Question Bank

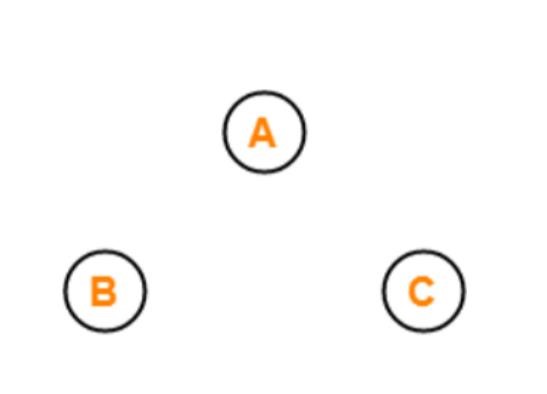
IAT-2-DS

**Q1) Explain graph terminology and different types of graphs with proper diagrams.**

1. Path: A path is the sequence of nodes that is followed to reach some terminal vertex X from the initial vertex Y.
2. Closed path: A path is a closed path if the initial vertex is the same as the terminal vertex.
3. Simple path: If all the vertices in the graph are distinct (exception V0=Vn), then path P is a closed simple path.
4. Cycle: A cycle is a path that has no repeated edges or vertices except the first and last vertices.
5. Connected graph: In a connected graph each pair of nodes has a path between them i.e. we can reach any vertex by starting from any vertex in the graph. There are no isolated nodes in the connected graph.
6. Complete graph: A complete graph is the one in which each pair of nodes has a direct path between them. A complete graph has n(n-1)/2 edges where n is the number of vertices in the graph.
7. Weighted graph: In a weighted graph, each edge is assigned with a data called weight. The weight of an edge E is given as W(E). Weight must be a positive value that defines the cost of traversing the edge R.
8. Digraph: A directed graph is sometimes referred to as a digraph.
9. Loop: An edge that has the same endpoints is a loop.
10. Adjacent nodes: If two nodes A and B are directly connected via an edge E, then the vertices A and B are adjacent nodes.
11. Degree of the node: A degree of a node is the number of edges associated with that node. A node with degree 0 is an isolated node.

* Null Graph

The Null Graph is also known as the order zero graph. The term “null graph” refers to a graph with an empty edge set. In other words, a null graph has no edges, and the null graph is present with only isolated vertices in the graph.



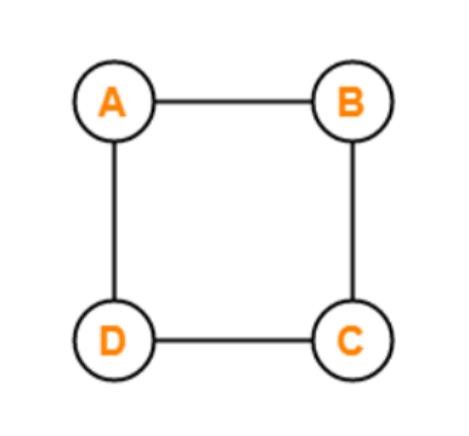
* Trivial Graph

A graph is called a trivial graph if it has only one vertex present in it. The trivial graph is the smallest possible graph that can be created with the least number of vertices that is one vertex only.



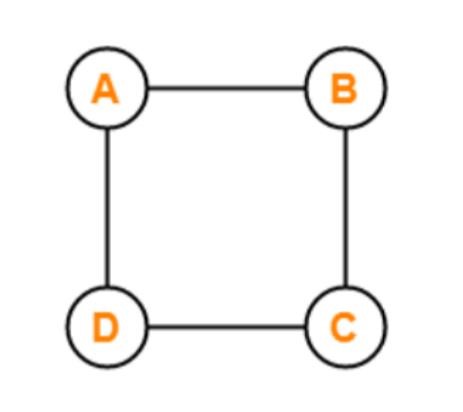
• Non-Directed Graph

A graph is called a non-directed graph if all the edges present between any graph nodes are non-directed. By non-directed edges, we mean the edges of the graph that cannot be determined from the node it is starting and at which node it is ending. All the edges for a graph need to be non-directed to call it a non-directed graph. All the edges of a non-directed graph don’t have any direction.



* Directed Graph

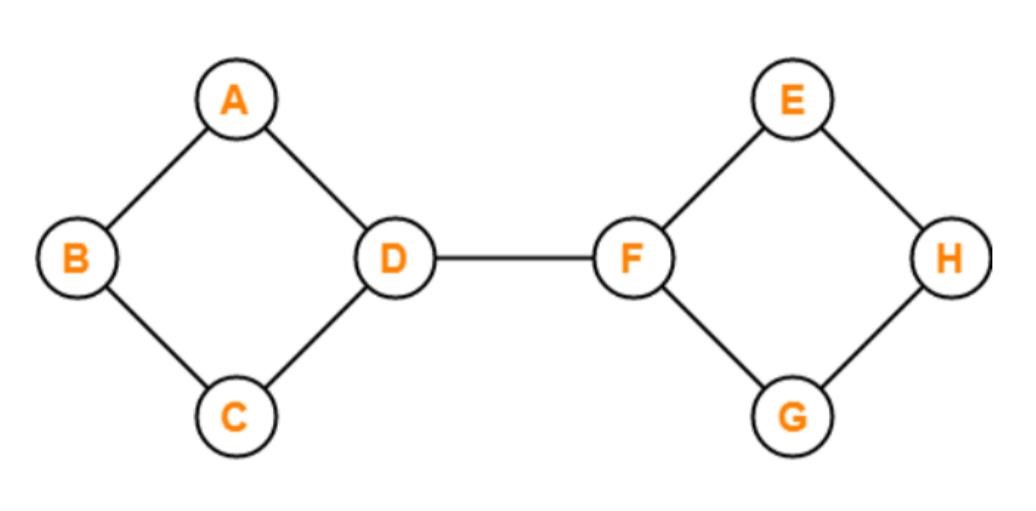
Another name for the directed graphs is digraphs. A graph is called a directed graph or digraph if all the edges present between any vertices or nodes of the graph are directed or have a defined direction. By directed edges, we mean the edges of the graph that have a direction to determine from which node it is starting and at which node it is ending.



* Connected Graph

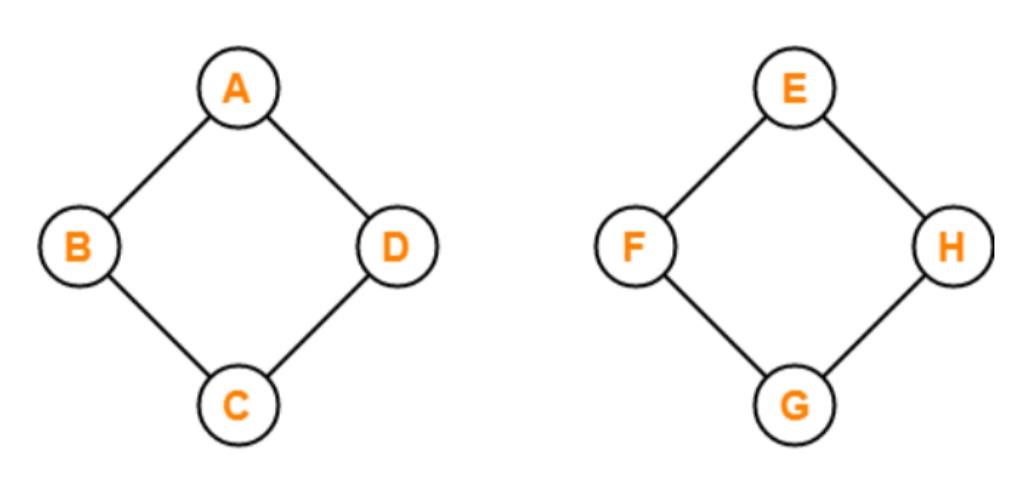
For a graph to be labelled as a connected graph, there must be at least a

single path between every pair of the graph’s vertices. In other words, we can say that if we start from one vertex, we should be able to move to any of the vertices that are present in that particular graph, which means there exists at least one path between all the vertices of the graph.



