1. Write the code for creating Binary Tree using pointers. Write a recursive function for in-order traversal of a binary tree.

Creating a binary tree using pointers and performing an in-order traversal can be achieved in many programming languages. Here, I'll provide an example in Python, which is a high-level programming language, to demonstrate this.

First, let's create a binary tree using pointers, and then we'll write a recursive function to perform an in-order traversal:

```
# Define a simple binary tree node
class TreeNode:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

# Function to perform in-order traversal recursively
def inorder_traversal(root):
    if root:
        # Traverse the left subtree
        inorder_traversal(root.left)
        # Visit the current node
        print(root.data, end=" ")
        # Traverse the right subtree
        inorder_traversal(root.right)

# Create a sample binary tree
#         1
#        / \
#       2   3
#      / \
#     4   5
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Perform in-order traversal
print("In-Order Traversal:")
inorder_traversal(root)
```

In this Python example, we define a simple `TreeNode` class to represent the nodes of the binary tree. We create a binary tree with a sample structure and then use the `inorder_traversal` function to perform an in-order traversal.

The output of the code will be:

```
In-Order Traversal:
4 2 5 1 3
```

This output represents the in-order traversal of the binary tree, which visits the nodes in ascending order when dealing with a binary search tree.

You can adapt this example to other programming languages by defining the appropriate data structures and functions for creating a binary tree and performing an in-order traversal recursively.

2. Write a nonrecursive in-order traversal of a binary tree.

A non-recursive (iterative) in-order traversal of a binary tree can be achieved using a stack data structure to mimic the function call stack that's used in the recursive version. Here's a Python example of a non-recursive in-order traversal:

```python
class TreeNode:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

def iterative_inorder_traversal(root):
    stack = []
    current = root

    while stack or current:
        # Traverse to the leftmost node
        while current:
            stack.append(current)
            current = current.left

        # Visit the node at the top of the stack
        current = stack.pop()
        print(current.data, end=" ")

        # Move to the right subtree
        current = current.right

# Create a sample binary tree
#         1
#        / \
#       2   3
#      / \
#     4   5
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
```

```
# Perform non-recursive in-order traversal
print("Non-Recursive In-Order Traversal:")
iterative_inorder_traversal(root)
```

In this code, we use a stack to mimic the function call stack, and we iteratively traverse the tree to perform an in-order traversal. The key idea is to start from the root and keep moving to the left until there are no more left children. Once a node is visited, we print its value and then move to the right subtree. This process is repeated until all nodes are traversed.

The output of the code will be:

```
Non-Recursive In-Order Traversal:
4 2 5 1 3
```

3. What are the properties of Binary Search Tree? Write a recursive function for implementation of binary search tree.

Properties of a Binary Search Tree (BST):

A Binary Search Tree is a binary tree with the following properties:

1. **Value Ordering:** For each node in the tree:
    - All values in its left subtree are less than the node's value.
    - All values in its right subtree are greater than the node's value.
2. **Unique Keys:** No two nodes in the tree can have the same value (keys are unique).
3. **Binary Tree:** Each node can have at most two children, typically referred to as the left child and the right child.

Recursive Function for Implementing a Binary Search Tree (BST) in Python:

Here's a Python implementation of a BST with a recursive function for insertion:

```
class TreeNode:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

def insert(root, key):
    if root is None:
        return TreeNode(key)

    if key < root.data:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)
    return root

def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.data, end=" ")
        inorder_traversal(root.right)

# Example of inserting values into a BST
root = None
root = insert(root, 50)
root = insert(root, 30)
root = insert(root, 20)
root = insert(root, 40)
root = insert(root, 70)
root = insert(root, 60)
root = insert(root, 80)

# Perform in-order traversal to check the BST structure
print("In-Order Traversal of the BST:")
inorder_traversal(root)
```

In this code, we define a `TreeNode` class to represent the nodes of the binary search tree. The `insert` function inserts a new key into the BST while maintaining the BST properties. The `inorder_traversal` function performs an in-order traversal to print the elements in ascending order, demonstrating the properties of a BST.

The output of the code will be:

```
In-Order Traversal of the BST:
20 30 40 50 60 70 80
```

4. Write the code for creating Binary Tree using pointers. Write a recursive function for Pre-order traversal of a binary tree with suitable example

Creating a binary tree using pointers and implementing a recursive function for pre-order traversal can be done in various programming languages. Below is an example in Python, demonstrating how to create a binary tree and perform a pre-order traversal:

```
class TreeNode:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

# Recursive function for pre-order traversal
```

```python
def preorder_traversal(root):
    if root:
        print(root.data, end=" ")  # Visit the current node
        preorder_traversal(root.left)  # Traverse the left subtree
        preorder_traversal(root.right)  # Traverse the right subtree

# Create a sample binary tree
#         1
#        / \
#       2   3
#      / \
#     4   5
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Perform pre-order traversal
print("Pre-Order Traversal:")
preorder_traversal(root)
```

In this Python code, we define a `TreeNode` class to represent the nodes of the binary tree. We create a binary tree with a sample structure and then use the `preorder_traversal` function to perform a pre-order traversal.

The output of the code will be:

```
Pre-Order Traversal:
1 2 4 5 3
```

5. Write an algorithm for deleting an item from binary search tree. Give example for different cases.

Deleting an item from a binary search tree (BST) involves handling several cases, depending on the node to be deleted. Here is an algorithm to delete a node from a binary search tree:

Algorithm to Delete a Node from a Binary Search Tree (BST):

1. Start at the root of the BST.
2. If the root is `None`, return `None` (the tree is empty).
3. If the value to be deleted is less than the value of the current node, recursively call the delete function on the left subtree.
4. If the value to be deleted is greater than the value of the current node, recursively call the delete function on the right subtree.
5. If the value to be deleted is equal to the value of the current node, handle the following cases: a. If the current node has no children, simply remove it. b. If the current node has one child, replace the node with its child. c. If the current node has two children, find the in-order successor (the smallest node in

the right subtree), replace the current node's value with the in-order successor's value, and recursively delete the in-order successor.

Example:

Let's illustrate these cases with an example. We have the following binary search tree:

```
      10
     /  \
    5    15
   / \   / \
  3   7 12  18
```

1. **Delete Node 7 (Node with No Children):**

   - Node 7 has no children. Simply remove it.
   - Resulting Tree:

     ```
           10
          /  \
         5    15
        /     / \
       3     12  18
     ```

2. **Delete Node 12 (Node with One Child):**

   - Node 12 has one child (15).
   - Replace node 12 with its child (15).
   - Resulting Tree:

     ```
           10
          /  \
         5    15
        /     / \
       3        18
     ```

3. **Delete Node 10 (Node with Two Children):**

   - Node 10 has two children (5 and 15).
   - Find the in-order successor (the smallest node in the right subtree), which is 15.
   - Replace the value of node 10 with 15.
   - Recursively delete the in-order successor (15).
   - Resulting Tree:

```
      15
     /  \
    5    18
   /
  3
```

6. Write function for creating the following

Certainly! Below are Python functions for creating both a Binary Tree and a Binary Search Tree (BST):

a) **Creating a Binary Tree:**

A Binary Tree is a tree structure where each node can have at most two children. There is no specific ordering of elements in a Binary Tree. Here's a function to create a Binary Tree:

```python
class TreeNode:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

def create_binary_tree():
    root = TreeNode(1)
    root.left = TreeNode(2)
    root.right = TreeNode(3)
    root.left.left = TreeNode(4)
    root.left.right = TreeNode(5)
    root.right.left = TreeNode(6)
    root.right.right = TreeNode(7)
    return root
```

b) **Creating a Binary Search Tree (BST):**

A Binary Search Tree is a binary tree with specific ordering properties, where the left subtree of a node contains values less than the node's value, and the right subtree contains values greater than the node's value. Here's a function to create a Binary Search Tree:

```python
class TreeNode:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

def insert(root, key):
    if root is None:
        return TreeNode(key)

    if key < root.data:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)

    return root

def create_binary_search_tree():
    root = None
    root = insert(root, 50)
    root = insert(root, 30)
    root = insert(root, 20)
    root = insert(root, 40)
    root = insert(root, 70)
    root = insert(root, 60)
    root = insert(root, 80)
    return root
```

7.Write a function for level order traversal of a binary tree. Illustrate with an example

A level-order traversal (also known as breadth-first traversal) of a binary tree visits the nodes level by level, starting from the root and moving to the next level before visiting the nodes on the level below. To perform level-order traversal, we use a queue data structure to keep track of the nodes at each level. Here's a Python function for level-order traversal of a binary tree:

```

class TreeNode:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

def level_order_traversal(root):
    if root is None:
        return

    queue = []
    queue.append(root)

    while queue:
        node = queue.pop(0)
        print(node.data, end=" ")

        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

# Create a sample binary tree
#          1
#         / \
#        2   3
#       / \
#      4   5
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Perform level-order traversal
print("Level-Order Traversal:")
level_order_traversal(root)
```

In this code, we define a `TreeNode` class to represent the nodes of the binary tree. The `level_order_traversal` function uses a queue to visit each node at each level of the tree, starting from the root and moving down level by level.

