**EXP 10.A LINEAR SEARCH**

def linear\_search(arr, target):

# Step 1: Traverse the array using a for loop

for i in range(len(arr)):

# Step 2: Compare the target value with the current value of the array

if arr[i] == target:

# Step 2.1: If values match, return the current index

return i

# Step 3: If no match is found, return -1

return -1

# Example usage:

array = [10, 23, 45, 70, 11, 15]

target\_value = 15

# Call the linear search function

result = linear\_search(array, target\_value)

# Output the result

if result != -1:

print(f"Element found at index {result}")

else:

print("Element not found in the array")

**OUTPUT**

**Element found at index 5**

**EXP 10.B BINARY SEARCH**

def binary\_search(arr, target):

# Step 2: Initialize pointers

beg = 0

end = len(arr) - 1

# Step 4: Repeat until the search space is valid

while beg <= end:

mid = (beg + end) // 2 # Step 4: Find mid index

# Step 5: Check if target is at mid

if arr[mid] == target:

return mid # Target found, return index

# Step 6: Target is smaller than mid, discard right half

elif arr[mid] > target:

end = mid - 1

# Step 7: Target is larger than mid, discard left half

else:

beg = mid + 1

# If target is not found, return -1

return -1

# Test the function with an example array

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

target = 6

# Call the binary search function

result = binary\_search(arr, target)

if result != -1:

print(f"Element found at index {result}")

else:

print("Element not found in the array")

**OUTPUT**

**Element found at index 5**

**EXP 11.A HASHING USING LINEAR PROBING**

class HashTable:

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

self.size = size

self.table = [None] \* size

# Hash function: H(K) = K mod size

def hash\_function(self, key):

return key % self.size

# Insert a key into the hash table using linear probing to handle collisions

def insert(self, key, value):

index = self.hash\_function(key)

original\_index = index

probe\_count = 0

# Linear probing: find the next available slot

while self.table[index] is not None and self.table[index][0] != key:

print(f"Collision at index {index} for key {key}. Probing to next index.")

index = (index + 1) % self.size

probe\_count += 1

if index == original\_index: # Table is full

print("Hash table is full, cannot insert more elements.")

return

# Insert the key-value pair

self.table[index] = (key, value)

print(f"Key {key} inserted at index {index} after {probe\_count} probe(s).")

# Search for a key in the hash table

def search(self, key):

index = self.hash\_function(key)

original\_index = index

probe\_count = 0

# Linear probing: search for the key

while self.table[index] is not None:

if self.table[index][0] == key:

print(f"Key {key} found at index {index} after {probe\_count} probe(s).")

return self.table[index][1] # Return the value associated with the key

index = (index + 1) % self.size

probe\_count += 1

if index == original\_index: # Searched all slots

break

print(f"Key {key} not found after {probe\_count} probe(s).")

return None

# Display the hash table

def display(self):

print("Hash Table:")

for i, entry in enumerate(self.table):

if entry is None:

print(f"Index {i}: Empty")

else:

print(f"Index {i}: Key = {entry[0]}, Value = {entry[1]}")

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Create a hash table with 10 memory locations (size = 10)

ht = HashTable(10)

# Employee records with 4-digit keys (K)

employee\_records = [

(1234, "Alice"),

(5678, "Bob"),

(9876, "Charlie"),

(4321, "David"),

(2468, "Eve"),

]

# Insert employee records into the hash table

for emp\_id, name in employee\_records:

ht.insert(emp\_id, name)

# Display the hash table

ht.display()

# Search for a key in the hash table

ht.search(9876) # Should find Charlie

ht.search(9999) # Should not find any record

**OUTPUT**

**Key 1234 inserted at index 4 after 0 probe(s).**

**Key 5678 inserted at index 8 after 0 probe(s).**

**Key 9876 inserted at index 6 after 0 probe(s).**

**Key 4321 inserted at index 1 after 0 probe(s).**

**Collision at index 8 for key 2468. Probing to next index.**

**Key 2468 inserted at index 9 after 1 probe(s).**

**Hash Table:**

**Index 0: Empty**

**Index 1: Key = 4321, Value = David**

**Index 2: Empty**

**Index 3: Empty**

**Index 4: Key = 1234, Value = Alice**

**Index 5: Empty**

**Index 6: Key = 9876, Value = Charlie**

**Index 7: Empty**

**Index 8: Key = 5678, Value = Bob**

**Index 9: Key = 2468, Value = Eve**

**Key 9876 found at index 6 after 0 probe(s).**

**Key 9999 not found after 1 probe(s).**

**EXP 11.B HASHING USING QUADRATIC PROBING**

class HashTable:

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

# Initialize the hash table and mark all entries as None (empty)

self.size = size

self.table = [None] \* size

self.keys = [None] \* size # To track keys in the hash table

def hash\_function(self, key):

# Basic hash function using modulo operation

return key % self.size

def quadratic\_probe(self, key):

# Quadratic probing to resolve collisions

index = self.hash\_function(key)

i = 1

# If collision occurs, try quadratic probing

while self.table[index] is not None and self.keys[index] != key:

index = (index + i \*\* 2) % self.size

i += 1

return index

def insert(self, key, value):

# Insert a key-value pair into the hash table

index = self.quadratic\_probe(key)

if self.table[index] is None:

# New key, inserting

self.table[index] = value

self.keys[index] = key

print(f"Inserted key {key} at index {index}")

else:

# Key already exists, updating value

self.table[index] = value

print(f"Updated key {key} at index {index}")

def search(self, key):

# Search for a value by key

index = self.quadratic\_probe(key)

if self.keys[index] == key:

return self.table[index]

else:

return None

def remove(self, key):

# Remove a key-value pair from the hash table

index = self.quadratic\_probe(key)

if self.keys[index] == key:

self.table[index] = None

self.keys[index] = None

print(f"Removed key {key} from index {index}")

else:

print(f"Key {key} not found.")

def display(self):

# Display all the elements in the hash table

print("Hash Table:")

for i in range(self.size):

if self.table[i] is not None:

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

else:

print(f"Index {i}: Empty")

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

ht = HashTable(10)

ht.insert(10, "Value 10")

ht.insert(20, "Value 20")

ht.insert(30, "Value 30")

ht.insert(23, "Value 23")

ht.display()

print("Searching for key 23:", ht.search(23))

ht.remove(20)

ht.display()

**OUTPUT**

**Inserted key 10 at index 0**

**Inserted key 20 at index 1**

**Inserted key 30 at index 5**

**Inserted key 23 at index 3**

**Hash Table:**

**Index 0: Key = 10, Value = Value 10**

**Index 1: Key = 20, Value = Value 20**

**Index 2: Empty**

**Index 3: Key = 23, Value = Value 23**

**Index 4: Empty**

**Index 5: Key = 30, Value = Value 30**

**Index 6: Empty**

**Index 7: Empty**

**Index 8: Empty**

**Index 9: Empty**

**Searching for key 23: Value 23**

**Removed key 20 from index 1**

**Hash Table:**

**Index 0: Key = 10, Value = Value 10**

**Index 1: Empty**

**Index 2: Empty**

**Index 3: Key = 23, Value = Value 23**

**Index 4: Empty**

**Index 5: Key = 30, Value = Value 30**

**Index 6: Empty**

**Index 7: Empty**

**Index 8: Empty**

**Index 9: Empty**