* Disconnected Graph

A graph is said to be a disconnected graph where there does not exist any path between at least one pair of vertices. In other words, we can say that if we start from any one of the vertices of the graph and try to move to the

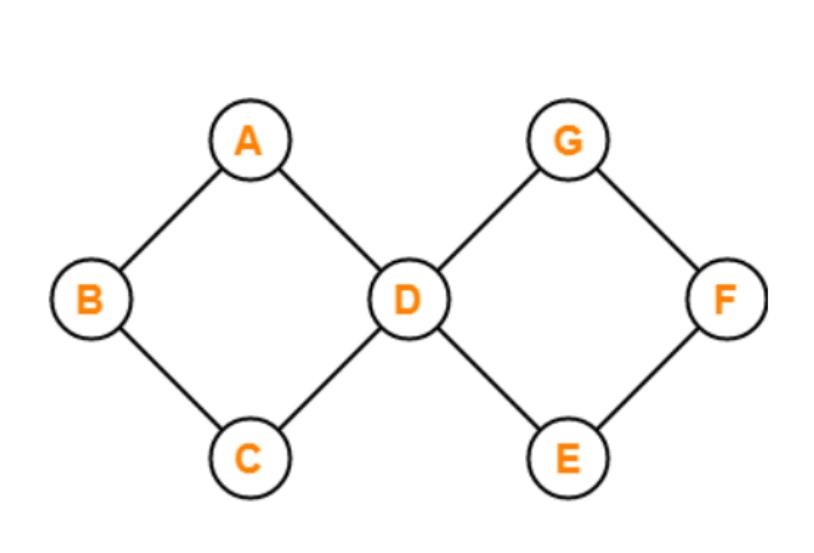
remaining present vertices of the graph and there exists not even a single path to move to that vertex, then it is the case of the disconnected graph. If

any one of such a pair of vertices doesn’t have a path between them, it is called a disconnected graph.

* Cyclic Graph

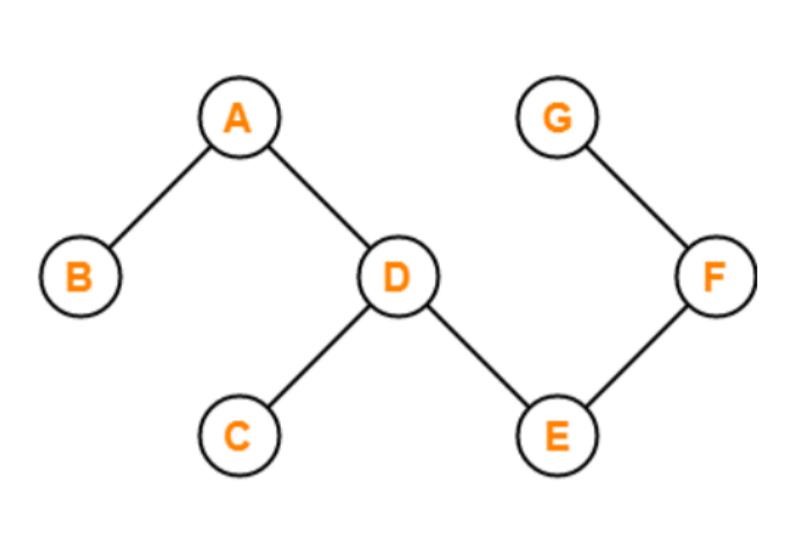
For a graph to be called a cyclic graph, it should consist of at least one cycle.

If a graph has a minimum of one cycle present, it is called a cyclic graph.



* Acyclic Graph

A graph is called an acyclic graph if zero cycles are present, and an acyclic graph is the complete opposite of a cyclic graph.

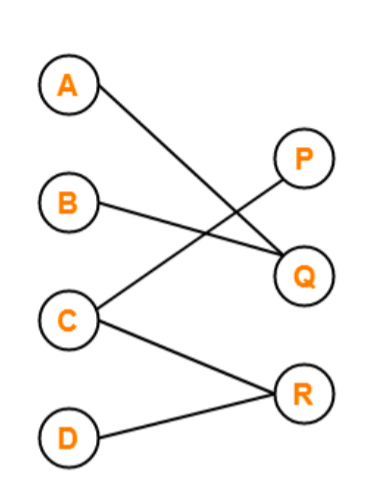


• Bipartite Graph

For a graph to be a Bipartite graph, it needs to satisfy some of the basic preconditions.

These conditions are:

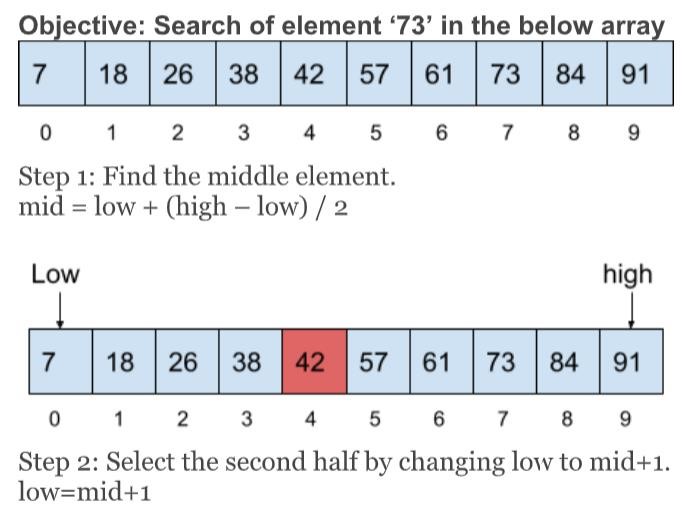
1. All the vertices of the graph should be divided into two distinct sets of vertices X and Y.
2. All the vertices present in the set X should only be connected to the vertices present in the set Y with some edges. That means the vertices present in a set should not be connected to the vertex that is present in the same set.
3. Both the sets that are created should be distinct that means both should not have the same vertices in them.