The output of the code will be:

```
Level-Order Traversal:
1 2 3 4 5
```

8. a) **Recursive Function for Creating Binary Search Tree (BST):**

In a Binary Search Tree (BST), you can insert elements recursively based on the properties of the tree. Here's a Python function to create a BST using recursion:

```
class TreeNode:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

def insert(root, key):
    if root is None:
        return TreeNode(key)

    if key < root.data:
        root.left = insert(root.left, key)
    else:
        root.right = insert(root.right, key)

    return root

# Example of creating a BST
def create_bst():
    root = None
    root = insert(root, 50)
    root = insert(root, 30)
    root = insert(root, 20)
    root = insert(root, 40)
    root = insert(root, 70)
    root = insert(root, 60)
    root = insert(root, 80)
    return root
```

The `insert` function inserts a new key into the BST while maintaining the BST properties. The `create_bst` function inserts several values to create a BST.

b) **Recursive Function for Searching an Element in a Binary Search Tree (BST):**

To search for an element in a BST, you can use a recursive function to traverse the tree based on the BST properties. Here's a Python function to search for an element in a BST:

```
def search(root, key):
    if root is None or root.data == key:
        return root

    if key < root.data:
        return search(root.left, key)

    return search(root.right, key)

# Example of searching for an element in the BST
def search_element_in_bst(root, key):
    result = search(root, key)
    if result:
        return f"Element {key} found in the BST."
    else:
        return f"Element {key} not found in the BST."
```

You can use the `search_element_in_bst` function to search for an element in the BST created earlier with the `create_bst` function.

Example usage:

```
bst_root = create_bst()
print(search_element_in_bst(bst_root, 30))  # Element 30 found in the BST.
print(search_element_in_bst(bst_root, 100))  # Element 100 not found in the BST.
```

9. Write functions for

a) **In-order Traversal (Left-Root-Right):**

```
# In-order Traversal (Left-Root-Right)
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.data, end=" ")
        inorder_traversal(root.right)

# Create a sample binary tree
#         1
#        / \
#       2   3
#      / \
#     4   5
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Perform In-order Traversal
print("In-order Traversal:")
inorder_traversal(root)
```

Output for In-order Traversal: `4 2 5 1 3`

b) **Pre-order Traversal (Root-Left-Right):**

```
# Pre-order Traversal (Root-Left-Right)
def preorder_traversal(root):
    if root:
        print(root.data, end=" ")
        preorder_traversal(root.left)
        preorder_traversal(root.right)

# Create a sample binary tree
#         1
#        / \
#       2   3
#      / \
#     4   5
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Perform Pre-order Traversal
print("Pre-order Traversal:")
preorder_traversal(root)
```

Output for Pre-order Traversal: `1 2 4 5 3`

c) **Post-order Traversal (Left-Right-Root):**

```
# Post-order Traversal (Left-Right-Root)
def postorder_traversal(root):
    if root:
        postorder_traversal(root.left)
        postorder_traversal(root.right)
        print(root.data, end=" ")

# Create a sample binary tree
#          1
#         / \
#        2   3
#       / \
#      4   5
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Perform Post-order Traversal
print("Post-order Traversal:")
postorder_traversal(root)
```

Output for Post-order Traversal: `4 5 2 3 1`

10. Write functions for

a) **Recursive Function for Finding the Height of a Binary Tree:** The height of a binary tree is the length of the longest path from the root to a leaf node. Here's a Python function to find the height of a binary tree using recursion:

```
class TreeNode:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

def find_tree_height(root):
    if root is None:
        return -1  # Height of an empty tree is -1

    left_height = find_tree_height(root.left)
    right_height = find_tree_height(root.right)

    return max(left_height, right_height) + 1

# Create a sample binary tree
#          1
#         / \
#        2   3
#       / \
#      4   5
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)

# Find the height of the binary tree
height = find_tree_height(root)
print("Height of the binary tree:", height)
```

Output:

```
Height of the binary tree: 2
```

b) **Recursive Functions for In-order and Post-order Traversal with Example:**

We'll use the same binary tree created in the previous examples to demonstrate in-order and post-order traversals.

```
# In-order Traversal (Left-Root-Right)
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.data, end=" ")
        inorder_traversal(root.right)

# Post-order Traversal (Left-Right-Root)
def postorder_traversal(root):
    if root:
        postorder_traversal(root.left)
        postorder_traversal(root.right)
        print(root.data, end=" ")

# Perform In-order Traversal
print("In-order Traversal:")
inorder_traversal(root)
print()

# Perform Post-order Traversal
print("Post-order Traversal:")
postorder_traversal(root)
```

Output for In-order Traversal: `4 2 5 1 3`

Output for Post-order Traversal: `4 5 2 3 1`

11. Giving an example write the function for building adjacency list of a Graph.

To build an adjacency list representation of a graph, you can use a dictionary where each key represents a vertex, and the corresponding value is a list of adjacent vertices. Here's a Python function to build the adjacency list of a graph using an example:

```python
def build_adjacency_list(graph_edges):
    adjacency_list = {}  # Initialize an empty dictionary for the adjacency list

    for edge in graph_edges:
        u, v = edge  # Assuming (u, v) represents an edge from vertex u to vertex
v

        # Add v to the adjacency list of u
        if u in adjacency_list:
            adjacency_list[u].append(v)
        else:
            adjacency_list[u] = [v]

        # Add u to the adjacency list of v (for undirected graph)
        if v in adjacency_list:
            adjacency_list[v].append(u)
        else:
            adjacency_list[v] = [u]

    return adjacency_list

# Example: Building the adjacency list for an undirected graph
graph_edges = [(1, 2), (1, 3), (2, 3), (3, 4), (4, 5)]
adjacency_list = build_adjacency_list(graph_edges)

# Print the adjacency list
for vertex, neighbors in adjacency_list.items():
    print(f"Vertex {vertex} is adjacent to vertices: {', '.join(map(str,
neighbors))}")
```

Output for the example:

```
Vertex 1 is adjacent to vertices: 2, 3
Vertex 2 is adjacent to vertices: 1, 3
Vertex 3 is adjacent to vertices: 1, 2, 4
Vertex 4 is adjacent to vertices: 3, 5
Vertex 5 is adjacent to vertices: 4
```

12. Discuss the Implement DFS with an example.

Depth-First Search (DFS) is a graph traversal algorithm used to explore and navigate through a graph or tree structure. It starts at the root (or an arbitrary node for graphs) and explores as far as possible along each branch before backtracking. DFS can be implemented using recursion or a stack data structure. Let's discuss the recursive implementation of DFS with an example.

Consider the following graph as an example:

```
1 -- 2 -- 4
|    |
3 -- 5
```

We'll implement DFS to traverse this graph starting from vertex 1.

