**Data Structures and Algorithms V**

**DTD117V**

**Assignment 3**

**Group 20**

**Members:**

|  |  |
| --- | --- |
| **Names** | **Student No** |
| Ntsundeni ME | 215382354 |
| Nana AG | 220114228 |
| Tsita MT | 211380373 |
| Ramolotsha MSA | 216324986 |
| Maila MB | 213543407 |
| Phalanndwa T | 215556212 |

**Question 1**

Discuss the basic concepts of graph data structures and their applications in real-world

scenarios. Explain different types of graphs and their representations. [5 Marks]

**Applications of Graphs:**

* **Vertices and Edges:** Vertices or nodes represent entities, while edges represent relationships or connections between these entities.
* **Directed vs. Undirected Graphs:** Directed graphs edges have a direction, indicating a one-way relationship between vertices. Undirected graphs, edges have no direction, indicating a two-way relationship.
* **Weighted vs. Unweighted Graphs:** Edges in weighted graphs have associated weights or costs, representing the strength or distance of the relationship between vertices.
* **Cycles and Acyclic Graphs:** A cycle is a path in a graph that starts and ends at the same vertex. Graphs without cycles are called acyclic.

**Types of Graphs:**

* **Directed Graphs:** Each edge has a direction, indicating a one-way relationship between vertices.
* **Undirected Graphs:** Edges have no direction, representing a two-way relationship between vertices.
* **Weighted Graphs:** Edges have weights, representing the cost or distance associated with the relationship between vertices.
* **Unweighted Graphs:** Edges have no weights, indicating equal cost or distance between vertices.

**Representation of Graphs:**

* **Adjacency Matrix**: A 2D array where the value at index (i, j) represents whether there is an edge between vertex i and vertex j. Suitable for dense graphs but requires more memory for sparse graphs.
* **Adjacency List:** Each vertex is associated with a list of its adjacent vertices. Suitable for sparse graphs, as it requires less memory compared to an adjacency matrix.
* **Edge List:** A list of all edges in the graph, each represented as a tuple (u, v) where u and v are vertices connected by the edge. Simple and efficient but lacks direct access to adjacency information.

**Ouestion2**

Explain the concept of hash tables and their role in efficient data storage and retrieval.

Discuss collision resolution techniques used in hash tables and their impact on

Performance. [5 Marks]

**Concept of Hash Tables:**

* **Hash Function:** A hash function takes a key as input and returns an index in the hash table where the corresponding value is stored. Ideally, the hash function distributes keys evenly across the hash table to minimize collisions and ensure efficient retrieval.
* **Array (Hash Table):** Hash tables are typically implemented using arrays, where each index corresponds to a "bucket" that can store multiple key-value pairs. Each bucket may store multiple entries due to collisions.
* **Collision Handling:** Collisions occur when two or more keys are mapped to the same index by the hash function. Efficient collision resolution techniques are crucial for maintaining performance and ensuring that all key-value pairs can be stored and retrieved correctly.
* Role in Efficient Data Storage and Retrieval:
* **Constant-Time Retrieval:** Hash tables offer constant-time retrieval complexity (O(1)) on average. This means that regardless of the size of the data set, the time taken to access a value associated with a key remains constant.
* **Space Efficiency:** Hash tables utilize space efficiently, as they only allocate memory proportional to the number of key-value pairs stored, rather than reserving space for all possible keys.
* **Flexibility:** Hash tables can accommodate dynamic data sets, allowing for easy insertion, deletion, and modification of key-value pairs.

**Collision Resolution Techniques:**

* **Separate Chaining:** In separate chaining, each bucket in the hash table stores a linked list or another data structure to handle multiple entries that hash to the same index. This technique ensures that all key-value pairs are stored and retrieved correctly but may result in increased memory usage and slower performance for large lists.
* **Linear Probing:** With linear probing, collisions are resolved by placing the colliding key-value pairs in the next available (unoccupied) slot in the hash table. This method ensures better memory locality and cache performance but may lead to clustering and decreased performance if not implemented carefully.
* **Quadratic Probing:** Quadratic probing addresses clustering issues by using a quadratic function to determine the next probing position in case of collisions. This technique helps distribute entries more evenly across the hash table, reducing the likelihood of long probe sequences.
* **Double Hashing:** Double hashing combines the use of two hash functions to calculate alternative probe positions in case of collisions. This method helps reduce clustering and provides better performance compared to linear probing and quadratic probing in certain scenarios.