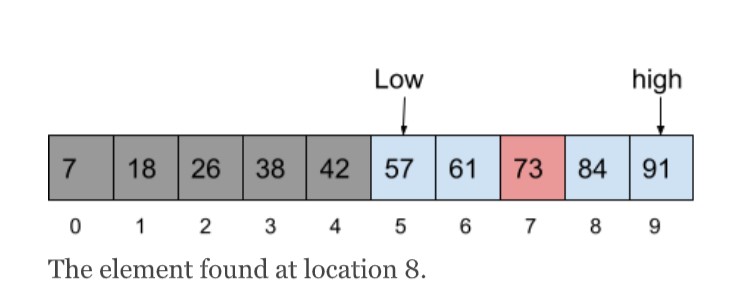
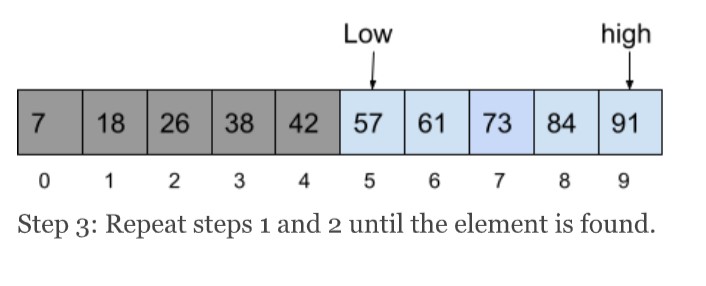


**Q2) Explain binary search with example.**

Binary search is an efficient searching algorithm with only the requirement that the elements in the array are sorted already. It is a great implementation of the divide and conquer algorithm technique. It divides the array at the middle and then selects the first or the second half based on the search element and then repeats the process with the selected half until the search element is found (or not).

1. Works on the Divide and conquer technique to search for the elements.
2. Efficient than linear search algorithm.





• Algorithm for Binary Searching

Step 1: set low = starting\_index, high = last\_index, loc = - 1

Step 2: repeat steps 3 and 4 while low <= high

Step 3: set mid = (low +high)/2

Step 4: if Arr[mid] = se

set loc=mid

print “ search element is present at location: “ +loc

go to step 6

else if Arr[mid] < se

set low = mid + 1

else set high = mid - 1

[end of if]

[end of loop]

step 5: if loc = -1

print “search element is not present in the array”

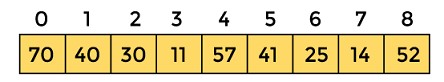
[end of if]

step 6: Stop

**Q3) Explain linear search with example.**

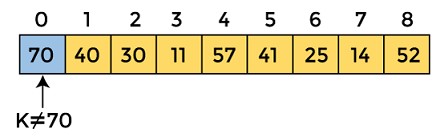
Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.Linear search is also called as sequential search algorithm. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL.

Let the elements of array are -

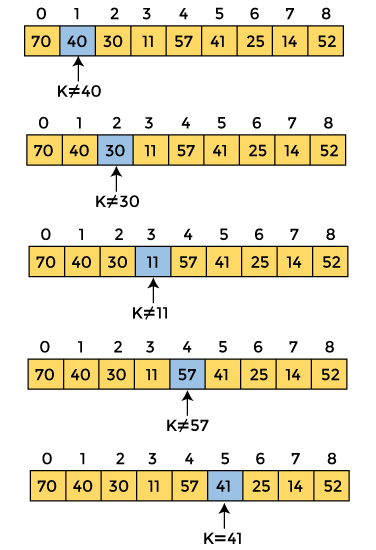


Let the element to be searched is **K = 41**

Now, start from the first element and compare **K** with each element of the array.



The value of **K,** i.e., **41,** is not matched with the first element of the array. So, move to the next element. And follow the same process until the respective element is found.



Now, the element to be searched is found. So algorithm will return the index of the element matched.

• Program

#include <stdio.h> **int** linearSearch(**int** a[], **int** n, **int** val) { // Going through array sequencially **for** (**int** i = 0; i < n; i++)

{**if** (a[i] == val) **return** i+1;

}

**return** -1;

}

**int** main() {

**int** a[] = {70, 40, 30, 11, 57, 41, 25, 14, 52}; // given array **int** val = 41; // value to be searched

**int** n = **sizeof**(a) / **sizeof**(a[0]); // size of array

**int** res = linearSearch(a, n, val); // Store result printf("The elements of the array are - ");

**for** (**int** i = 0; i < n; i++)

printf("%d ", a[i]);

printf("\nElement to be searched is - %d", val);

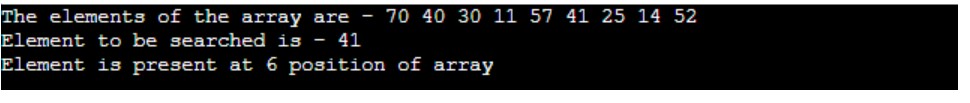
**if** (res == -1)

printf("\nElement is not present in the array");

**else**

printf("\nElement is present at %d position of array", res); **return** 0;

}



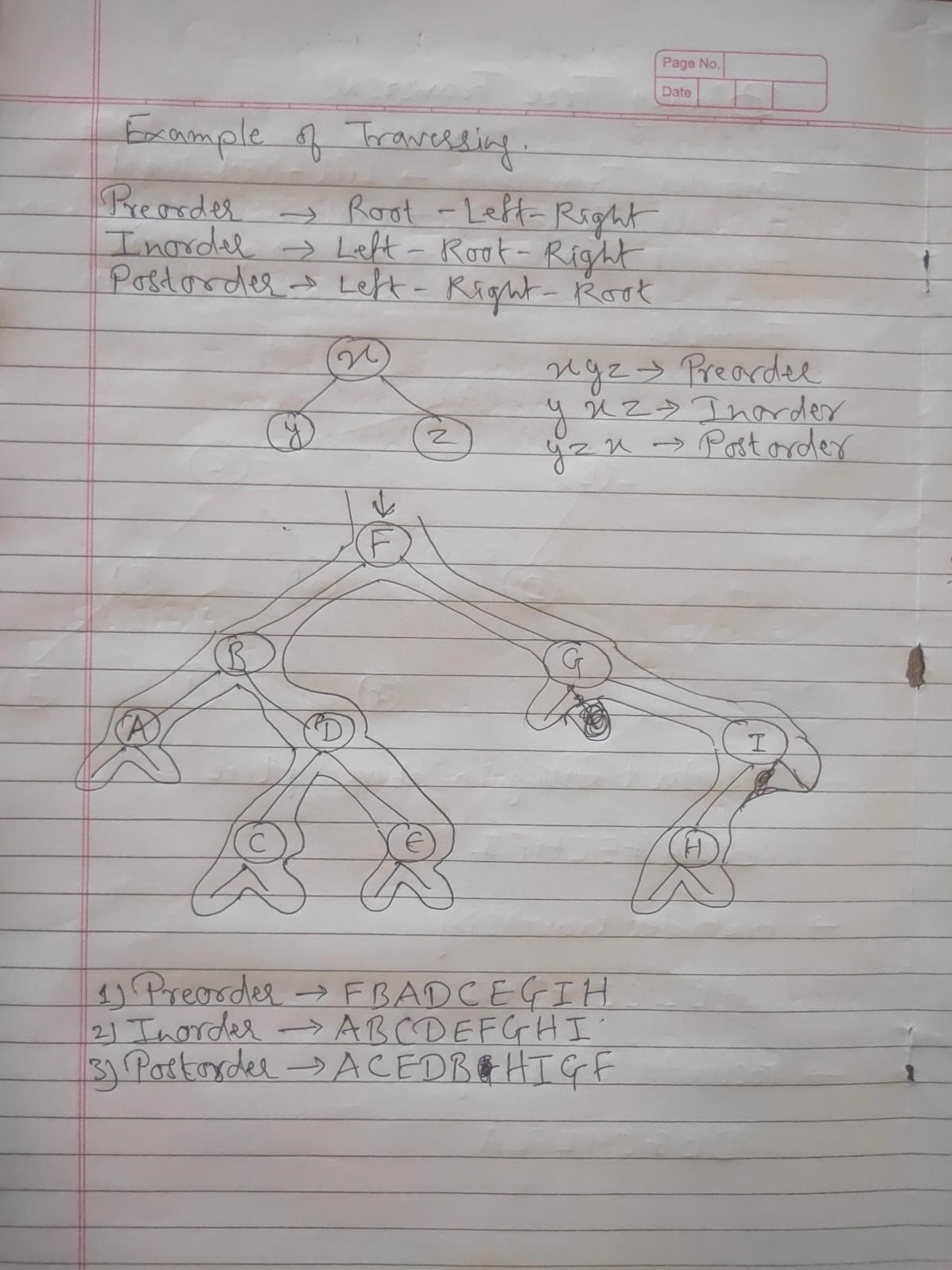
**Q4) Explain Topological sorting in detail.**