Python Recursive DFS Implementation:

```python
# Define a class for representing a graph using an adjacency list
class Graph:
    def __init__(self):
        self.graph = {}

    # Function to add an edge to the graph
    def add_edge(self, u, v):
        if u in self.graph:
            self.graph[u].append(v)
        else:
            self.graph[u] = [v]

    # Recursive DFS function
    def dfs(self, vertex, visited):
        visited.add(vertex)  # Mark the current vertex as visited
        print(vertex, end=" ")  # Print the current vertex

        # Recur for all the adjacent vertices that haven't been visited yet
        for neighbor in self.graph.get(vertex, []):
            if neighbor not in visited:
                self.dfs(neighbor, visited)

# Create the graph
g = Graph()
g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(2, 4)
g.add_edge(2, 5)

# Perform DFS from vertex 1
print("Depth-First Search (DFS) starting from vertex 1:")
visited_set = set()  # Initialize a set to keep track of visited vertices
g.dfs(1, visited_set)
```

Output of the DFS starting from vertex 1:

```
Depth-First Search (DFS) starting from vertex 1:
1 2 4 5 3
```

13. Write a function for Breadth First Search traversal of a Graph.

Breadth-First Search (BFS) is a graph traversal algorithm used to explore and navigate through a graph or tree structure. It explores all the vertices at the current level before moving to the next level. BFS can be implemented using a queue data structure. Here's a Python function for BFS traversal of a graph:

```
from collections import defaultdict, deque

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)

    def bfs(self, start_vertex):
        visited = set()
        queue = deque()

        visited.add(start_vertex)
        queue.append(start_vertex)

        while queue:
            vertex = queue.popleft()
            print(vertex, end=" ")

            for neighbor in self.graph[vertex]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)

# Create the graph
g = Graph()
g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(2, 4)
g.add_edge(2, 5)

# Perform BFS starting from vertex 1
print("Breadth-First Search (BFS) starting from vertex 1:")
g.bfs(1)
```

Output of the BFS starting from vertex 1:

```
Breadth-First Search (BFS) starting from vertex 1:
1 2 3 4 5
```

14.Define minimum spanning tree. Describe Prim's algorithms for finding the minimum spanning tree and illustrate with an example.

A Minimum Spanning Tree (MST) of a connected, undirected graph is a subgraph that includes all the vertices of the original graph and a subset of the edges such that the subgraph is a tree and the sum of the weights of its edges is minimized. In simpler terms, it's a tree that connects all the vertices of the graph with the minimum total edge weight. MSTs are commonly used in network design, clustering, and optimization problems.

Prim's Algorithm for Finding Minimum Spanning Tree: Prim's algorithm is a greedy algorithm used to find the Minimum Spanning Tree of a graph. It starts with an arbitrary vertex and repeatedly adds the vertex with the minimum-weight edge connecting it to the existing tree. The process continues until all vertices are included in the tree. Here's how Prim's algorithm works:

1. Initialize an empty set to represent the Minimum Spanning Tree (MST).
2. Select an arbitrary starting vertex and add it to the MST.
3. While the MST has fewer vertices than the original graph: a. Find the minimum-weight edge that connects a vertex in the MST to a vertex outside the MST. b. Add the vertex at the other end of this edge to the MST.
4. Repeat step 3 until the MST includes all vertices.

Example of Prim's Algorithm:

Consider the following weighted graph:

```
   2   3
A --- B --- C
|   /   \   |
1  1     4  5
| /       \ |
D --------- E
```

Let's find the Minimum Spanning Tree using Prim's algorithm, starting from vertex A:

1. Start with A in the MST: `A`
2. Iteration 1:
   - Add the minimum-weight edge (1) connecting A to D: `A - D`
3. Iteration 2:
   - Add the minimum-weight edge (2) connecting A to B: `A - D - B`

4. Iteration 3:
   - Add the minimum-weight edge (3) connecting B to E: `A - D - B - E`
5. Iteration 4:
   - Add the minimum-weight edge (4) connecting B to C: `A - D - B - E - C`

15. Write code for the SINGLE SOURCE SHORTEST PATHS problem. Illustrate this with an example.

The Single Source Shortest Paths (SSSP) problem involves finding the shortest paths from a single source vertex to all other vertices in a weighted graph. One of the most commonly used algorithms to solve this problem is Dijkstra's algorithm. I'll provide a Python code example to illustrate how to find the shortest paths using Dijkstra's algorithm:

```
import heapq

def dijkstra(graph, source):
    # Initialize distances and predecessors
    distances = {vertex: float('infinity') for vertex in graph}
    distances[source] = 0
    predecessors = {}

    # Create a priority queue with (distance, vertex) pairs
    priority_queue = [(0, source)]

    while priority_queue:
        # Get the vertex with the smallest distance
        current_distance, current_vertex = heapq.heappop(priority_queue)

        # If we already processed this vertex, skip it
        if current_distance > distances[current_vertex]:
            continue

        # Visit all the neighbors of the current vertex
        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                predecessors[neighbor] = current_vertex
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances, predecessors

# Example graph represented as an adjacency dictionary
graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'A': 1, 'C': 2, 'D': 5},
    'C': {'A': 4, 'B': 2, 'D': 1},
    'D': {'B': 5, 'C': 1}
}
```

```
# Find shortest paths from source vertex 'A'
source_vertex = 'A'
shortest_distances, predecessors = dijkstra(graph, source_vertex)

# Print the shortest distances and paths from 'A' to all other vertices
for vertex, distance in shortest_distances.items():
    path = [vertex]
    current = vertex
    while current != source_vertex:
        current = predecessors[current]
        path.insert(0, current)
    print(f"Shortest path from {source_vertex} to {vertex}: {path}, Distance:
{distance}")
```

Output for the example:

```
Shortest path from A to A: ['A'], Distance: 0
Shortest path from A to B: ['A', 'B'], Distance: 1
Shortest path from A to C: ['A', 'C'], Distance: 3
Shortest path from A to D: ['A', 'C', 'D'], Distance: 4
```

16. Define minimum spanning tree. Describe Kruskal's algorithms for finding the minimum spanning tree and illustrate with an example.

A Minimum Spanning Tree (MST) of a connected, undirected graph is a subgraph that includes all the vertices of the original graph and a subset of the edges such that the subgraph is a tree and the sum of the weights of its edges is minimized. In other words, it's a tree that connects all the vertices of the graph with the minimum total edge weight. MSTs are commonly used in network design, clustering, and optimization problems.

Kruskal's Algorithm for Finding Minimum Spanning Tree: Kruskal's algorithm is a greedy algorithm used to find the Minimum Spanning Tree of a graph. It starts with an empty set of edges and repeatedly adds the smallest-weight edge that connects two disconnected components of the graph. The process continues until there's a single connected component. Here's how Kruskal's algorithm works:

1. Initialize an empty set to represent the Minimum Spanning Tree (MST).

2. Sort all the edges of the graph in non-decreasing order of their weights.

3. For each edge in the sorted list: a. If adding the edge to the MST does not create a cycle, add it to the MST. b. Otherwise, discard the edge.

4. Repeat step 3 until the MST includes all vertices.

Example of Kruskal's Algorithm:

Consider the following weighted graph:

```
      4
A --------- B
| \       | \
|  1      |  5
|   \     |   \
C --- 3   D --- 2
```

Let's find the Minimum Spanning Tree using Kruskal's algorithm:

1. Start with an empty MST: `[]`
2. Sort all the edges by weight in non-decreasing order: `[C-D (2), A-C (3), A-B (4), B-D (5), A-D (6), B-C (7)]`
3. Iteration 1:
   - Add the smallest-weight edge `C-D (2)` to the MST: `[C-D (2)]`
4. Iteration 2:
   - Add the next smallest-weight edge `A-C (3)` to the MST: `[C-D (2), A-C (3)]`
5. Iteration 3:
   - Add the next smallest-weight edge `A-B (4)` to the MST: `[C-D (2), A-C (3), A-B (4)]`
6. Iteration 4:
   - Add the next smallest-weight edge `B-D (5)` to the MST: `[C-D (2), A-C (3), A-B (4), B-D (5)]`
7. The MST now includes all vertices, and we have a Minimum Spanning Tree.

The resulting Minimum Spanning Tree is:

```
   4
A ----- B
 \       \
  3       5
   \     /
    C---D
```

This tree connects all the vertices of the original graph while minimizing the sum of edge weights. Kruskal's algorithm is efficient and works for both connected and disconnected graphs.

