1.Draw Binary tree from given traversal [5]

inorder : 4, 2, 5, 1, 6, 7, 3, 8

Postorder : 4, 5, 2, 6, 7, 8, 3, 1

Ans:

To construct a binary tree from its inorder and postorder traversals, is as follows:

1. The last element in the postorder traversal is the root of the tree.
2. Find the root element in the inorder traversal to determine the left and right subtrees.
3. Recursively build the right subtree using elements to the right of the root in the inorder traversal and the corresponding elements in the postorder traversal.
4. Recursively build the left subtree using elements to the left of the root in the inorder traversal and the corresponding elements in the postorder traversal.

Let's construct the binary tree from the given inorder and postorder traversals:

Inorder: 4, 2, 5, 1, 6, 7, 3, 8

Postorder: 4, 5, 2, 6, 7, 8, 3, 1

Step 1: The last element in postorder (1) is the root of the tree.

Step 2: In the inorder traversal, find the root (1) to determine the left and right subtrees.

Inorder Left Subtree: 4, 2, 5

Inorder Right Subtree: 6, 7, 3, 8

Step 3: Recursively build the right subtree using elements to the right of the root in inorder and corresponding elements in postorder.

Inorder Right Subtree: 6, 7, 3, 8

Postorder Right Subtree: 6, 7, 8, 3

The last element in postorder Right Subtree (3) is the root of the right subtree.

Inorder Right Subtree (Left of 3): 6, 7

Inorder Right Subtree (Right of 3): 8

Postorder Right Subtree (Left of 3): 6, 7, 8

Postorder Right Subtree (Right of 3): 3

Continue recursively to build the right subtree.

The last element in postorder Right Subtree (8) is the root of the right subtree.

Inorder Right Subtree (Left of 8): 6, 7

Inorder Right Subtree (Right of 8): (empty)

Postorder Right Subtree (Left of 8): 6, 7

Postorder Right Subtree (Right of 8): (empty)

Continue recursively to build the right subtree (empty).

Step 4: Recursively build the left subtree using elements to the left of the root in inorder and corresponding elements in postorder.

Inorder Left Subtree: 4, 2, 5

Postorder Left Subtree: 4, 5, 2

The last element in postorder Left Subtree (2) is the root of the left subtree.

Inorder Left Subtree (Left of 2): 4

Inorder Left Subtree (Right of 2): 5

Postorder Left Subtree (Left of 2): 4, 5

Postorder Left Subtree (Right of 2): (empty)

Continue recursively to build the left subtree.

The binary tree is constructed as follows:

2.Define collisions. **[2]**

Ans: A collision occurs when more than one value to be hashed by a particular hash function hash to the same slot in the table or data structure (hash table) being generated by the hash function.

3.Apply the algorithm to draw Binary search tree for following data. 10, 08, 15, 12, 13, 07, 09, 17, 20, 18, 04, 05{VeryIMP}
[5]

Ans:

1. Start with an empty BST.
2. Insert each element from the list into the BST one by one, maintaining the BST property.
3. Here's how you can construct a BST from the given data:

4. Data: 10, 08, 15, 12, 13, 07, 09, 17, 20, 18, 04, 05

5. Start with an empty BST:

Insert 10:

Insert 08:

Insert 15:

Insert 12:

Insert 13:

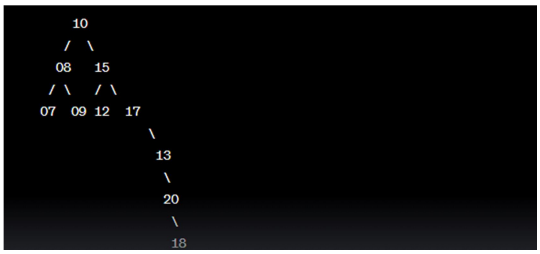
Insert 07:

Insert 09:

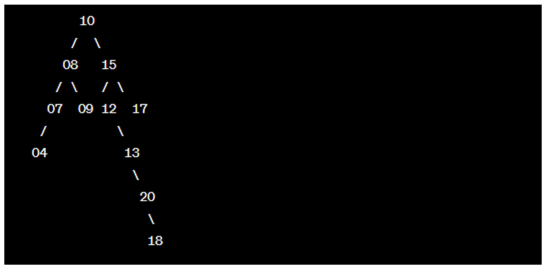
Insert 17:

Insert 20:

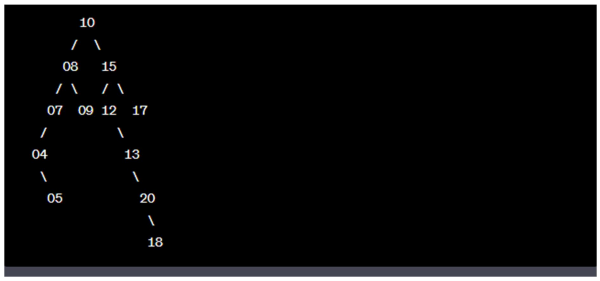
Insert 18:



Insert 04:



Insert 05:



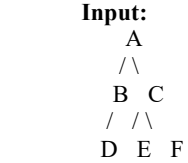
4.Compare BFS and DFS. {VeryIMP} **[3]**

Ans: Breadth-First Search:

BFS, Breadth-First Search, is a vertex-based technique for finding the shortest path in the graph. It uses a Queue data structure that follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.

Example:

Input:



Output:

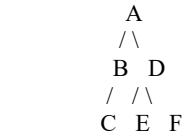
A, B, C, D, E, F

Depth First Search:

DFS, Depth First Search, is an edge-based technique. It uses the Stack data structure and performs two stages, first visited vertices are pushed into the stack, and second if there are no vertices then visited vertices are popped.

Example:

Input:



Output:

A, B, C, D, E, F

BFS vs DFS

S. No.	Parameters	BFS	DFS
1.	Stands for	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	Data Structure	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	Definition	BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level.	DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.
4.	Technique	BFS can be used to find a single source shortest path in an unweighted graph because, in BFS, we reach a vertex with a minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
5.	Conceptual Difference	BFS builds the tree level by level.	DFS builds the tree sub-tree by sub-tree.
6.	Approach used	It works on the concept of FIFO (First In First Out).	It works on the concept of LIFO (Last In First Out).
7.	Suitable for	BFS is more suitable for searching vertices	DFS is more suitable when there are

S. No.	Parameters	BFS	DFS
		closer to the given source.	solutions away from source.
8.	**Suitability for Decision-Trees**	BFS considers all neighbors first and therefore not suitable for decision-making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, and the then explore all paths through this decision. And if this decision leads to win situation, we stop.
9.	**Time Complexity**	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.
10.	**Visiting of Siblings/ Children**	Here, siblings are visited before the children.	Here, children are visited before the siblings.
11.	**Removal of Traversed Nodes**	Nodes that are traversed several times are deleted from the queue.	The visited nodes are added to the stack and then removed when there are no more nodes to visit.
12.	**Backtracking**	In BFS there is no concept of backtracking.	DFS algorithm is a recursive algorithm that uses the idea of backtracking
13.	**Applications**	BFS is used in various applications such as bipartite graphs, shortest paths, etc.	DFS is used in various applications such as acyclic graphs and topological order etc.
14.	**Memory**	BFS requires more memory.	DFS requires less memory.
15.	**Optimality**	BFS is optimal for finding the shortest path.	DFS is not optimal for finding the shortest path.
16.	**Space complexity**	In BFS, the space complexity is more critical as compared to time complexity.	DFS has lesser space complexity because at a time it needs to store only a single path from the root to the leaf node.
17.	**Speed**	BFS is slow as compared to DFS.	DFS is fast as compared to BFS.
18,	**Tapping in loops**	In BFS, there is no problem of trapping into infinite loops.	In DFS, we may be trapped in infinite loops.
19.	**When to use?**	When the target is close to the source, BFS performs better.	When the target is far from the source, DFS is preferable.