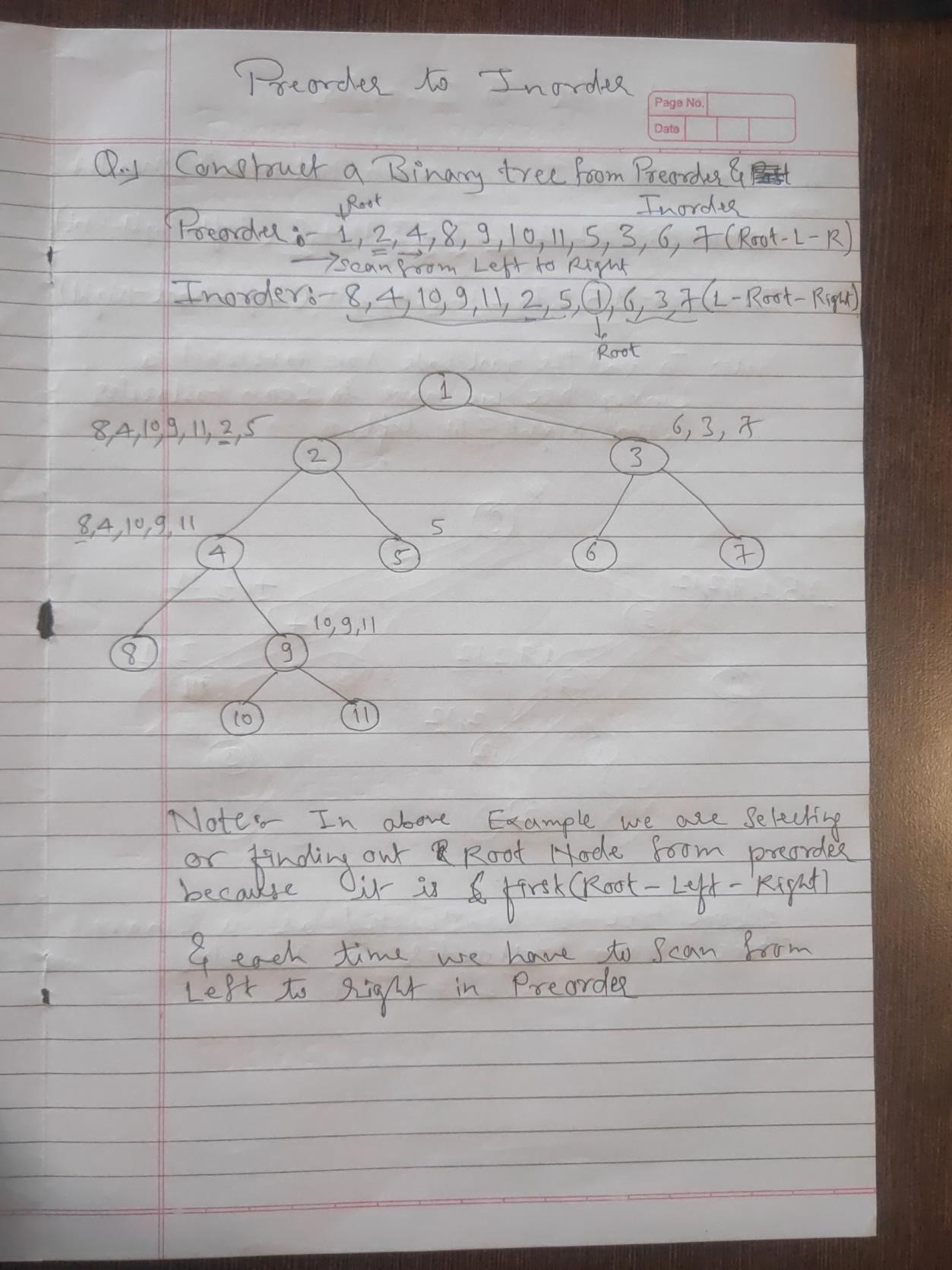
A topological sort or topological ordering of a directed graph is a linear ordering of its vertices in which u occurs before v in the ordering for every directed edge uv from vertex u to vertex v. For example, the graph's vertices could represent jobs to be completed, and the edges could reflect requirements that one work must be completed before another. In this case, a topological ordering is just a legitimate task sequence. A topological sort is a graph traversal in which each node v is only visited after all of its dependencies have been visited. If the graph contains no directed cycles, then it is a directed acyclic graph. Any DAG has at least one topological ordering, and there exist techniques for building topological orderings in linear time for any DAG. Topological sorting has many applications, particularly in ranking issues like the feedback arc set. Even if the DAG includes disconnected components, topological sorting is possible. Topological Sorting is mostly used to schedule jobs based on their dependencies. Instruction scheduling, ordering formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linker are all examples of applications of this type in computer science.

Q5) Write Comparison of B tree and B+ tree in detail.

| Basis of Comparision | B tree | B+ tree |
| --- | --- | --- |
| **Pointers** | All internal and leaf nodes have data pointers | Only leaf nodes have data pointers |
| **Search** | Since all keys are not available at leaf, search often takes more time. | All keys are at leaf nodes, hence search is faster and more accurate. |
| **Redundant Keys** | No duplicate of keys is maintained in the tree. | Duplicate of keys are maintained and all nodes are present at the leaf. |
| **Insertion** | Insertion takes more time and it is not predictable sometimes. | Insertion is easier and the results are always the same. |
| **Deletion** | Deletion of the internal node is very complex and the tree has to undergo a lot of transformations. | Deletion of any node is easy because all node are found at leaf. |
| **Leaf Nodes** | Leaf nodes are not stored as structural linked list. | Leaf nodes are stored as structural linked list. |
| **Access** | Sequential access to nodes is not possible | Sequential access is possible just like linked list |
| **Height** | For a particular number nodes height is larger | Height is lesser than B tree for the same number of nodes |
| **Application** | B-Trees used in Databases, Search engines | B+ Trees used in Multilevel Indexing, Database indexing |
| **Number of Nodes** | Number of nodes at any intermediary level ‘l’ is 2l. | Each intermediary node can have n/2 to n children. |

**Q6) Construct a binary search tree from preorder and in order with suitable example**





**Q7) Explain AVL tree with one example**

An AVL tree is a type of binary search tree. Named after it's inventors Adelson, Velskii, and Landis, AVL trees have the property of dynamic self-balancing in addition to all the other properties exhibited by binary search trees.

• AVL trees have an additional guarantee:

1. The difference between the depth of right and left sub-trees cannot be more than one. This difference is called the balance factor.