17. Write short notes on the following (with respect to graph).

a) Spanning Tree:

```
In graph theory, a spanning tree of a connected, undirected graph is a subgraph
that includes all the vertices of the original graph while being a tree, which
means it is acyclic (no cycles) and connected (there is a path between any two
vertices). Spanning trees are essential in various applications, including network
design, optimization, and ensuring connectivity in a graph.
```

b) Connected Graph:

```
A graph is said to be connected if there is a path between every pair of vertices
in the graph. In a connected graph, no vertices are isolated; they are all
reachable from one another. A connected graph can be thought of as a single
connected component. If a graph is not connected, it can be divided into connected
components, where each component is a connected subgraph.
```

c) Simple Path:

```
A simple path in a graph is a path (a sequence of edges and vertices) in which no
vertex or edge is repeated. In other words, it's a unique sequence of vertices and
edges from one vertex to another without revisiting any vertex or edge. Simple
paths are essential for analyzing various properties of graphs, including finding
shortest paths and determining connectivity.
```

d) Complete Graph:

```
A complete graph is a simple graph in which every pair of distinct vertices is
connected by a unique edge. In other words, every vertex in a complete graph is
adjacent to all other vertices. A complete graph is denoted by the symbol K_n,
where n is the number of vertices in the graph. Complete graphs have specific
properties, including being regular, having a high degree of connectivity, and
having a high number of edges. They are often used in mathematical and
combinatorial problems.
```

18. Write short notes on following with respect to graph

b) Walk:

```
A walk in a graph is a sequence of vertices and edges that can potentially revisit
the same vertices and edges multiple times. It can be open-ended or closed.
```

c) Path:

```
A path in a graph is a special type of walk that doesn't revisit the same vertex
or edge. It connects a starting vertex to an ending vertex.
```

d) Incidence:

```
Incidence in graph theory refers to the relationship between vertices and edges.
It indicates which edges are connected to which vertices in a graph.
```

e) Degree:

```
The degree of a vertex is the number of edges incident to that vertex. It measures
a vertex's connectivity in the graph.
```

f) Cycle:

```
A cycle in a graph is a closed path where the starting and ending vertices are the
same. It can be simple (no repeated vertices or edges) or non-simple (repeated
vertices or edges). Cycles are essential for understanding graph structure and
connectivity.
```

19. Write short notes on the following (with respect to graph)

a) Complete Graph:

```
In a complete graph, every pair of vertices is directly connected by an edge.
```

b) Connected and Disconnected Graph:

```
A connected graph has a path between any two vertices, while a disconnected graph
has isolated components.
```

c) Directed and Undirected Graph:

```
Directed graphs have one-way edges, while undirected graphs have bidirectional
edges.
```

d) Component:

```
A component in a graph is a self-contained, connected subgraph of vertices within
the larger graph.
```

20. Describe Prim's and Kruskal's algorithms for finding the minimum spanning tree.

Prim's and Kruskal's algorithms are used to find the minimum spanning tree (MST) in a connected, undirected graph. Both algorithms aim to connect all vertices with the minimum possible total edge weight, forming a tree that spans the entire graph.

Prim's Algorithm:

1. Start with an arbitrary vertex as the initial MST.
2. In each step, add the minimum-weight edge that connects a vertex in the MST to a vertex outside the MST.
3. Continue this process until all vertices are included in the MST.
4. The result is the minimum spanning tree.

Kruskal's Algorithm:

1. Start with an empty set of edges as the initial MST.
2. Sort all the edges in the graph in ascending order of weight.
3. Consider the edges one by one in their sorted order and add each edge to the MST if it doesn't form a cycle.
4. To check for cycles, you can use a data structure like a disjoint-set (union-find) data structure.
5. Continue this process until the MST contains (V - 1) edges, where V is the number of vertices.
6. The result is the minimum spanning tree.

Both algorithms are greedy and guarantee an MST, but they may differ in their performance and specific applications. Prim's algorithm is often more efficient for dense graphs, while Kruskal's algorithm can be more efficient for sparse graphs.

21. Write short notes on the following (with respect to graph)

a) Complete Graph

```
A **complete graph** is a type of graph in which every pair of vertices is
connected by an edge. In other words, it is a graph where there is an edge between
every pair of distinct vertices. The complete graph with 'n' vertices is denoted
as K<sub>n</sub>. Complete graphs are useful in various mathematical and
theoretical contexts.
```

b) Connected and Disconnected Graph

```
A **connected graph** is a graph in which there is a path between every pair of
vertices. In other words, you can reach any vertex from any other vertex by
following the edges. On the other hand, a graph that is not connected is called a
**disconnected graph**. In a disconnected graph, there are one or more pairs of
vertices for which there is no path connecting them.
```

c) Directed & Undirected Graph

Undirected Graphs

```
An **undirected graph** is a type of graph in which edges have no specific
direction. They are bidirectional, and there is no concept of parent-child
relationships between vertices. Undirected graphs exhibit symmetry, and there is
always a way to reach any vertex from another. Two common algorithms for
undirected graphs are:
```

- **Breadth-First Search (BFS)**
- **Depth-First Search (DFS)**

Directed Graphs

```
A **directed graph** is a graph in which edges have a specific direction. They can
be unidirectional or bidirectional, and cycles can exist in directed graphs.
Directed graphs lack symmetry, and connectivity can vary. There may be multiple
ways to traverse from one vertex to another.
```

d) Component

In graph theory, a **component** of an undirected graph is a connected subgraph that is not part of any larger connected subgraph. Components partition the graph's vertices into disjoint sets and are the induced subgraphs of those sets. A graph itself is considered a component if it is connected. The number of components is an important graph invariant.

Components are useful in various applications, including image analysis, where connected-component labeling is a fundamental technique. Dynamic connectivity algorithms can maintain components efficiently as edges are inserted or deleted. Additionally, components are studied in computational complexity theory, and they are relevant in the analysis of random graphs.

21. Write short notes on the following (with respect to graph)

a) Complete Graph: A **complete graph** is an undirected graph in which every pair of distinct vertices is connected by a unique edge. In other words, every vertex in a complete graph is adjacent to all other vertices. A complete graph is denoted by the symbol K_n, where n is the number of vertices in the graph.

Characteristics of Complete Graph:

- **Connectedness:** A complete graph is a connected graph, meaning there exists a path between any two vertices in the graph.
- **Count of Edges:** Every vertex in a complete graph has a degree of (n-1), where n is the number of vertices. Total edges in a complete graph are n*(n-1)/2.
- **Symmetry:** Every edge in a complete graph is symmetric, un-directed, and connects two vertices in the same way.
- **Transitivity:** A complete graph is transitive, meaning if vertex A is connected to vertex B and vertex B is connected to vertex C, then vertex A is also connected to vertex C.
- **Regularity:** A complete graph is regular, meaning every vertex has the same degree.

Applications of Complete Graph:

- **Transportation networks:** Representing motorways, aircraft routes, where every point is connected to every other location directly.
- **Network analysis:** Measuring characteristics like clustering, connectedness, or shortest path length.
- **Optimization:** Used in optimization problems like finding the maximum clique in a graph.

b) Connected Graph:

```
In graph theory, a graph is said to be connected if there exists a path between
every pair of vertices. If there are two vertices for which no path connects them,
the graph is considered disconnected. Even a single vertex can be considered a
connected graph, while a graph with two or more vertices but no edges is
disconnected. In directed graphs, additional criteria like weak connectivity and
strong connectivity can be applied.
```

c) Degree of Graph:

```
The degree of a graph refers to the number of edges connected to a vertex in the
graph. It is denoted as deg(v), where v represents a vertex. The degree of a graph
is essentially the number of relations a particular node makes with the other
nodes in the graph. In a multigraph, a loop contributes 2 to a vertex's degree.
The degree of the graph is the largest vertex degree. In other words, it's the
number of edges incident to a vertex (node).
```

d) Cycle:

```
A cycle in graph theory is a closed chain of vertices and edges. A cycle graph
consists of a single cycle and at least three vertices. The cycle graph with n
vertices is called C_n. It has the same number of vertices as edges, and every
vertex has a degree of 2, meaning each vertex has exactly two edges connected to
it. Cycle graphs have various properties, including being 2-edge colorable, 2-
regular, 2-vertex colorable (for even vertices), and being connected, Eulerian,
and Hamiltonian.
```

22 Write a recursive function for binary search. What is its time complexity?

Binary search is an efficient algorithm for searching for a target element within a sorted array. This algorithm takes advantage of the sorted nature of the array by repeatedly dividing the search space in half, making it significantly faster than linear search.