5.Explain Min Heap. {VeryIMP}
[2]

Ans: A min heap can be implemented using the **priority_queue** container from the Standard Template Library (STL). The **priority_queue** container is a type of container adapter that provides a way to store elements in a queue-like data structure in which each element has a priority associated with it.

Syntax: priority_queue<int, vector<int>, greator<int>>minH;

6. Apply Rain Terrace algorithm to following problem Input : [4, 2, 0, 3, 2, 5] Draw the figure and find solution. {VeryIMP}
[4]

8.Write on algorithm for knight's Tour.
[3]

Ans: isValid(x, y, solution)

Input − Place x and y and the solution matrix.

Output − Check whether the (x,y) is in place and not assigned yet.

```
Begin
  if 0 ≤ x ≤ Board Size and 0 ≤ y ≤ Board Size, and (x, y) is empty, then
    return true
End
```

knightTour(x, y, move, sol, xMove, yMove)

Input − (x, y) place, number of moves, solution matrix, and possible x and y movement lists.

Output − The updated solution matrix if it exists.

```
Begin
  if move = Board Size * Board Size, then //when all squares are visited
    return true
  for k := 0 to number of possible xMovement or yMovement, do
    xNext := x + xMove[k]
    yNext := y + yMove[k]
    if isValid(xNext, yNext, sol) is true, then
      sol[xNext, yMext] := move
      if knightTour(xNext, yNext, move+1, sol, xMove, yMove), then
        return true
      else
        remove move from the sol[xNext, yNext] to backtrack
  done
  return false
```

End

9.Discuss use of Priority queve. [3]

Ans:

- Dijkstra's Shortest Path Algorithm using priority queue: When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing Dijkstra's algorithm.
- Prim's algorithm: It is used to implement Prim's Algorithm to store keys of nodes and extract minimum key node at every step.
- Data compression : It is used in Huffman codes which is used to compresses data.
- **Artificial Intelligence** : A* Search Algorithm : The A* search algorithm finds the shortest path between two vertices of a weighted graph, trying out the most promising routes first.
- The priority queue (also known as the fringe) is used to keep track of unexplored routes, the one for which a lower bound on the total path length is smallest is given highest priority.
- Heap Sort : Heap sort is typically implemented using Heap which is an implementation of Priority Queue.
- Operating systems: It is also used in Operating System for load balancing (load balancing on server), interrupt handling.
- **Optimization problems:** Priority Queue is used in optimization problems such as Huffman coding, Kruskal's Algorithm and Prim's Algorithm
- **Robotics:** Priority Queue is used in robotics to plan and execute tasks in a priority-based manner.
- **Event-driven simulations:** Priority queues are used in event-driven simulations, such as network simulations, to determine which events should be processed next.
- **Medical systems:** Priority queues are used in medical systems, such as triage systems in emergency departments, to prioritize patients based on the urgency of their condition.

10. Apply the maximum subarray algorithm to the Input : arr = [–2, –5, 6, –2, –3, 1, 5, –6] and find sum of maximum subarray

[4]

Ans:

The maximum subarray problem can be solved using the Kadane's algorithm, which efficiently finds the maximum sum of a subarray within a given array. Here's how you can apply it to the input array arr = [-2, -5, 6, -2, -3, 1, 5, -6]:

```
def max_subarray_sum(arr):
    max_ending_here = max_so_far = arr[0]  # Initialize variables to the first element of the array.

    for num in arr[1:]:
        # For each element, calculate the maximum ending here, considering the current element or starting a new subarray.
        max_ending_here = max(num, max_ending_here + num)

        # Update the maximum seen so far if necessary.
        max_so_far = max(max_so_far, max_ending_here)
```

```
    return max_so_far


# Input array
arr = [-2, -5, 6, -2, -3, 1, 5, -6]


# Find the maximum subarray sum
result = max_subarray_sum(arr)


# Print the maximum subarray sum
print("Maximum Subarray Sum:", result)
```

