than **–1, 0, and 1**. There are basically four types of rotations which are as follows:

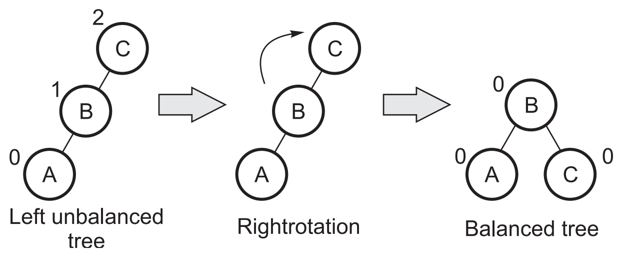
1. **L L rotation:** Inserted node is in the left subtree of left subtree of A
2. **R R rotation:** Inserted node is in the right subtree of right subtree of A
3. **L R rotation:** Inserted node is in the right subtree of left subtree of A
4. **R L rotation:** Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

1. **RR Rotation:** When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor 2.



2. **LL Rotation:** When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



3. **LR Rotation:** Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than –1, 0, or 1.

4. **RL Rotation:** As already discussed, that double rotations are bit tougher than single rotation which has already explained above. RL rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than –1, 0, or 1.

BST operation time complexity analysis

Operation	Best case	Worst case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

In best case, the Binary Search tree is similar to a complete binary tree (Balanced tree). In worst case, the Binary Search tree can be either completely skewed towards the left or the right.

AVL tree operation time complexity analysis

Operation	Average Case	Worst case
Space	O(n)	O(nW)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

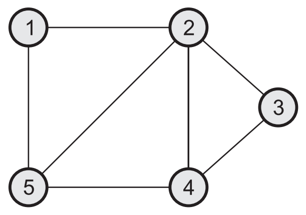
6.6 Graphs and its representation

The graph is a non-linear data structures. This represents data using nodes, and their relations using edges. A graph G has two sections. The vertices, and edges. Vertices are represented using set V, and Edges are represented as set E. So the graph notation is G(V,E). A graph can be classified in different categories:

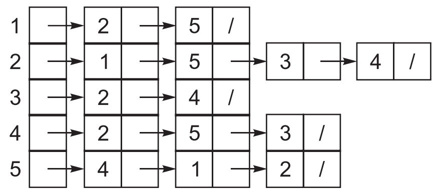
1. **Directed Graphs:** A directed graph, also called a digraph, is a graph in which the edges have a direction. This is usually indicated with an arrow on the edge.
2. **Undirected Graphs:** An undirected graph is graph, i.e., a set of objects (called vertices or nodes) that are connected together, where all the edges are bidirectional. An undirected graph is sometimes called an undirected network.
3. **Simple Graph:** A simple graph is a graph which does not contains more than one edge between the pair of vertices.
4. **Multi Graphs:** A graph in which multiple edges may connect the same pair of vertices is called a multigraph. Since there can be multiple edges between the same pair of vertices, the multiplicity of edge tells the number of edges between two vertices.
5. **Weighted Graphs:** A weighted graph is a graph in which each edge is given a numerical weight. A weighted graph is therefore a special type of labeled graph in which the labels are numbers (which are usually taken to be positive).
6. **Unweighted Graphs:** An unweighted graph is a graph in which every edge is equally weighted i.e., there are no edge weights on any of the edges.
7. **Connected Graph:** A connected graph is an undirected graph where there is a path present between every pair of vertices.
8. **Disconnected Graph:** An undirected graph in which there exists atleast 1 pair of vertices which do not have a path between them is called as a disconnected graph.
9. **Complete Graph:** A simple graph of n vertices having exactly one edge between each pair of vertices is called a complete graph. A complete graph of n vertices is denoted by K_n. Total number of edges are n*(n-1)/2 with n vertices in complete graph.

The graphs are non-linear, and it has no regular structure. To represent a graph in memory, there are few different styles. These styles are:

- **Adjacency matrix representation:** We can represent a graph using Adjacency matrix. The given matrix is an adjacency matrix. It is a binary, square matrix and from ith row to jth column, if there is an edge, that place is marked as 1. When we will try to represent an undirected graph using adjacency matrix, the matrix will be symmetric.
- **Adjacency List representation:** This is another type of graph representation. It is called the adjacency list. This representation is based on Linked Lists. In this approach, each Node is holding a list of Nodes, which are Directly connected with that vertices. At the end of list, each node is connected with the null values to tell that it is the end node of that list.