Algorithm

1. Compare the target element 'x' with the middle element of the current search space.
2. If 'x' matches the middle element, return the index of the middle element, indicating a successful search.
3. If 'x' is greater than the middle element, the target 'x' can only exist in the right (greater) half of the subarray. Recursively apply the binary search algorithm to the right half.
4. If 'x' is smaller than the middle element, the target 'x' must lie in the left (lower) half of the subarray. Recursively apply the binary search algorithm to the left half.

Here is a Python implementation of the binary search algorithm:

```python
def binarySearch(arr, left, right, number):
    # Base case: The search space is exhausted.
    if left > right:
        return -1

    # Calculate the mid-point in the search space and compare it to the target
number.
    mid = (left + right) // 2

    # Base case: The target number is found.
    if number == arr[mid]:
        return mid

    # If the target number is smaller, search the left half.
    elif number < arr[mid]:
        return binarySearch(arr, left, mid - 1, number)

    # If the target number is larger, search the right half.
    else:
        return binarySearch(arr, mid + 1, right, number)

# Input: Sorted array 'arr', left and right indices, and the target number
'number'
# Output: The index of the target number or -1 if not found

# Example usage:
arr = [1, 3, 5, 7, 9, 11, 13, 15]
number = 7
(left, right) = (0, len(arr) - 1)
index = binarySearch(arr, left, right, number)
```

```
if index != -1:
    print('Element found at index', index)
else:
    print('Element not found in the array')
```

Time Complexity

The time complexity of the binary search algorithm is O(log n), where 'n' is the number of elements in the sorted array. This efficiency is achieved because binary search reduces the search space by half with each comparison, making it significantly faster than linear search (O(n)) for large datasets. The algorithm efficiently narrows down the search space in a divide-and-conquer manner, resulting in logarithmic time complexity.

23. Write a dynamic programming algorithm for the ALL PAIRS SHORTEST PATHS problem. Illustrate this with an example.

The Floyd-Warshall algorithm is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. It works by iteratively updating a distance matrix, where each entry represents the shortest distance between two vertices. The algorithm is guaranteed to find the shortest paths for any graph, whether it is directed or undirected.

Algorithm Pseudocode

```
for k in range(0, n):
  for i in range(0, n):
    for j in range(0, n):
      if (cost[i][k] + cost[k][j] < cost[i][j]):
        cost[i][j] = cost[i][k] + cost[k][j]
```

- `n` is the number of vertices in the graph.
- `cost[i][j]` represents the shortest path from vertex `i` to vertex `j`.

Example

Let's illustrate the Floyd-Warshall algorithm with an example:

```
# Create a graph with 5 vertices and 6 edges
graph = [[0, 10, 20, 30, 40],
         [10, 0, 50, 60, 70],
         [20, 50, 0, 80, 90],
         [30, 60, 80, 0, 100],
         [40, 70, 90, 100, 0]]

# Initialize the distance matrix with infinities
distance_matrix = [[float("inf") for i in range(5)] for j in range(5)]

# Copy the original graph values to the distance matrix
```

```
for i in range(5):
    for j in range(5):
        if i == j:
            distance_matrix[i][j] = 0
        else:
            distance_matrix[i][j] = graph[i][j]

# Apply the Floyd-Warshall algorithm
for k in range(5):
    for i in range(5):
        for j in range(5):
            if (distance_matrix[i][k] + distance_matrix[k][j] < distance_matrix[i]
[j]):
                distance_matrix[i][j] = distance_matrix[i][k] + distance_matrix[k]
[j]

# Print the distance matrix
for row in distance_matrix:
    print(row)
```

The output of the above code is the following distance matrix:

```
[[0, 10, 20, 30, 40],
 [10, 0, 50, 60, 70],
 [20, 50, 0, 80, 90],
 [30, 60, 80, 0, 100],
 [40, 70, 90, 100, 0]]
```

This distance matrix shows the shortest paths between all pairs of vertices in the graph. For example, the shortest path from vertex 0 to vertex 4 is 10 + 20 + 30 = 60. The algorithm has successfully computed the shortest paths between all pairs of vertices in the graph.

24. What is hashing? Briefly explain various Hashing techniques.

Hashing is a technique used in computer science to map data of arbitrary size (such as a file, a password, or any data) to a fixed-size value, which is typically a numerical value or a small string of characters. This mapped value is known as a hash or a hash code. Hashing is commonly used in data structures like hash tables to quickly locate a data record given its search key.

Hash Function: A hash function is a mathematical function that takes an input (or 'key') and returns a fixed-size string of characters or a numerical value. The output (hash code) is typically a digest of the input data, making it difficult to reverse the process and derive the original input from the hash code. Hash functions are designed to be fast and efficient, and they should produce a unique hash code for different inputs, but there can be hash collisions where different inputs produce the same hash code.

There are various hashing techniques used, including:

1. **Division Method:**

  - Formula: `h(K) = k mod M`
  - Divides the key `k` by the size of the hash table `M` and uses the remainder as the hash value.
  - M is often chosen as a prime number to distribute keys more evenly.

2. **Mid Square Method:**

   - Formula: `h(K) = h(k x k)`
   - Involves squaring the key `k` and extracting the middle digits as the hash value.
   - The number of digits to extract can depend on the table size.

3. **Digit Folding Method:**

   - Formula: `k = k1, k2, k3, k4, ….., kn`, `h(K) = s`
   - Divides the key `k` into parts and adds them to obtain `s`.
   - The final hash value is derived from `s`.

4. **Multiplication Method:**

   - Formula: `h(K) = floor(M (kA mod 1))`
   - Multiplies the key `k` by a constant value `A`, extracts the fractional part of `kA`, and then multiplies it by the table size `M`.
   - The result is truncated to an integer to get the hash value.

These hashing techniques are used to efficiently map keys to indexes in a hash table, allowing for fast data retrieval in applications like dictionaries, databases, and caches. Proper choice of a hash function and handling hash collisions are important for the efficient operation of data structures that rely on hashing.

25. Explain Closed hashing technique (using arrays) with required functions

Closed hashing, also known as open addressing or closed addressing, is a collision resolution technique used in hash tables. In closed hashing, when a collision occurs (i.e., two or more keys hash to the same index), the new key is placed in another location within the same array, typically through a predefined probing sequence. This means that all keys are stored directly in the array, and there is no need for additional data structures like linked lists or separate chaining. Here, I'll provide an explanation of closed hashing using arrays along with key functions:

Key Concepts:

1. **Hash Function:** The first step in closed hashing is to define a hash function that takes a key and returns an index in the array where the key should be stored. The hash function should be designed to minimize collisions as much as possible.

2. **Initialization:** Create an array with a fixed size to store the keys and initialize all elements to a special value (e.g., null or an empty marker) to indicate empty slots.

3. **Insertion:** When inserting a new key into the hash table, you calculate the hash value using the hash function. If the calculated index is already occupied (a collision has occurred), you use a probing sequence to find the next available slot in the array. Common probing techniques include linear probing, quadratic probing, and double hashing.