When you run this code with the provided input, it will output:

Maximum Subarray Sum: 7

11. Give the explicit and implicit constraints in 8 queen's problem. [3]

1. This pseudocode uses a backtracking algorithm to find a solution to the 8 Queen problem, which consists of placing 8 queens on a chessboard in such a way that no two queens threaten each other.
2. The algorithm starts by placing a queen on the first column, then it proceeds to the next column and places a queen in the first safe row of that column.
3. If the algorithm reaches the 8th column and all queens are placed in a safe position, it prints the board and returns true.
4. If the algorithm is unable to place a queen in a safe position in a certain column, it backtracks to the previous column and tries a different row.
5. The "isSafe" function checks if it is safe to place a queen on a certain row and column by checking if there are any queens in the same row, diagonal or anti-diagonal.
6. It's worth to notice that this is just a high-level pseudocode and it might need to be adapted depending on the specific implementation and language you are using.

Here is an example.

```
#include <iostream>

#include <vector>


using namespace std;


const int N = 8;
```

```
bool isSafe(vector<vector<int> >& board, int row, int col)
{
        for (int x = 0; x < col; x++)
                if (board[row][x] == 1)
                        return false;
        for (int x = row, y = col; x >= 0 && y >= 0; x--, y--)
                if (board[x][y] == 1)
                        return false;
        for (int x = row, y = col; x < N && y >= 0; x++, y--)
                if (board[x][y] == 1)
                        return false;
        return true;
}

bool solveNQueens(vector<vector<int> >& board, int col)
{
        if (col == N) {
                for (int i = 0; i < N; i++) {
                        for (int j = 0; j < N; j++)
                                cout << board[i][j] << " ";
                        cout << endl;
                }
                cout << endl;
                return true;
        }
        for (int i = 0; i < N; i++) {
                if (isSafe(board, i, col)) {
                        board[i][col] = 1;
                        if (solveNQueens(board, col + 1))
```

```
                                        return true;
                                board[i][col] = 0;
                        }
                }
                return false;
}

int main()
{
        vector<vector<int> > board(N, vector<int>(N, 0));
        if (!solveNQueens(board, 0))
                cout << "No solution found";
        return 0;
}
```

Output:

[[1, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 1], [0, 1, 0, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0, 0], [0, 0, 1, 0, 0, 0, 0, 0]]

Time Complexity : O((m + q) log^2 n)

Space Complexity : O((m + q) log n)

12. Discuss Hamiltonian Cycle. [3]

Ans:

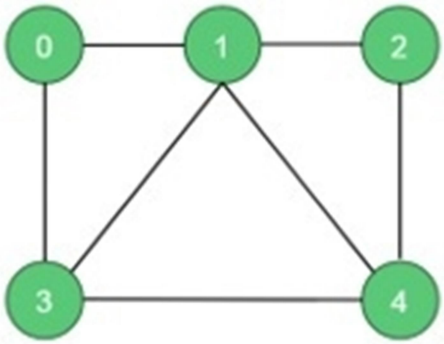
In an undirected graph, the Hamiltonian path is a path, that visits each vertex exactly once, and the Hamiltonian cycle or circuit is a Hamiltonian path, that there is an edge from the last vertex to the first vertex.

In this problem, we will try to determine whether a graph contains a Hamiltonian cycle or not. And when a Hamiltonian cycle is present, also print the cycle.

Input and Output

Input:

The adjacency matrix of a graph G(V, E).



Output:

The algorithm finds the Hamiltonian path of the given graph. For this case it is (0, 1, 2, 4, 3, 0). This graph has some other Hamiltonian paths.

If one graph has no Hamiltonian path, the algorithm should return false.

Algorithm

isValid(v, k)

Input − Vertex v and position k.

Output − Checks whether placing v in the position k is valid or not.

```
Begin
  if there is no edge between node(k-1) to v, then
    return false
  if v is already taken, then
    return false
  return true; //otherwise it is valid
End
```

13. Sort the following data using Mergesort [38, 27, 43, 3, 9, 82, 10]. {VeryIMP} **[4]**

Ans:

Here are the steps to sort the given data:

Split the list into two halves: Divide the list into two approximately equal halves.