**Question 3**

Implement a graph data structure in Java, C++ or Python and define methods to add

vertices and edges. Demonstrate the usage of these methods by creating a graph and

adding vertices and edges to it. [10 Marks]

import java.util.ArrayList;

import java.util.List;

public class Graph {

private List<List<Integer>> adjList;

private int numVertices;

public Graph(int numVertices) {

this.numVertices = numVertices;

adjList = new ArrayList<>(numVertices);

for (int i = 0; i < numVertices; i++) {

adjList.add(new ArrayList<>());

}

}

public void addVertex() {

numVertices++;

adjList.add(new ArrayList<>());

}

public void addEdge(int source, int destination) {

if (source >= 0 && source < numVertices && destination >= 0 && destination <

numVertices) {

adjList.get(source).add(destination);

} else {

System.out.println("Invalid vertex");

}

}

public void printGraph() {

for (int i = 0; i < numVertices; i++) {

System.out.print("Adjacency list of vertex " + i + " head ");

for (int neighbor : adjList.get(i)) {

System.out.print("-> " + neighbor);

}

System.out.println();

}

}

public static void main(String[] args) {

Graph graph = new Graph(5);

graph.addEdge(0, 1);

graph.addEdge(0, 4);

graph.addEdge(1, 2);

graph.addEdge(2, 3);

graph.addEdge(3, 0);

graph.printGraph();

}

}

**Question 4**

Implement a hash table data structure in Java, C++ or Python using separate chaining

collision resolution technique. Define methods to insert, delete, and search for key-value

pairs. Demonstrate the usage of these methods by inserting, deleting, and searching for

key-value pairs. [10 Marks]

class ListNode:

def \_\_init\_\_(self, key, value):

self.key = key

self.value = value

self.next = None

class HashTable:

def \_\_init\_\_(self, size):

self.size = size

self.buckets = [None] \* size

def hash\_function(self, key):

return hash(key) % self.size

def insert(self, key, value):

index = self.hash\_function(key)

if self.buckets[index] is None:

self.buckets[index] = ListNode(key, value)

else:

current = self.buckets[index]

while current.next:

current = current.next

current.next = ListNode(key, value)

def search(self, key):

index = self.hash\_function(key)

current = self.buckets[index]

while current:

if current.key == key:

return current.value

current = current.next

return None

def delete(self, key):

index = self.hash\_function(key)

current = self.buckets[index]

prev = None

while current:

if current.key == key:

if prev:

prev.next = current.next

else:

self.buckets[index] = current.next

return

prev = current

current = current.next

def display(self):

for i in range(self.size):

current = self.buckets[i]

while current:

print(f"Key: {current.key}, Value: {current.value}")

current = current.next

hash\_table = HashTable(10)

hash\_table.insert("apple", 10)

hash\_table.insert("banana", 20)

hash\_table.insert("orange", 30)

print("Searching for 'banana':", hash\_table.search("banana"))

print("Searching for 'grape':", hash\_table.search("grape"))

hash\_table.delete("banana")

print("After deleting 'banana':")

hash\_table.display()

**Question 5**

Implement a hash table data structure in Java, C++ or Python using linear probing

collision resolution technique. Define methods to insert, delete, and search for key-value

pairs. Demonstrate the usage of these methods by inserting, deleting, and searching for

key-value pairs. [10 Marks]

class HashTable:

def \_\_init\_\_(self, size):

self.size = size

self.keys = [None] \* size

self.values = [None] \* size

def hash\_function(self, key):

return hash(key) % self.size

def insert(self, key, value):

index = self.hash\_function(key)

while self.keys[index] is not None:

index = (index + 1) % self.size

self.keys[index] = key

self.values[index] = value

def search(self, key):

index = self.hash\_function(key)

while self.keys[index] is not None:

if self.keys[index] == key:

return self.values[index]

index = (index + 1) % self.size

return None

def delete(self, key):

index = self.hash\_function(key)

while self.keys[index] is not None:

if self.keys[index] == key:

self.keys[index] = None

self.values[index] = None

return

index = (index + 1) % self.size

def display(self):

for i in range(self.size):

if self.keys[i] is not None:

print(f"Key: {self.keys[i]}, Value: {self.values[i]}")

hash\_table = HashTable(10)

hash\_table.insert("apple", 10)

hash\_table.insert("banana", 20)

hash\_table.insert("orange", 30)

print("Searching for 'banana':", hash\_table.search("banana"))

print("Searching for 'grape':", hash\_table.search("grape"))

hash\_table.delete("banana")

print("After deleting 'banana':")

hash\_table.display()