Required Functions:

Here are the key functions involved in closed hashing using arrays:

1. Hash Function:

```
def hash_function(key, table_size):
    # Calculate the hash value for the given key
    return key % table_size
```

2. Initialization:

```
def initialize_table(table_size):
    # Create an array with all elements set to a special marker (e.g., None)
    table = [None] * table_size
    return table
```

3. Insertion (Linear Probing):

```
def insert(key, value, table):
    table_size = len(table)
    index = hash_function(key, table_size)

    while table[index] is not None:
        # Linear probing: Move to the next slot if the current slot is occupied
        index = (index + 1) % table_size

    # Insert the key-value pair into the first available slot
    table[index] = (key, value)
```

4. Search (Linear Probing):

```
def search(key, table):
    table_size = len(table)
    index = hash_function(key, table_size)

    while table[index] is not None:
        if table[index][0] == key:
            return table[index][1]  # Return the value associated with the key
        # Linear probing: Move to the next slot if the current slot is not the
target
        index = (index + 1) % table_size

    return None  # Key not found
```

5. Deletion (Linear Probing):

```
def delete(key, table):
    table_size = len(table)
    index = hash_function(key, table_size)

    while table[index] is not None:
        if table[index][0] == key:
            table[index] = None  # Mark the slot as empty
            return
        # Linear probing: Move to the next slot if the current slot is not the
target
        index = (index + 1) % table_size
```

6. Load Factor: It's important to monitor the load factor (the ratio of the number of stored keys to the table size) and, if it exceeds a certain threshold, resize the array and rehash the keys to maintain efficiency.

Closed hashing is a straightforward technique for handling collisions in hash tables, but it may require careful consideration of probing sequences and load factor management to ensure good performance. There are variations and optimizations of closed hashing, but the basic concept remains the same.

26. Explain separate chain hashing (using linked list) technique with required functions.

Separate chaining, also known as open hashing, is a collision resolution technique used in hash tables. In separate chaining, when a collision occurs (i.e., two or more keys hash to the same index), each index of the hash table contains a linked list or another data structure to store multiple values that map to the same index. This allows for the efficient handling of collisions by maintaining a collection of values associated with each index. Here, I'll provide an explanation of separate chain hashing using linked lists along with the key functions:

Key Concepts:

1. **Hash Function:** The first step in separate chaining is to define a hash function that takes a key and returns an index in the array where the key should be stored. The hash function should be designed to minimize collisions as much as possible.

2. **Initialization:** Create an array with a fixed size to store the linked lists or other data structures, and initialize all elements to empty linked lists.

3. **Insertion:** When inserting a new key-value pair into the hash table, you calculate the hash value using the hash function and then add the value to the linked list at the calculated index.

4. **Search:** To search for a key, you calculate the hash value and traverse the linked list at that index to locate the key or determine if it's not present.

5. **Deletion:** Deleting a key involves searching for it in the linked list at the calculated index and removing it if found.

6. **Load Factor:** Monitor the load factor (the ratio of the number of stored keys to the table size) and, if it exceeds a certain threshold, resize the array and rehash the keys to maintain efficiency.

Required Functions:

Here are the key functions involved in separate chain hashing using linked lists:

1. Hash Function:

```
def hash_function(key, table_size):
    # Calculate the hash value for the given key
    return key % table_size
```

2. Initialization:

```
def initialize_table(table_size):
    # Create an array of empty linked lists
    table = [LinkedList() for _ in range(table_size)]
    return table
```

3. Insertion:

```
def insert(key, value, table):
    table_size = len(table)
    index = hash_function(key, table_size)

    # Add the key-value pair to the linked list at the calculated index
    table[index].add(key, value)
```

4. Search:

```
def search(key, table):
    table_size = len(table)
    index = hash_function(key, table_size)

    # Search for the key in the linked list at the calculated index
    return table[index].find(key)
```

5. Deletion:

```
def delete(key, table):
    table_size = len(table)
    index = hash_function(key, table_size)

    # Delete the key from the linked list at the calculated index
    table[index].remove(key)
```

6. Load Factor: Monitoring the load factor and resizing the array involves comparing the number of stored keys to the table size and, if needed, creating a new, larger array, rehashing the keys, and transferring them to the new array.

Separate chaining with linked lists is a flexible and efficient technique for handling collisions in hash tables. It provides a simple and effective way to store multiple values that map to the same index, ensuring that the hash table remains efficient even when collisions occur.

27. What do you mean by collision in hashing? Write short notes on various collision resolution techniques.

Collision in Hashing:

In hashing, a collision occurs when two or more distinct keys hash to the same index or slot in a hash table. Collisions are common because the number of possible keys is usually much larger than the number of available slots in the hash table. Handling collisions is a fundamental problem in hash table design because you need a strategy to manage multiple keys hashing to the same location.

Collision Resolution Techniques:

There are several techniques to resolve collisions in hash tables. Here are some common collision resolution methods:

1. **Separate Chaining:**

   - In separate chaining, each index in the hash table holds a linked list or another data structure to store multiple values that map to the same index.
   - When a collision occurs, the new key-value pair is simply added to the linked list at the corresponding index.
   - This method is simple and allows multiple values to be stored at the same index.

2. **Open Addressing (Closed Hashing):**

   - In open addressing, when a collision occurs, the new key is placed in another location within the same array, typically through a predefined probing sequence.
   - Common probing techniques include linear probing (move to the next slot), quadratic probing (move to slots with quadratic increments), and double hashing (use a second hash function for probing).
   - Open addressing directly stores all keys in the hash table itself.

3. **Robin Hood Hashing:**

   - Robin Hood hashing is a variation of open addressing.
   - When a collision occurs, the new key is placed in the first available slot along the probing sequence.
   - If the newly inserted key displaces an existing key with more or equal probe steps, it is moved forward in the probing sequence until it finds an empty slot.

4. **Cuckoo Hashing:**

   - Cuckoo hashing uses multiple hash functions to distribute keys across two or more hash tables.

- When a collision occurs in one table, the key is moved to the other table using a different hash function.
- This process can repeat until a cycle is detected, which may trigger table resizing.

5. **Double Hashing:**
    - Double hashing is another open addressing technique that uses a secondary hash function to determine the step size when probing.
    - It helps avoid clustering and provides more uniform distribution of keys in the table.
6. **Linear Probing:**
    - Linear probing is a simple open addressing technique where, if a collision occurs, you move to the next slot linearly (with a fixed step) until an empty slot is found.
7. **Quadratic Probing:**
    - Quadratic probing is an open addressing technique where the probing sequence follows a quadratic pattern, typically with step sizes of 1, 4, 9, 16, and so on.
8. **External Chaining:**
    - In external chaining, a separate data structure (e.g., linked list or tree) is used to store keys that collide.
    - Each slot in the hash table points to this external structure.
9. **Rehashing:**
    - Rehashing involves resizing the hash table to a larger size when the load factor (number of keys / table size) exceeds a certain threshold.
    - All keys are rehashed and redistributed to the new table.

The choice of a collision resolution technique depends on the specific use case, the characteristics of the keys, and the expected behavior of the hash table. Each method has its advantages and trade-offs, and the choice of method can significantly impact the efficiency and performance of a hash table.

29. Why hashing technique is required? Explain different hashing techniques.

Hashing is a fundamental technique in computer science and data management that is required for various reasons. It allows for efficient data retrieval and storage, particularly when dealing with large datasets or when searching for specific items in a collection. Here's why hashing techniques are essential and an explanation of different hashing techniques:

Why Hashing Techniques are Required:

1. **Fast Data Retrieval:** Hashing enables quick data retrieval by mapping a key (or data) to an index in an array or table. This index can be used to directly access the desired information, reducing the time complexity of search operations.
2. **Efficient Data Storage:** Hashing techniques can efficiently store and manage data by using a fixed-size array or table. This simplifies data organization and management.

3. **Reduced Search Time:** Hashing minimizes the search time for specific items, making it suitable for applications like databases, dictionaries, and caches where rapid data access is crucial.

4. **Security and Data Integrity:** Hash functions are used in data security and integrity verification. Cryptographic hash functions can verify the integrity of data by generating a fixed-size hash code for a given input, making it challenging for attackers to tamper with data.

5. **Load Balancing:** In distributed systems, hashing is used for load balancing. By mapping keys to different servers or nodes, it ensures that data distribution is even across multiple resources.

6. **Unique Identification:** Hashing can generate unique identifiers for data elements, making it useful for scenarios like indexing, unique key generation, and data deduplication.