Original List: [38, 27, 43, 3, 9, 82, 10]

Split into: [38, 27, 43] and [3, 9, 82, 10]

Recursively sort each half: Apply Mergesort to each of the two sublists.

First Half: [38, 27, 43]

Split into: [38], [27, 43]

Sort [38]: [38]

Sort [27, 43]:

Split into: [27] and [43]

Sort [27]: [27]

Sort [43]: [43]

Merge [27] and [43]: [27, 43]

Second Half: [3, 9, 82, 10]

Split into: [3, 9] and [82, 10]

Sort [3, 9]:

Split into: [3] and [9]

Sort [3]: [3]

Sort [9]: [9]

Merge [3] and [9]: [3, 9]

Sort [82, 10]:

Split into: [82] and [10]

Sort [82]: [82]

Sort [10]: [10]

Merge [82] and [10]: [10, 82]

Merge the sorted sublists: Combine the two sorted sublists from step 2 into one sorted list.

Merge [27, 43] and [3, 9, 82, 10]: [3, 9, 10, 27, 43, 82]

The final sorted list is [3, 9, 10, 27, 43, 82, 38].

So, the sorted list using Mergesort is [3, 9, 10, 27, 43, 82, 38].

14.Consider the following array [1, 3, 5, 8, 9, 2, 6, 7, 6] what is minimum number of jump required to reach the end of the array? [4]

Ans:

1. Initialize an array minJumps of the same length as the input array to store the minimum number of jumps required to reach each position. Initialize all elements of minJumps to a large value except for the first element, which is set to 0 since you start at the first position.
2. Iterate through the input array starting from the second element (i from 1 to n-1, where n is the length of the array).
3. For each element at index i, calculate the minimum number of jumps to reach that position by considering all possible jumps from previous positions (from j = 0 to i-1).
4. Check if it's possible to jump from position j to position i (i.e., arr[j] + j >= i).
5. If it's possible, calculate the minimum jumps required to reach position i from position j as minJumps[i] = min(minJumps[i], minJumps[j] + 1).
6. After iterating through the entire array, the value at minJumps[n-1] will represent the minimum number of jumps required to reach the end of the array.

15. Explain need of circular queve. [2]

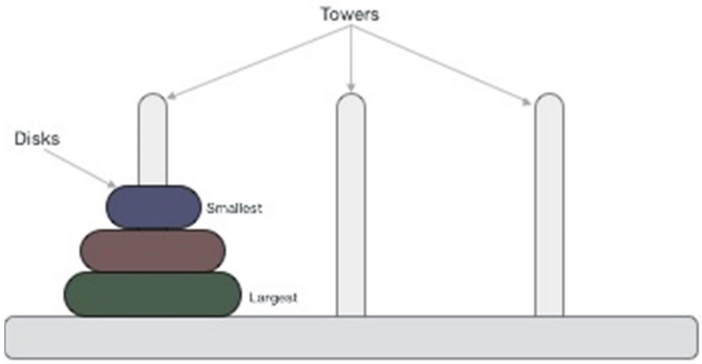
Ans:

In a circular queue, even if the position where the front was pointing becomes empty after removing an element, the space does not get wasted. This is because the rear can move circularly and start over from the first index once it reaches the end of the queue

16.Explain Rules for Tower of Hanoi with an suitable example. {VeryIMP} [4]

Ans:

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted −



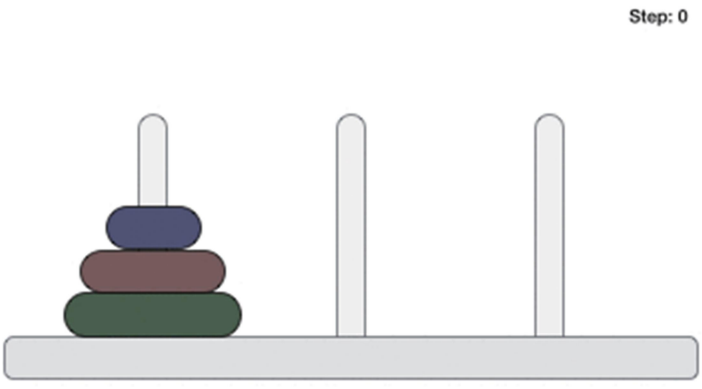
These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one. There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement. A few rules to be followed for Tower of Hanoi are −

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Following is an animated representation of solving a Tower of Hanoi puzzle with three disks.



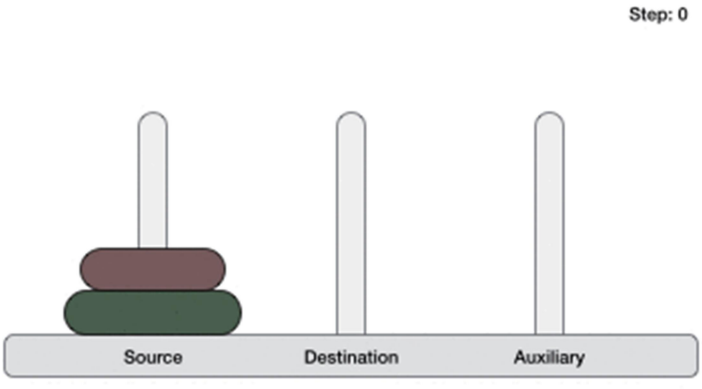
Tower of Hanoi puzzle with n disks can be solved in minimum $\mathbf{2^n-1}$ steps. This presentation shows that a puzzle with 3 disks has taken $\mathbf{2^3 - 1 = 7}$ steps.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, **source**, **destination** and **aux** (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.



So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk **n** from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

The steps to follow are −

Step 1 − Move n-1 disks from **source** to **aux**
Step 2 − Move n^{th} disk from **source** to **dest**
Step 3 − Move n-1 disks from **aux** to **dest**

A recursive algorithm for Tower of Hanoi can be driven as follows −

```
START
Procedure Hanoi(disk, source, dest, aux)

  IF disk == 1, THEN
    move disk from source to dest
  ELSE
    Hanoi(disk - 1, source, aux, dest)    // Step 1
    move disk from source to dest         // Step 2
    Hanoi(disk - 1, aux, dest, source)    // Step 3
  END IF

END Procedure
STOP
```

17. What is the purpose of Linked List.

[2]

Ans:

Task Scheduling- Operating systems use linked lists to manage task scheduling, where each process waiting to be executed is represented as a node in the list. Image Processing- Linked lists can be used to represent images, where each pixel is represented as a node in the list

18.Write an algorithm delete element from linked list whose sum is equal to zero. {VeryIMP}

Ans:

Suppose we have given the head of a linked list; we have to repeatedly delete consecutive sequences of nodes that sum to 0 until there are no such sequences. So after doing so, we have to return the head of the final linked list. So if the list is like [1,2,-3,3,1], then the result will be [3,1].

To solve this, we will follow these steps −

- Create a node called dummy, and store 0 into it, set next of dummy := head
- create one map m, store dummy for the key 0 into m, set sum = 0
- while head is not null −
  - sum := sum + value of head, set m[sum] := head, and head := next of head
- head := dummy
- sum := 0
- while head is not null
  - sum := sum + value of head
  - temp := m[sum]
  - if temp is not head, then next of head := next of temp
  - head := next of head
- return next of dummy

19. Find the longest common subsequence for following string using dynamic programming. [7]

X = {A, B, C, D, B, A, C, D, F}

Y = {C, B, A, F}{VeryIMP}

Ans:

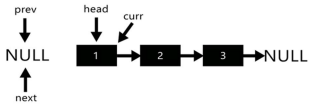
```
#include <stdio.h>

#include <string.h>


int max(int a, int b) {

   return (a > b) ? a : b;

}


int longestCommonSubsequence(char X[], int m, char Y[], int n) {

   int dp[m + 1][n + 1];
```

```
    // Initialize the first row and first column to zeros.
    for (int i = 0; i <= m; i++) {
        dp[i][0] = 0;
    }
    for (int j = 0; j <= n; j++) {
        dp[0][j] = 0;
    }

    // Fill in the dp table.
    for (int i = 1; i <= m; i+p+) {
        for (int j = 1; j <= n; j++) {
            if (X[i - 1] == Y[j - 1]) {
                dp[i][j] = 1 + dp[i - 1][j - 1];
            } else {
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    return dp[m][n];
}

int main() {
    char X[] = {'A', 'B', 'C', 'D', 'B', 'A', 'C', 'D', 'F'};
    int m = sizeof(X) / sizeof(X[0]);

    char Y[] = {'C', 'B', 'A', 'F'};
    int n = sizeof(Y) / sizeof(Y[0]);

    int lengthOfLCS = longestCommonSubsequence(X, m, Y, n);
```

```
    printf("Length of Longest Common Subsequence: %d\n", lengthOfLCS);


    return 0;
}
```

20. Write an algorithm to reverse the nodes of a linked list. **[3]**

Ans:




```
while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
```

- Initialize three pointers **prev** as NULL, **curr** as **head**, and **next** as NULL.
- Iterate through the linked list. In a loop, do the following:
    - Before changing the **next** of **curr**, store the **next** node
        - next = curr -> next
    - Now update the **next** pointer of **curr** to the **prev**
        - curr -> next = prev
    - Update **prev** as **curr** and **curr** as **next**
        - prev = curr
        - curr = next

21. Construct Binary Tree for following data 10, 25, 2, 4, 7, 13, 11, 22 and determine inorder, postorder & preorder?

[5]

1. **Ans:**
2. Start with an empty binary tree.
3. Insert each element from the given data into the binary tree one by one. When inserting elements, follow the rule that smaller elements go to the left subtree, and larger elements go to the right subtree.
4. Perform inorder, postorder, and preorder traversals to obtain the desired orderings.
5. Here's a step-by-step illustration of constructing the binary tree:

Initial Tree (Empty):

[ ]

Insert 10:

[10]

Insert 25 (larger than 10, goes to the right):

```
  [10]
    \
   [25]
```

Insert 2 (smaller than 10, goes to the left):

```
    [10]
   /   \
 [2]   [25]
```

Insert 4 (smaller than 10, goes to the left):

```
    [10]
   /   \
 [2]   [25]
   \
   [4]
```

Insert 7 (larger than 10, goes to the right):

```
    [10]
   /   \
 [2]   [25]
   \
   [4]
   \
   [7]
```

Insert 13 (larger than 10, goes to the right):

```
    [10]
   /   \
 [2]   [25]
   \     \
```

[4] [13]

\

[7]

Insert 11 (larger than 10, goes to the right):

[10]

/ \

[2] [25]

\ \

[4] [13]

\

[7]

\

[11]

Insert 22 (larger than 10, goes to the right):

[10]

/ \

[2] [25]

\ \

[4] [13]

\

[7]

\

[11]

\

[22]

Now that we have constructed the binary tree, we can find the inorder, postorder, and preorder traversals:

Inorder Traversal: Left -> Root -> Right

Inorder: 2, 4, 7, 10, 11, 13, 22, 25

Postorder Traversal: Left -> Right -> Root

Postorder: 4, 7, 2, 11, 22, 13, 25, 10

Preorder Traversal: Root -> Left -> Right

Preorder: 10, 2, 4, 7, 25, 13, 11, 22

These traversals give you the order in which the elements are visited when you traverse the binary tree in the specified manner.

22.Define Hash function 2 collision. [2]

Ans: Definition: A collision occurs when more than one value to be hashed by a particular hash function hash to the same slot in the table or data structure (hash table) being generated by the hash function.