2. In order to maintain this guarantee, an implementation of an AVL will include an algorithm to rebalance the tree when adding an additional element would upset this guarantee

AVL trees have a worst case lookup, insert, and delete time of O(log n), where n is the number of nodes in the tree. The worst case space complexity is O(n).

• Operations on AVL Tree

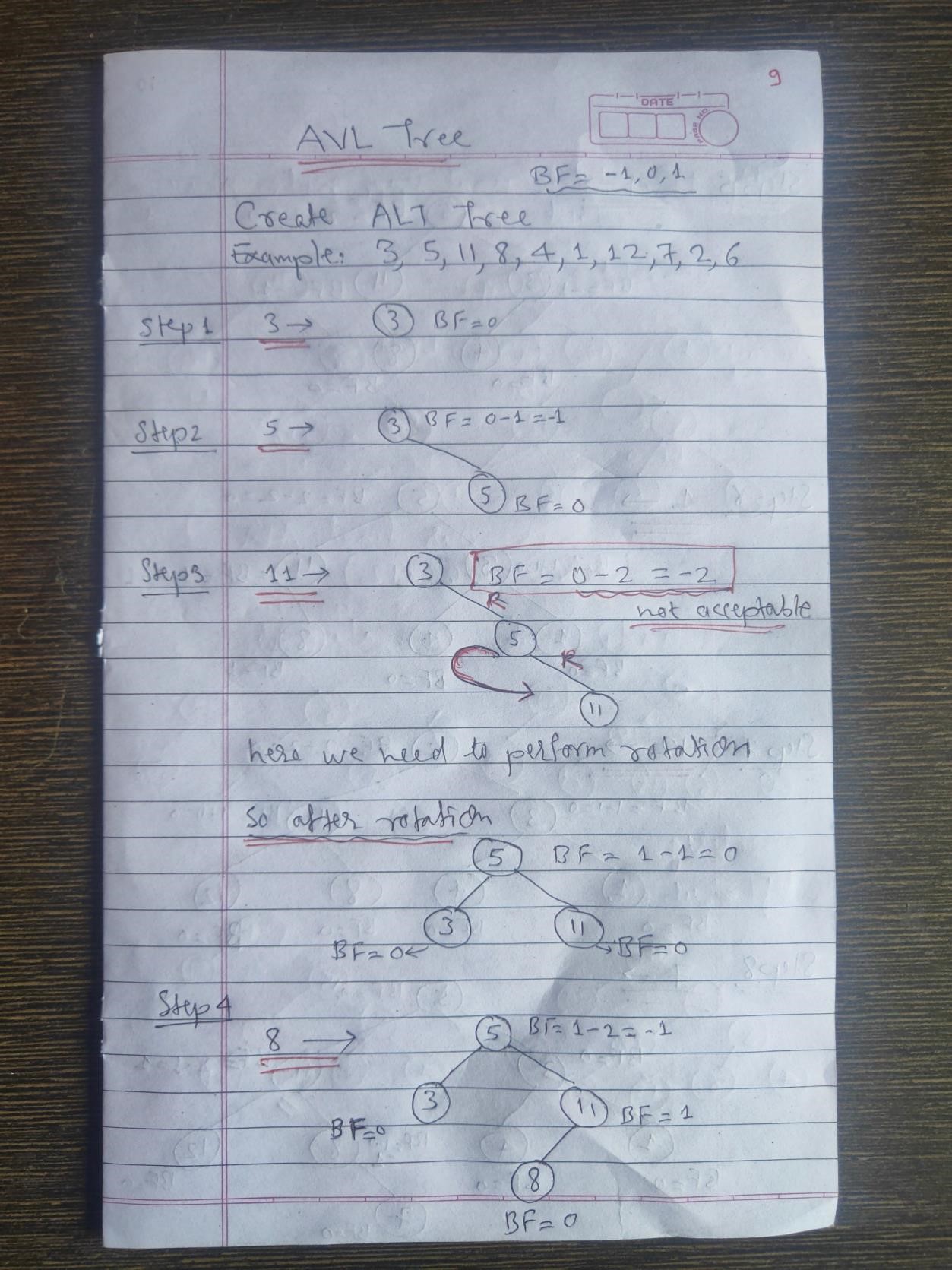
The AVL tree is a balancing binary tree, and therefore it follows the same operations we perform in the binary search tree.