Different Hashing Techniques:

1. **Division Method:** This basic hashing technique calculates the hash value by taking the remainder of the key when divided by the table size. The formula is: `hash(key) = key % table_size`.

2. **Multiplication Method:** In this method, you multiply the key by a constant (usually between 0 and 1) and then take the fractional part of the result to calculate the hash value.

3. **Mid Square Method:** This method squares the key and extracts a middle portion of the result as the hash value.

4. **Folding Method:** In this technique, the key is divided into parts (e.g., chunks of digits), and these parts are added together to compute the hash value.

5. **Universal Hashing:** Universal hashing involves selecting a random hash function from a family of hash functions for each table. This minimizes the likelihood of collisions, making it useful in security and cryptography.

6. **Perfect Hashing:** Perfect hashing aims to create a hash function that eliminates collisions, ensuring that each key maps to a unique index in the table. This is useful in scenarios where collisions are not permissible.

7. **Cryptographic Hash Functions:** These are specialized hash functions designed to provide data integrity and security. They generate fixed-size hash codes that are extremely sensitive to changes in input data. Examples include SHA-256 and MD5.

8. **Consistent Hashing:** Primarily used in distributed systems, consistent hashing assigns keys to nodes or servers using a hash function. It minimizes data movement when the number of nodes changes.

9. **Custom Hash Functions:** In some cases, custom hash functions are designed to address specific needs or requirements of an application. These functions can be tailored to the characteristics of the data being hashed.

Each hashing technique has its strengths and weaknesses, and the choice of which one to use depends on the specific application and requirements. The effectiveness of hashing largely depends on the quality of the chosen hash function, which should aim to distribute keys as evenly as possible across the hash table to minimize collisions and optimize data retrieval and storage.

30. Explain with example various collision resolution techniques used in closed hashing (Array based).

Hashing Techniques

```
Hashing is a fundamental technique in computer science and data management. It
allows for efficient data retrieval and storage, especially when dealing with
large datasets or searching for specific items in a collection. There are various
hashing techniques for different applications:
```

1. **Division Method:**
   - Calculate the hash value by taking the remainder of the key when divided by the table size.
   - Formula: `hash(key) = key % table_size`.
2. **Multiplication Method:**
   - Multiply the key by a constant (usually between 0 and 1) and take the fractional part of the result.
3. **Mid Square Method:**
   - Square the key and extract a middle portion of the result as the hash value.
4. **Folding Method:**
   - Divide the key into parts (e.g., chunks of digits) and add these parts to compute the hash value.
5. **Universal Hashing:**
   - Select a random hash function from a family of hash functions for each table.
   - Minimizes the likelihood of collisions.
6. **Perfect Hashing:**
   - Create a hash function that eliminates collisions, ensuring each key maps to a unique index in the table.
7. **Cryptographic Hash Functions:**
   - Generate fixed-size hash codes that are sensitive to changes in input data.
   - Examples: SHA-256, MD5.
8. **Consistent Hashing:**
   - Assign keys to nodes or servers using a hash function.
   - Minimizes data movement when the number of nodes changes.
9. **Custom Hash Functions:**
   - Design custom hash functions to address specific application needs.

Collision Resolution Techniques

```
In hash tables, collisions occur when two or more keys hash to the same index.
Several collision resolution techniques are used to manage collisions:
```

1. **Separate Chaining:**
   - Store multiple values that map to the same index in a linked list or another data structure.
2. **Open Addressing (Closed Hashing):**
   - Use a predefined probing sequence to find another location within the same array for the new key.
   - **Linear Probing:**
     - Probe linearly (with a fixed step) until an empty slot is found.
   - **Quadratic Probing:**
     - Probe with a quadratic pattern using step sizes like 1, 4, 9, and so on.
   - **Double Hashing:**
     - Use a secondary hash function to determine the step size for probing.
3. **Robin Hood Hashing:**
   - Move keys forward in the probing sequence if they displace keys with more or equal probe steps.
4. **Cuckoo Hashing:**
   - Use multiple hash tables and rehash keys to distribute them across tables.
5. **External Chaining:**
   - Store keys in separate data structures (e.g., linked lists or trees) when collisions occur.
6. **Rehashing:**
   - Resize the hash table when the load factor exceeds a threshold.

The choice of a collision resolution technique depends on the specific application and the behavior of the hash table.

---

31. Discuss the implementation of naïve string-matching algorithm.

```
The naive string-matching algorithm, also known as the brute-force or
straightforward string-matching algorithm, is a simple method for finding all
occurrences of a pattern (substring) within a text (string). It works by comparing
the pattern against all possible substrings of the text and noting where the
pattern matches. Here's an overview of its implementation:
```

Naive String-Matching Algorithm:

1. Start with the first character of the text and slide the pattern over it.
2. At each position, compare each character of the pattern with the corresponding character in the text.
3. If a mismatch is found, move the pattern one position to the right and continue the comparison.
4. If a match is found for all characters of the pattern, record the position of the match in the text.
5. Continue this process until the entire text has been searched for occurrences of the pattern.

Pseudocode:

```
function naiveStringMatch(text, pattern):
    n = length(text)
    m = length(pattern)

    for i from 0 to n - m:
        j = 0
        while j < m and text[i + j] == pattern[j]:
            j = j + 1

        if j == m:
            print("Pattern found at position", i)
```

Example:

Let's say we have a text "ABABDABACDABABCABAB" and we want to find the pattern "ABABCABAB". The naive string-matching algorithm would find the pattern at positions 0, 10, and 14 in the text.

The naive string-matching algorithm is simple to understand and implement but is not the most efficient. Its time complexity is O((n-m+1)*m), where n is the length of the text, and m is the length of the pattern.

32. Discuss the implementation of Rabin-Karp string matching algorithm.

```
The Rabin-Karp string matching algorithm is a more efficient algorithm for finding
occurrences of a pattern (substring) within a text (string) than the naive string-
matching algorithm. It uses a rolling hash function to quickly compare patterns
and substrings of the text. Here's an overview of its implementation:
```

Rabin-Karp String Matching Algorithm:

1. Calculate the hash value for the pattern and the initial substring of the text of the same length.
2. Compare the hash values. If they match, perform a full character-by-character comparison to confirm.
3. If the hash values do not match, slide the pattern one position to the right and recalculate the hash value for the new substring of the text.
4. Repeat steps 2 and 3 until the entire text has been searched for occurrences of the pattern.

Rolling Hash Function:

The key to the Rabin-Karp algorithm is the rolling hash function, which allows efficient calculation of the hash value for a new substring based on the previous substring's hash value. A simple rolling hash function can use polynomial hashing:

```
# Calculate the hash value of a string using polynomial rolling hash
def hash_string(s, prime, modulus):
    hash_value = 0
    for char in s:
        hash_value = (hash_value * prime + ord(char)) % modulus
    return hash_value
```

Pseudocode:

```
function rabinKarp(text, pattern):
    n = length(text)
    m = length(pattern)
    prime = a prime number
    modulus = a large prime number

    pattern_hash = hash_string(pattern, prime, modulus)
    text_hash = hash_string(text[0:m], prime, modulus)

    for i from 0 to n - m:
        if text_hash == pattern_hash:
            if text[i:i+m] == pattern:
                print("Pattern found at position", i)

        if i < n - m:
            text_hash = (text_hash - ord(text[i]) * (prime^(m-1))) % modulus
            text_hash = (text_hash * prime + ord(text[i+m])) % modulus
            if text_hash < 0:
                text_hash = text_hash + modulus
```

Example:

Let's say we have a text "ABABDABACDABABCABAB" and we want to find the pattern "ABABCABAB" using Rabin-Karp. The algorithm would find the pattern at positions 0, 10, and 14 in the text.

The Rabin-Karp algorithm is efficient for string matching, with a typical time complexity of O((n-m+1)*m) in practice, where n is the length of the text, and m is the length of the pattern.