(a)



(b)

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$

(c)

Figure (a) represents an undirected connected graph. Figure (b) represents the corresponding adjacency list representation for the given graph. Figure (c) represents the corresponding adjacency matrix representation for the given graph.

6.7 Searching and Sorting

In this section we are going to discuss widely used searching and sorting algorithms. The underlying data structure involved in all these algorithms will be arrays.

Linear Search

Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

Algorithm

```
Linear Search ( Array A, Value x)
Step 1   :  Set i to 1
Step 2   :  if i > n then go to step 7
Step 3   :  if A[i] = x then go to step 6
Step 4   :  Set i to i + 1
Step 5   :  Go to Step 2
Step 6   :  Print Element x Found at index i and go
to step 8
Step 7   :  Print element not found
Step 8   :  Exit
```

Time Complexity for Linear Search

Best Case - O(1)

Average Case - O(n)

Worst case - O(n)

Binary Search

Binary search is the search technique that works efficiently on sorted lists. Hence, to search an element into some list using the binary search technique, we must ensure that the list is sorted.

Binary search follows the divide and conquer approach in which the list is divided into two halves, and the item is compared with the middle element of the list. If the match is found then, the location of the middle element is returned. Otherwise, we search into either of the halves depending upon the result produced through the match.

Algorithm

```
binarySearch(arr, x, low, high)
    if low > high
        return False
    else
        mid = (low + high) / 2
    if x == arr[mid]
        return mid
    else if x > arr[mid]      // x is on the right side
        return binarySearch(arr, x, mid + 1, high)
        else                  // x is on the right side
        return binarySearch(arr, x, low, mid - 1)
```

Time Complexity for Binary Search

Given an array of n elements following are the time complexities :-

Best case - O(1)

Average case - O(log n)

Worst case - O(log n)

Merge Sort

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

Algorithm

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a
   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]
   l1 = mergesort( l1 )
   l2 = mergesort( l2 )
   return merge( l1, l2 )
end procedure
procedure merge( var a as array, var b as array )
   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
            add b[0] to the end of c
            remove b[0] from b
      else
            add a[0] to the end of c
            remove a[0] from a
      end if
   end while
   while ( a has elements )
            add a[0] to the end of c
            remove a[0] from a
   end while
   while ( b has elements )
            add b[0] to the end of c
            remove b[0] from b
   end while
```

Time Complexity for Merge sort

-Given an array of n elements, following are the time complexities for different cases :-

Best case - O(n log n)

Average case - O(n log n)

Worst case - O(n log n)

Quick Sort

Quicksort is a sorting algorithm based on the divide and conquer approach where

1. An array is divided into subarrays by selecting a pivot element (element selected from the array).

   While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.
2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.
3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Quick sort recursive Algorithm

```
quickSort(array, leftmostIndex, rightmostIndex)
  if (leftmostIndex < rightmostIndex)
       pivotIndex <- partition(array,leftmostIndex, rightmostIndex)
  quickSort(array, leftmostIndex, pivotIndex - 1)
  quickSort(array, pivotIndex, rightmostIndex)
```

Partition Algorithm

```
partition(array, leftmostIndex, rightmostIndex)
  set rightmostIndex as pivotIndex
  storeIndex <- leftmostIndex - 1
 for i <- leftmostIndex + 1 to rightmostIndex
    if element[i] < pivotElement
      swap element[i] and element[storeIndex]
   storeIndex++
 swap pivotElement and element[storeIndex+1]
return storeIndex + 1
```

Time Complexity for Quick Sort

Given an array of n elements, following are the time complexities for quick sort :-

Best case - O(n log n)

Average case - O(n log n)

Worst case - O(n^2)

Bubble Sort

Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water. Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.

Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world. It is not suitable for large data sets.

Algorithm

```
1.  begin BubbleSort(arr)
2.    for all array elements
3.      if arr[i] > arr[i+1]
4.        swap(arr[i], arr[i+1])
5.      end if
6.    end for
7.    return arr
8.  end BubbleSort
```

Time Complexity for Bubble sort

Consider an array of n elements and bubble sort is applied to it. Following are the time complexities in different scenarios :-

Best case - O(n) - When the array is already sorted.

Average case -

Worst case - O(n2)

Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position. After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

Algorithm

```
SELECTION SORT(arr, n)
    Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
    Step 2: CALL SMALLEST(arr, i, n, pos)
    Step 3: SWAP arr[i] with arr[pos]
    [END OF LOOP]
    Step 4: EXIT