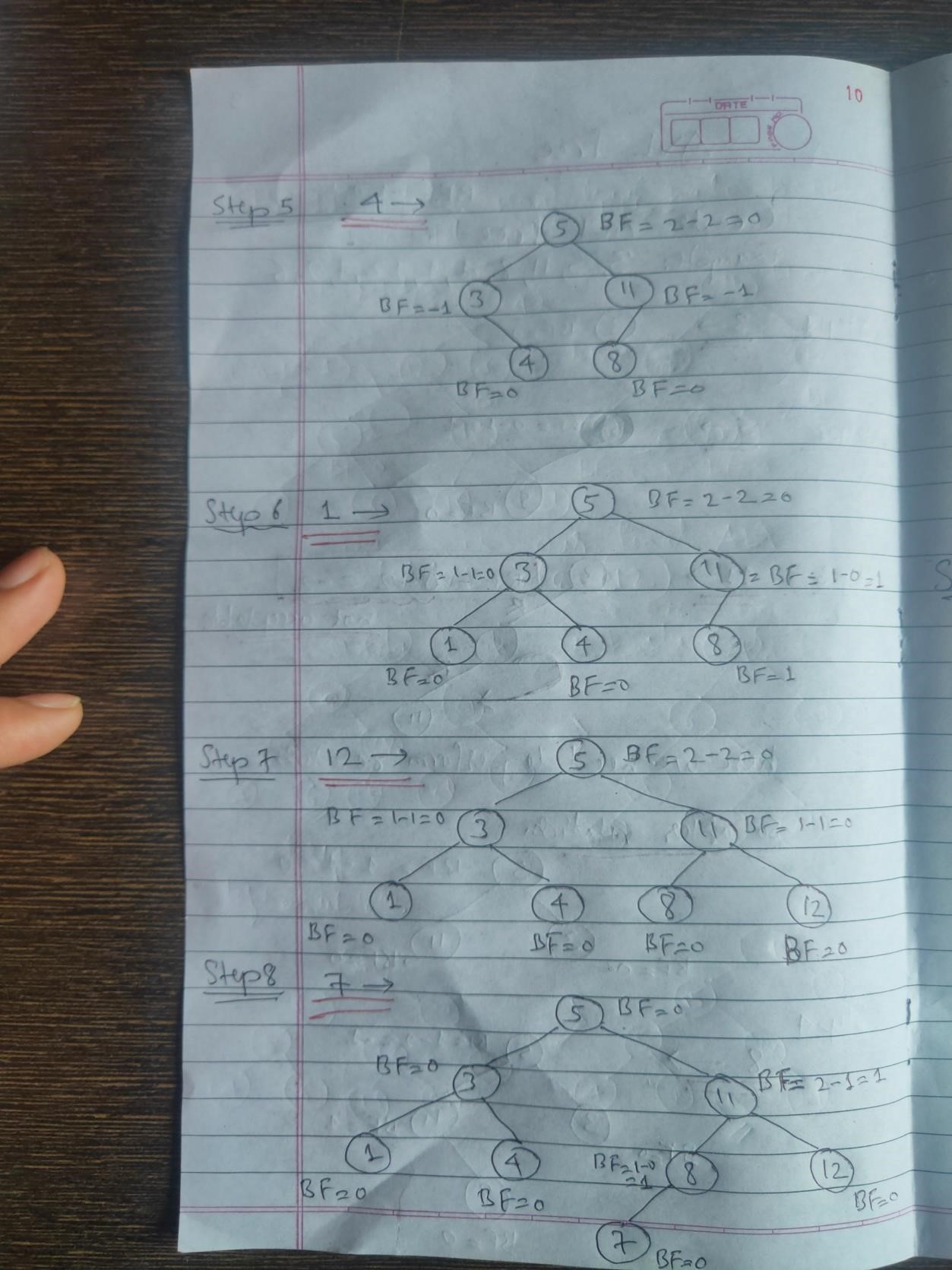
1. Insertion

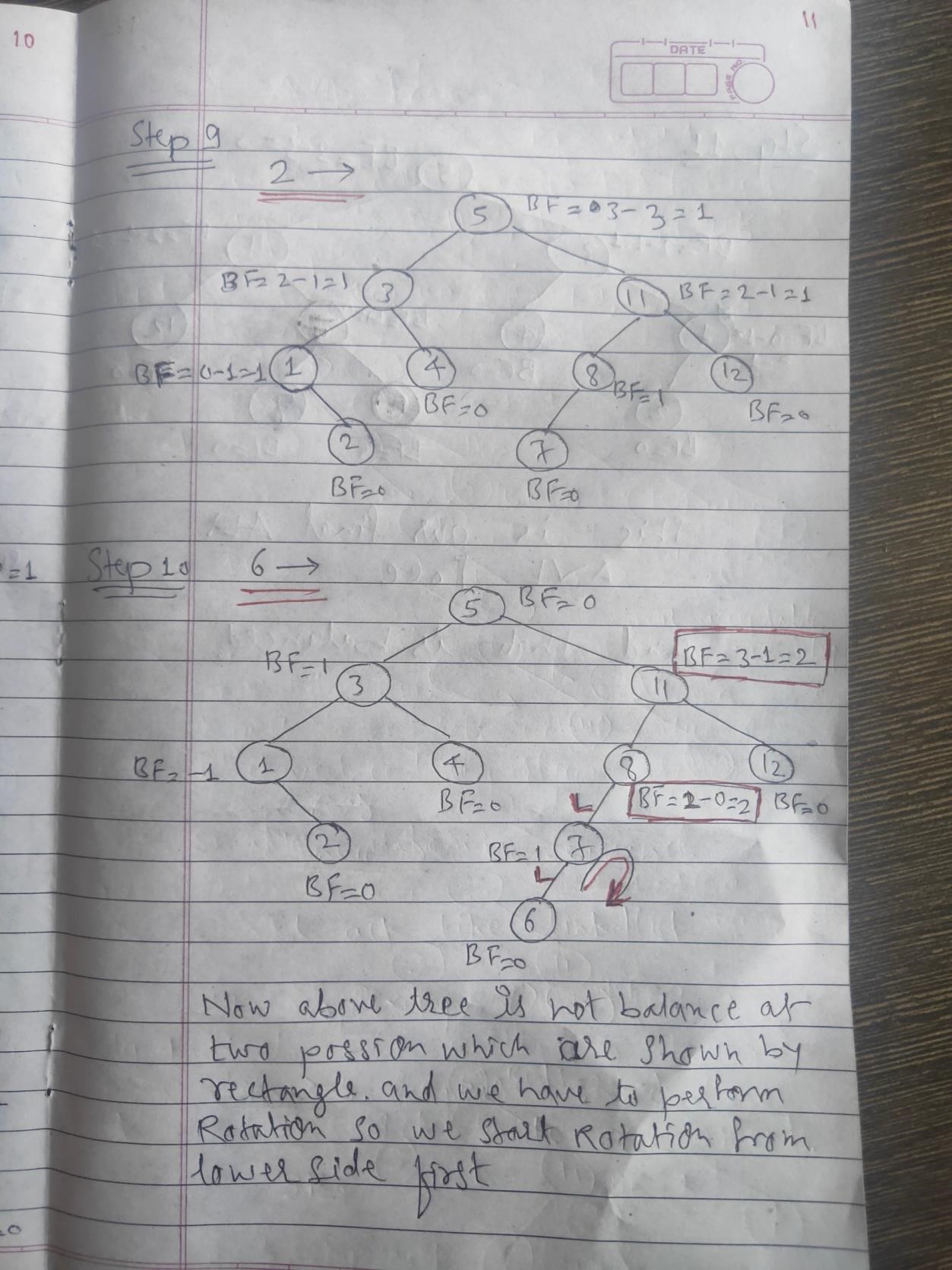
2. Deletion

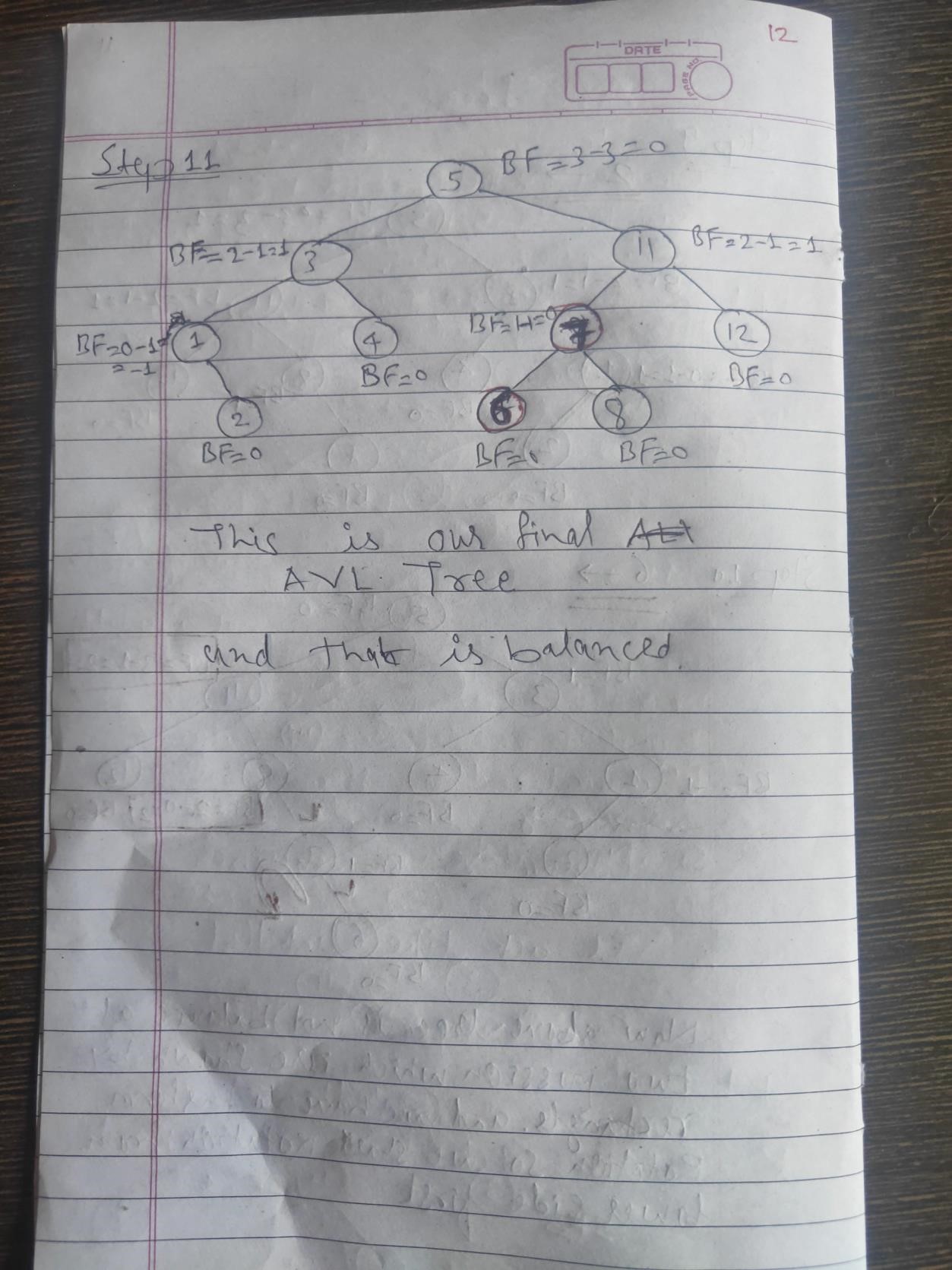
1. Insertion: The process of insertion is the same as it is executed in the binary search tree. However, there are chances that it may point to a violation in the AVL tree property, and the tree may require balancing. To balance a tree we can apply rotations.