33.Discuss the implementation of KMP string matching algorithm.

```
The Knuth-Morris-Pratt (KMP) string matching algorithm is an efficient algorithm
for finding occurrences of a pattern (substring) within a text (string). It works
by using a preprocessing step to build a "partial match" table, which is then used
```

```
to avoid unnecessary character comparisons during the search phase. Here's an
overview of its implementation:
```

KMP String Matching Algorithm:

1. Preprocessing Phase: Build a partial match table (also known as the "failure function" or "pi function") for the pattern.
2. Search Phase: Compare the characters of the text and pattern while using the partial match table to skip unnecessary comparisons.

Partial Match Table (Pi Function):

The partial match table is constructed during the preprocessing phase and helps determine how much the pattern can be shifted when a mismatch occurs. It's essentially a table of the longest proper prefixes that are also suffixes of the pattern.

Pseudocode:

```
# Preprocessing phase to build the partial match table
function buildPartialMatchTable(pattern):
    m = length(pattern)
    pi = [0] * m
    j = 0

    for i from 1 to m-1:
        while j > 0 and pattern[i] != pattern[j]:
            j = pi[j-1]

        if pattern[i] == pattern[j]:
            j = j + 1
        pi[i] = j

    return pi

# Search phase using the partial match table
function kmpSearch(text, pattern):
    n = length(text)
    m = length(pattern)
    pi = buildPartialMatchTable(pattern)
    j = 0

    for i from 0 to n-1:
        while j > 0 and text[i] != pattern[j]:
            j = pi[j-1]

        if text[i] == pattern[j]:
            j = j + 1

        if j == m:
            print("Pattern found at position", i - j + 1)
            j = pi[j-1]
```

Example:

Let's say we have a text "ABABDABACDABABCABAB" and we want to find the pattern "ABABCABAB" using the KMP algorithm. The algorithm would find the pattern at positions 10 and 14 in the text.

The KMP algorithm is efficient with a time complexity of O(n + m), where n is the length of the text, and m is the length of the pattern. It avoids unnecessary character comparisons, making it especially efficient for long texts and patterns.

34.Discuss the implementation of Automata based string matching algorithm.

```
The Automata-based string matching algorithm, specifically the Finite Automaton
(FA) or Finite State Machine (FSM) algorithm, is a method for efficiently finding
occurrences of a pattern (substring) within a text (string). It constructs a
finite automaton that recognizes the pattern and then uses the automaton to search
the text. Here's an overview of its implementation:
```

Automata-based String Matching Algorithm:

1. Preprocessing Phase: Construct a finite automaton that represents the pattern.
2. Search Phase: Use the automaton to process the text while following its transitions to find occurrences of the pattern.

Finite Automaton:

The finite automaton is essentially a graph where nodes represent states and edges represent transitions between states based on the characters in the pattern. The automaton allows for efficient tracking of the pattern in the text.

Pseudocode:

```
# Preprocessing phase to construct the finite automaton
function buildFiniteAutomaton(pattern, alphabet):
    m = length(pattern)
    automaton = {}  # A dictionary to store transitions

    for q from 0 to m:
        for char in alphabet:
            k = min(m, q + 1)
            while pattern[q] != char and k > 0:
                k = k - 1
            automaton[(q, char)] = k
    return automaton

# Search phase using the finite automaton
function automataSearch(text, pattern, automaton):
    n = length(text)
    m = length(pattern)
    q = 0  # Initial state

    for i from 0 to n-1:
        if (q, text[i]) in automaton:
            q = automaton[(q, text[i])]
            if q == m:
                print("Pattern found at position", i - m + 1)

# Example usage
pattern = "ababc"
text = "abababcababcababcababc"
alphabet = set(text)  # Collect the alphabet from the text
automaton = buildFiniteAutomaton(pattern, alphabet)
automataSearch(text, pattern, automaton)
```

Example:

In the example provided, the automaton-based string matching algorithm efficiently finds the pattern "ababc" at positions 4, 10, and 16 in the text "abababcababcababcababc."

Automaton-based algorithms are efficient with a linear time complexity, making them suitable for large texts and patterns. They are particularly useful when searching for multiple patterns within a text, as the automaton can be reused for different patterns.

35. Implement the Huffman algorithm for string compression.

```
The Huffman coding algorithm is a widely used technique for lossless data
compression. It assigns variable-length codes to input characters, with shorter
codes assigned to more frequently occurring characters. Here's an implementation
of the Huffman algorithm for string compression in Python:
```

```
import heapq
from collections import defaultdict, Counter

class HuffmanNode:
    def __init__(self, char, freq):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other):
        return self.freq < other.freq
```

```python
def build_huffman_tree(text):
    char_freq = Counter(text)
    min_heap = [HuffmanNode(char, freq) for char, freq in char_freq.items()]
    heapq.heapify(min_heap)

    while len(min_heap) > 1:
        left = heapq.heappop(min_heap)
        right = heapq.heappop(min_heap)
        merged_node = HuffmanNode(None, left.freq + right.freq)
        merged_node.left = left
        merged_node.right = right
        heapq.heappush(min_heap, merged_node)

    return min_heap[0]

def build_huffman_codes(root, code, huffman_codes):
    if root is None:
        return
    if root.char:
        huffman_codes[root.char] = code
    build_huffman_codes(root.left, code + "0", huffman_codes)
    build_huffman_codes(root.right, code + "1", huffman_codes)

def compress_string(text):
    root = build_huffman_tree(text)
    huffman_codes = {}
    build_huffman_codes(root, "", huffman_codes)
    compressed_text = "".join(huffman_codes[char] for char in text)
    return compressed_text, root

def decompress_string(compressed_text, root):
    decompressed_text = []
    current = root

    for bit in compressed_text:
        if bit == "0":
            current = current.left
        else:
            current = current.right

        if current.char:
            decompressed_text.append(current.char)
            current = root

    return "".join(decompressed_text)

# Example usage
text = "this is an example for huffman encoding"
compressed_text, huffman_tree = compress_string(text)
print(f"Original text: {text}")
print(f"Compressed text: {compressed_text}")
decompressed_text = decompress_string(compressed_text, huffman_tree)
print(f"Decompressed text: {decompressed_text}")
```

In this code, we first build a Huffman tree for the input text, then generate Huffman codes for each character. The `compress_string` function compresses the input text using the generated codes, and the `decompress_string` function decompresses the compressed text using the Huffman tree.

This implementation demonstrates how the Huffman algorithm can be used for string compression and decompression.

36.Discuss the implementation of Tries.

```
A Trie (pronounced "try") is a tree-like data structure used for efficient
retrieval of a key in a large dataset of strings or words. It is commonly employed
in text-related applications like autocomplete, spell-checking, and dictionary
searching. Tries are particularly useful when you need to search for words with
common prefixes.
```

Here's a high-level overview of the implementation of Tries:

Trie Node: The basic building block of a Trie is a node. Each node typically contains the following components:

- A character: Representing a character in the key.
- Child nodes: Pointers to child nodes, each corresponding to a possible next character.
- A flag (or boolean): Indicating if the node represents the end of a valid key (word).

Trie Structure: The Trie structure consists of a root node, from which all other nodes are descended. The root node usually does not contain a character but serves as the starting point for searching.

Insertion: To insert a key (a word) into a Trie, you start from the root and follow a path through the Trie according to the characters in the key. If a node representing a character does not exist, you create it. Repeat this process for each character in the key, marking the last node as the end of a valid key (word).

Searching: To search for a key in a Trie, you start from the root and follow the path based on the characters in the key. If you reach the end of the key and the last node is marked as the end of a valid key, the key is found.

Deletion: Deleting a key from a Trie involves marking the last node of the key as invalid and potentially removing any empty branches. Deletion in Tries can be more complex because you need to handle various scenarios like nodes with shared prefixes and non-shared branches.

Common Operations:

- Insertion of a key.
- Searching for a key.
- Deletion of a key.
- Prefix search: Finding all keys with a common prefix.
- Finding all keys in lexicographic order (e.g., for dictionary applications).

Implementation Example in Python:

Here's a simplified Python implementation of a Trie data structure with insertion and search operations:

```python
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True

    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word

# Example usage:
trie = Trie()
trie.insert("apple")
trie.insert("app")
print(trie.search("apple"))  # Output: True
print(trie.search("app"))    # Output: True
print(trie.search("ap"))     # Output: False
```