SMALLEST (arr, i, n, pos)
    Step 1: [INITIALIZE] SET SMALL = arr[i]
    Step 2: [INITIALIZE] SET pos = i
    Step 3: Repeat for j = i+1 to n
    if (SMALL > arr[j])
    SET SMALL = arr[j]
    SET pos = j
    [END OF if]
```

17. [END OF LOOP]

18. Step 4: RETURN pos

Time complexity for Selection sort

Consider an array containing n elements. Following are different scenarios with their time complexities :-

Best case - O(n^2)

Average case - O(n^2)

Worst case - O(n^2)

Insertion Sort

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

The same approach is applied in insertion sort. The idea behind the insertion sort is that first take one element, iterate it through the sorted array.

Algorithm

The simple steps of achieving the insertion sort are listed as follows -

Step-1:

If the element is the first element, assume that it is already sorted. Return 1.

Step-2:

Pick the next element, and store it separately in a key.

Step-3:

Now, compare the key with all elements in the sorted array.

Step4:

If the element in the sorted array is smaller than the current element, then move to the next element. Else, shift greater elements in the array towards the right.

Step 5: Insert the value.

Step 6: Repeat until the array is sorted.

Time complexity for Insertion sort

Consider an array containing n elements. Following are different scenarios with their time complexities.

Best case - O(n)

Average case - O(n^2)

Worst case - O(n^2)

6.8 Stable and Inplace algorithms

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

An inplace algorithm is one which does not require an additional data structure, say array (apart from the data structure in which the input is stored). This additional data structure, if required, is part of the auxiliary space. Another way to say this is an inplace algorithm is one which has O(1) auxiliary space.

Algorithm	Stable	Inplace
Quick Sort	No	Yes
Merge Sort	Yes	No
Bubble Sort	Yes	Yes
Selection Sort	No	Yes
Insertion Sort	Yes	Yes

6.9 Symbol Table

A symbol table is a data structure employed by a language translator, like a compiler or interpreter, within which each identifier in a program's source code is connected with information about its declaration or presence in the source code, like its type, scope level, and infrequently its position.

A symbol table is a significant data structure used in a compiler that correlates characteristics with program identifiers. The analysis and synthesis stages employ symbol table information to verify that used identifiers have been specified, to validate that expressions and assignments are semantically accurate, and to build intermediate or target code.

A symbol table's basic operations are allocate, free, insert, lookup, set attribute and get attribute. The allocate operation assigns an empty symbol table. To remove all records and free the storage of a symbol table, free operation is used. The insert operation as the name suggests inserts a name in a symbol table and return a pointer to its entry. The lookup function searches for a name and returns a pointer to its entry. The set attribute and the get attribute associate an attribute with a given entry and get an attribute associated with a given respectively. Other procedures might be added based on the needs. A delete operation, for instance, removes a previously entered name.

Following are the data structures which can be used to implement a symbol table:

1. Linear List
2. Binary Search Tree
3. Hash Table

Items stored in Symbol table

- Variable names and constants
- Procedure and function names
- Literal constants and strings
- Compiler generated temporaries
- Labels in source languages

DATA STRUCTURES AND ALGORITHMS

Practice Questions

Q.1 Which of the following uses FIFO method ?
(a) Queue (b) Stack
(c) Hash Table (d) Binary Search Tree

Q.2 What data structure can be used to check if a syntax has balanced paranthesis ?
(a) queue (b) tree
(c) list (d) Stack

Q.3 If the array is already sorted, which of these algorithms will exhibit the best performance
(a) Merge Sort (b) Insertion Sort
(c) Quick Sort (d) Heap Sort

Q.4 Apriori algorithm analysis does not include
(a) Time Complexity
(b) Space Complexity
(c) Program Complexity
(d) None of the above

Q.5 Which of the below mentioned sorting algorithms are not stable?
(a) Selection Sort (b) Bubble Sort
(c) Merge Sort (d) Insertion Sort

Q.6 Tower of Hanoi is a classic example of which programming paradigm ?
(a) Divide And Conquer
(b) Greedy Algorithm
(c) Dynamic Programming
(d) Randomized Algorithms

Q.7 Re-balancing of AVL tree costs
(a) (1) (b) (log n)
(c) (n) (d) (n^2)

Q.8 The worst case time complexity of AVL tree is better in comparison to binary search tree for
(a) Search and insert operations
(b) Search and delete operations
(c) Insert and delete operations
(d) Search, insert and delete operations

Q.9 Given an empty stack, after performing push(1), push(2), pop, push(3), push(4), pop, pop, push(5), pop. What is the value of the top of the stack?
(a) 4 (b) 3
(c) 2 (d) 1

Q.10 Given two sorted lists of size 'm' and 'n' respectively. The number of comparisons needed in the worst case by the merge sort algorithm will be
(a) m*n (b) max(m, n)
(c) min(m, n) (d) m + n - 1

Q.11 Find the number of minimum comparisons required in the worst case to find both the minimum and the maximum value among n elements in an array.
(a) 2n - 2 (b) n - 1
(c) floor(3n/2) - 2 (d) 2n - log n

Q.12 You have to sort a list L, consisting of a sorted list followed by a few 'random' elements. Which of the following sorting method would be most suitable for such a task ?
(a) Bubble sort (b) Selection sort
(c) Quick sort (d) Insertion sort

Q.13 Consider an array A[20. 10], assume 4 words per memory cell and the base address of array A is 100. What is the address of A[11,5]? Assume row major storage.
(a) 560 (b) 565
(c) 570 (d) 575

Q.42 Which of the following statements is true?

1. As the number of entries in a hash table increases, the number of collisions increases.
2. Recursive programs are efficient.
3. The worst case complexity for Quicksort is $O(n^2)$
4. Binary search using a linear linked list is efficient.

(a) 1 and 2 (b) 2 and 3
(c) 1 and 4 (d) 1 and 3

Q.43 The concept of order (Big O) is important because

(a) It can be used to decide the best algorithm that solves a given problem
(b) It determines the maximum size of a problem that can be solved in a given amount of time
(c) It is the lower bound of the growth rate of algorithm
(d) Both (a) and (b).

Q.44 Which of the following is false?

(a) $100n\log n = O\left(\frac{n\log n}{100}\right)$

(b) $\sqrt{(\log n)} = O(\log\log n)$

(c) If $0 < x < y$ then $n^x = O(n^y)$
(d) $2n \neq O(nk)$

Q.45 The concatenation of two lists is to be performed in O(1) time. Which of the following implementations of a list should be used?

(a) Singly linked list
(b) Doubly linked list
(c) Circular doubly linked list
(d) Array implementation of list

Q.46 Consider the following three claims:

1. $(n + k)m = \Theta(n^m)$, where k and m are constants.
2. $2^{n+1} = O(2^n)$
3. $2^{2n+1} = O(2^n)$

Which of these claims are correct?
(a) 1 and 2 (b) 1 and 3
(c) 2 and 3 (d) 1, 2 and 3

Q.47 The tightest lower bound on the number of comparisons, in the worst case, for comparison based sorting is of the order of

(a) n (b) n^2
(c) n log n (d) n log2 n

Q.48 The time complexity of the following C function is (assume n > 0)