2. Deletion: The process of deletion is the same as it is executed in a binary search tree. It can affect the balance factor of the tree, therefore, we need to utilize different types of rotations to balance the tree.









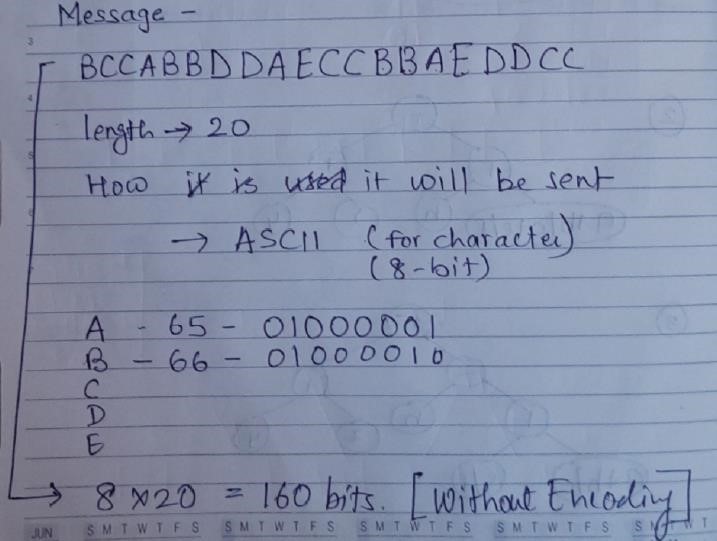
**Q8) Explain Huffman encoding with example.**

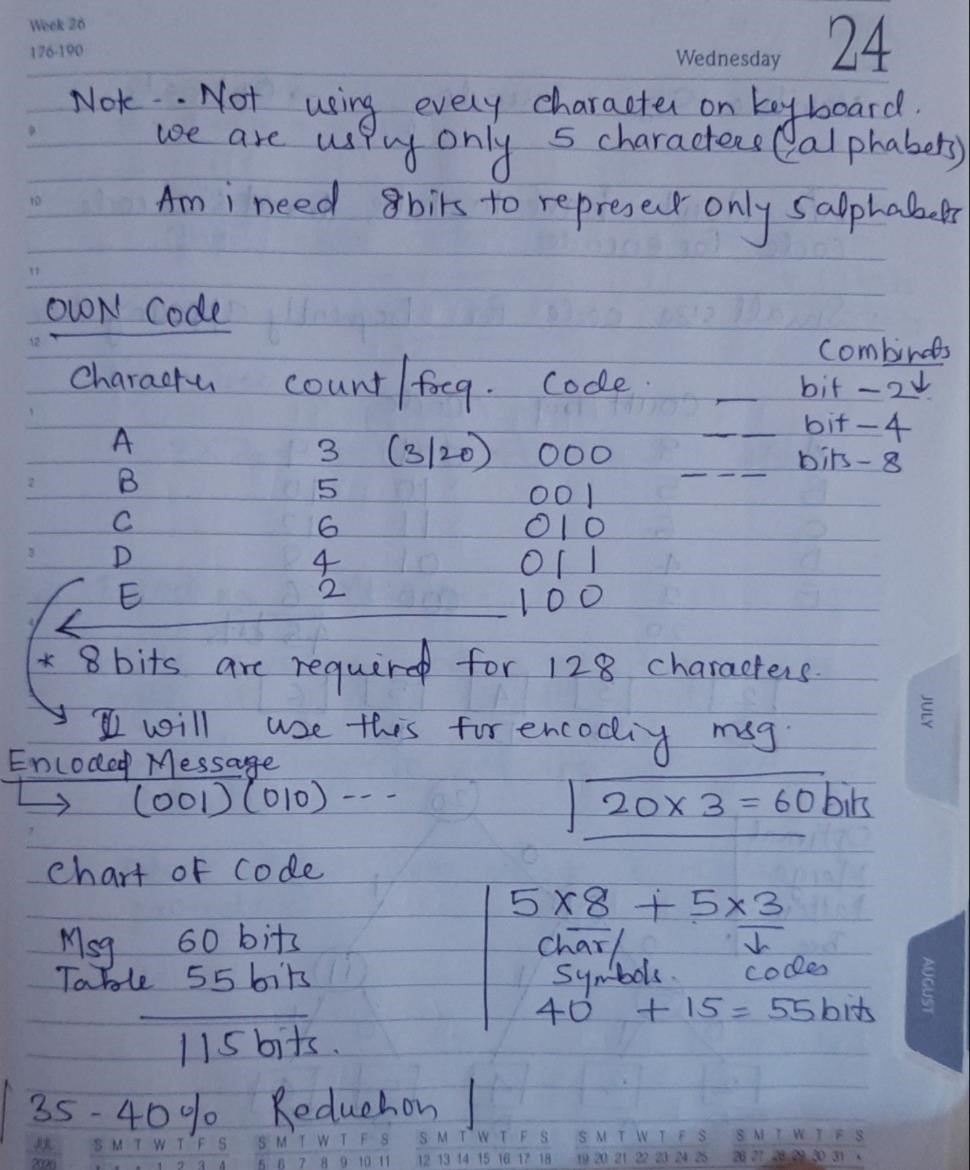
Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

