National University of Computer and Emerging Sciences



# Lab Manual

*for*

# Data Structure

Department of Computer Science FAST-NU, Lahore, Pakistan

**Objectives:**

After performing this lab, students shall be able to revise:

* Hash Table

## Problem 1

### ****Hash Table for Integer Keys****

1. **Class Design:**  
   Design a class HashTable with the following structure:

class HashTable {

private:

int size; // Size of the hash table (odd number)

int\* arr; // Array to store keys

public:

// Constructor

HashTable(bool rehashFlag, int collisionMethod);

// Hash Table Functions

void Insert(int key);

void Delete(int key);

void Update(int key, int newValue);

int Access(int key);

};

1. **Constructor:**
   * The constructor accepts:
     + A boolean flag for rehashing (rehashFlag):
       - If rehashFlag is 0: Rehashing is disabled.
       - If rehashFlag is 1: Rehashing is enabled, and the hash table size doubles when the load factor exceeds 0.5.
     + An integer to select the collision resolution method:
       - Linear Probing
       - Linear Probing with Steps
       - Pseudo-Random Probing
       - Double Hashing
2. **Hashing Function:**  
   Use the modulo function (key % size) to compute the hash index.
3. **Collision Resolution Methods:**  
   Implement the following collision resolution techniques:
   * **Linear Probing:**  
     Use the formula:  
     i = p(k, i) = (hashIndex + i) % size  
     where i is the probe sequence number.
   * **Linear Probing with Steps:**  
     Use the formula:  
     i = p(k, i) = (hashIndex + i \* c) % size  
     where c is a prime number smaller than the hash table size.
   * **Pseudo-Random Probing:**  
     Use a permutation array perm[i] to determine the probe sequence:  
     i = p(k, i) = (hashIndex + perm[i]) % size Generate perm[i] randomly.
   * **Double Hashing:**  
     Use a secondary hash function:  
     hf2 = 1 + key % (size - 2)  
     and resolve collisions with:  
     i = (hashIndex + i \* hf2) % size
4. **Tasks:**
   * Create 4 objects of HashTable with the same keys, each using one of the above collision resolution methods.
   * Insert keys into the hash tables and record probe sequences.
   * Calculate the average access cost (number of probes) for:
     + Without rehashing.
     + With rehashing (when the load factor exceeds 0.5).

## Problem 2

### ****Hash Table for String Keys****

1. **Class Design:**  
   Design a hash table to store string keys and their associated values (air pollution indices). Use chaining for collision resolution.

class HashTable {

private:

struct Node {

std::string key; // City name

int value; // Air pollution index

Node\* next; // Pointer for chaining

};

int size; // Size of the hash table

Node\*\* table; // Array of linked lists

public:

HashTable(int size); // Constructor

void Insert(std::string key, int value);

void Delete(std::string key);

void Update(std::string key, int newValue);

int Access(std::string key);

};

1. **Hash Function:**  
   Use a polynomial hash function:  
   hashCode = Σ(key[i] \* a^i) % size  
   where a = 33.
2. **Collision Resolution:**  
   Use chaining to handle collisions. Each bucket in the hash table is a linked list.
3. **Application:**  
   Store the air pollution indices of 50 major cities in Pakistan:
   * **Key:** City name (e.g., "Karachi").
   * **Value:** Air pollution index percentage (e.g., 45%).

Implement the following scenarios using the hash table functions:

* + **Insertion:** Add air pollution data for 50 cities.
  + **Update:** Update the pollution index of a specific city.
  + **Access:** Retrieve the pollution index of a specific city.
  + **Deletion:** Remove data for a city if it is no longer needed.