```
int recursive (int n)
{
 if(n= = 1)
   return (1);
 else
    return(recursive(n-1)+ recursive(n-1));
```

(a) O(n) (b) O(n log n)
(c) $O(n^2)$ (d) $O(2^n)$

Q.49 The time complexity of computing the transitive closure of a binary relation on a set of n elements is known to be

(a) O(n) (b) O(niogn)
(c) $O(n^{3/2})$ (d) $O(n^3)$

Q.50 The minimum number of comparison required to determine if an integer appears more than n/2 times in a sorted array of n integer is

(a) Θ(n) (b) Θ(logn)
(c) $\Theta(\log^2 n)$ (d) Θ(1)

ANSWER KEY ▶ DATA STRUCTURES AND ALGORITHMS

1. (a)	2. (d)	3. (b)	4. (c)	5. (a)	6. (a)	7. (b)	8. (d)
9. (d)	10. (d)	11. (c)	12. (d)	13. (a)	14. (a)	15. (a)	16. (b)
17. (b)	18. (b)	19. (b)	20. (b)	21. (b)	22. (a)	23. (b)	24. (b)
25. (c)	26. (c)	27. (d)	28. (b)	29. (c)	30. (d)	31. (b)	32. (b)
33. (a)	34. (b)	35. (a)	36. (d)	37. (d)	38. (c)	39. (d)	40 (b)
41. (b)	42. (d)	43. (a)	44. (b)	45. (c)	46. (a)	47. (c)	48. (d)
49. (d)	50. (b						