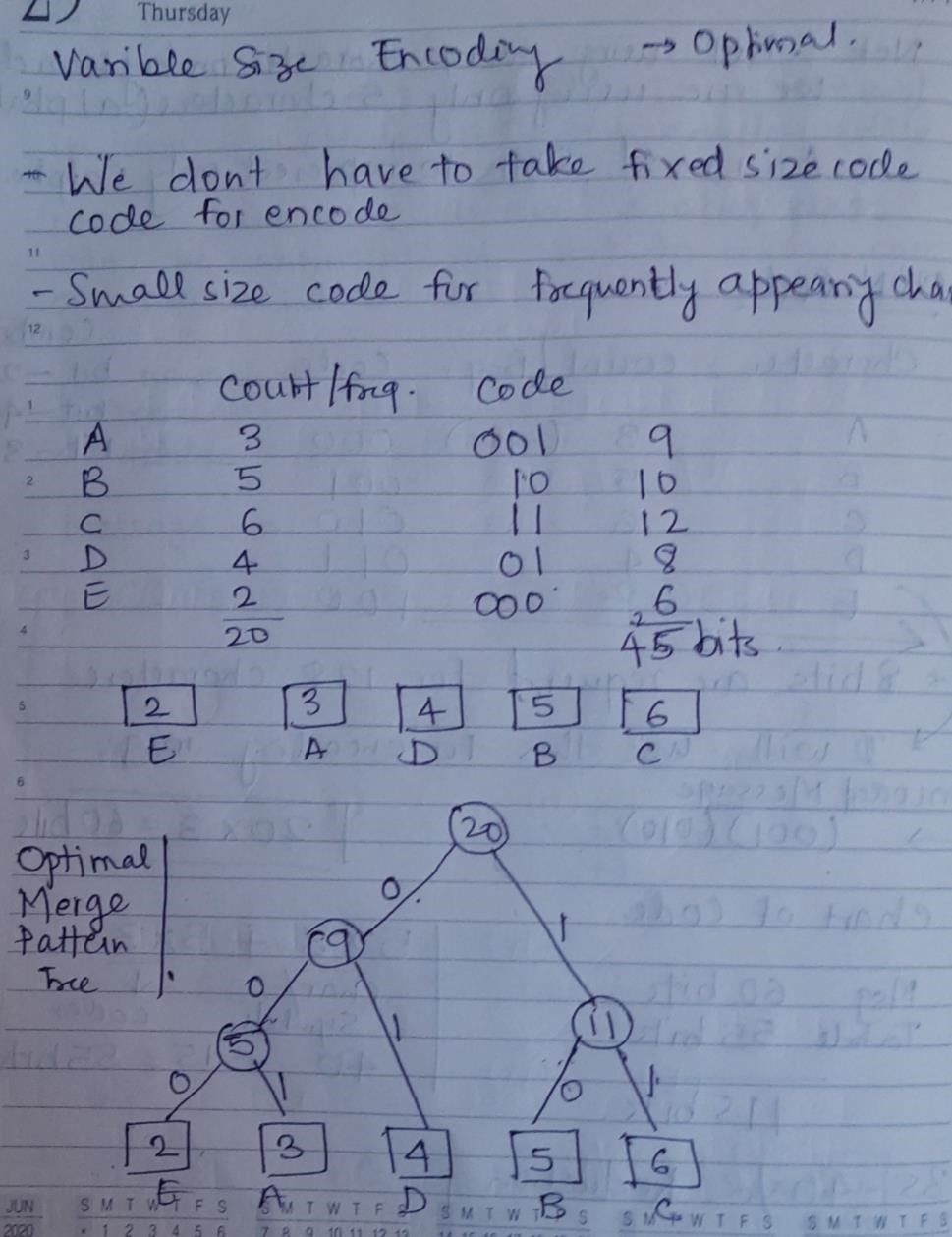
The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

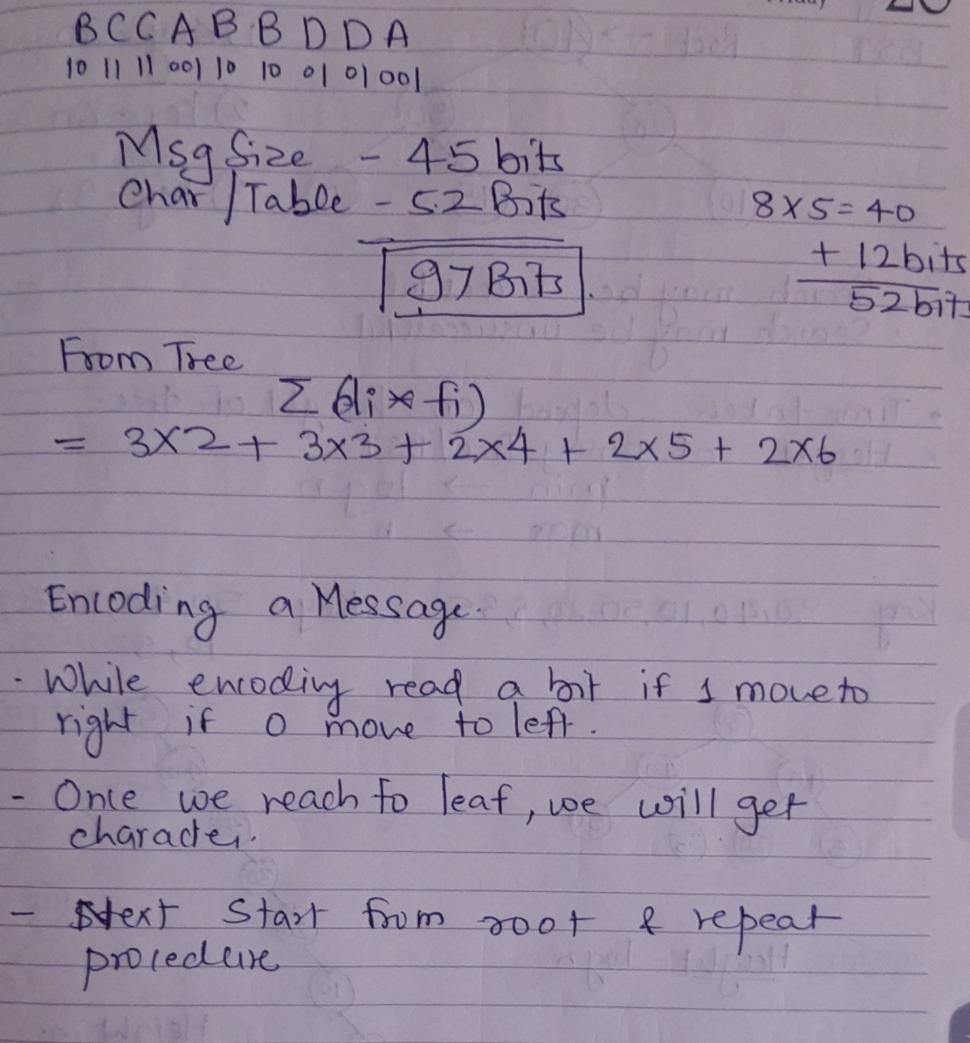
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the decompressed output may be “cccd” or “ccb” or “acd” or “ab

• Example









**Q9) Explain linear searching in detail.**

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.Linear search is also called as sequential search algorithm. It is the simplest searching algorithm. In Linear search, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match is found, then the location of the item is returned; otherwise, the algorithm returns NULL

The steps used in the implementation of Linear Search are listed as follows -

1. First, we have to traverse the array elements using a for loop.
2. In each iteration of for loop, compare the search element with the current array element, and -
3. If the element matches, then return the index of the corresponding array element.
4. If the element does not match, then move to the next element.
5. If there is no match or the search element is not present in the given array, return -1.

• Linear Search complexity

* Best Case Complexity **-** In Linear search, best case occurs when the element we are finding is at the first position of the array. The best-case time complexity of linear search is **O(1).**
* Average Case Complexity **-** The average case time complexity of linear search is **O(n).**
* Worst Case Complexity **-** In Linear search, the worst case occurs when the element we are looking is present at the end of the array. The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array. The worst-case time complexity of linear search is **O(n).**