|  |  |
| --- | --- |
| NAME | Sujal Dingakar |
| UID no | 2024301005 |
| Experiment No | 9 |
|  | |
| AIM: | Hashing Techniques: Write a program to implement Hash Table for the given input and solve thje collision using using quadratic probing and linear probing. |
| THEORY: | **Why Do We Use Hashing?**   * **Purpose of Hashing**: Explain why hashing is essential in data structures, especially in cases where fast data retrieval is required, like searching, inserting, or deleting elements. * **Advantages Over Other Structures**: Describe how hashing provides constant-time complexity on average (O(1)) for operations, which is more efficient than structures like arrays or linked lists for large data sets.   **What is Hashing and is type?**  Hashing is a data storage and retrieval method that maps data to a fixed-size numerical value, called a hash, which serves as an index in a hash table. The hash function generates this value, allowing for quick data access in constant time, O(1), making hashing valuable for applications requiring fast lookup, such as databases and caches.  **Types of Hashing Techniques for Collision Resolution**   1. **Linear Probing:**    * Concept: Moves sequentially to the next position until an empty spot is found.    * Pros: Simple to implement.    * Cons: May cause primary clustering, increasing lookup time as the table fills. 2. **Quadratic Probing:**    * Concept: Moves at quadratic intervals (e.g., i+12,i+22i+1^2, i+2^2i+12,i+22) to reduce clustering.    * Pros: Distributes data more evenly, reducing clustering.    * Cons: Can lead to secondary clustering, with some slots possibly missed. 3. **Double Hashing:**    * Concept: Uses a second hash function to calculate intervals, minimizing clustering.    * Pros: Spreads elements more uniformly, reducing both primary and secondary clustering.    * Cons: Requires two hash functions and the second hash must be non-zero to avoid loops.   **What Makes a Good Hash Function?**   * Distribute Data Uniformly: Avoids clustering by spreading values evenly across the hash table. * Be Deterministic: Produces the same hash value for the same input every time. * Minimize Collisions: Reduces the likelihood of different inputs producing the same hash. * Be Efficient to Compute: Quickly generates the hash, keeping operations fast. * Use the Full Range of Table: Maximizes table utilization by covering all possible indices.   **Advantages of Hashing Over Other Data Structures**   * **Efficiency:** Faster access to data compared to linear structures like linked lists or arrays. * **Memory Usage:** Discuss the memory efficiency of hashing, especially when hash functions are well-optimized. * **Suitability for Large Datasets:** Ideal for applications where high-speed access to large datasets is necessary.   **Operations That Can Be Performed on Hashing**   * **Insertion**: How items are added to a hash table. * **Searching:** How hashing allows for quick lookups of data. * **Deletion:** How items are removed from a hash table and how the table manages collisions during deletions. * **Handling Collisions:** Describe methods like chaining and open addressing to manage collisions, which occur when multiple items hash to the same index.   **Applications of Hashing**   * **Data Indexing**: Used in databases to quickly locate records. * **Cryptography:** Essential in encrypting data where a hash represents sensitive information. * **Caching:** Used to cache data efficiently, such as in web applications to speed up access. * **Checksum and Data Integrity:** Used to verify data integrity by comparing hashes.   **What is Collision, and How is it Handled in Hashing?**  A collision occurs in hashing when two different inputs produce the same hash value, causing them to map to the same index in a hash table. This can disrupt data retrieval, as it’s unclear which value to access at the shared index.  **Collisions are typically handled by collision resolution techniques such as:**   1. **Chaining:** Stores multiple elements at the same index using linked lists. 2. **Open Addressing:** Finds another open spot in the table based on a probing sequence (e.g., linear or quadratic probing, double hashing). |
| ALGORITHM**:** |  **Graph Representation:**   * Define a Node structure to represent each vertex's adjacency list. * Define a List structure to represent the linked list for each vertex. * Define a Graph structure that contains the number of vertices and an array of adjacency lists.    **Queue Data Structure:**   * Define a Queue structure for use in BFS traversal, including methods to enqueue and dequeue elements.    **Graph Initialization:**   * Create a graph using the createGraph function, which allocates memory for the graph and initializes the adjacency lists.    **Adding Edges:**   * Define the addEdge function to add directed edges between vertices. If you want an undirected graph, the same edge is added in both directions.    **BFS Traversal:**   * Initialize a queue to manage the BFS process. * Maintain an array to track the level of each vertex. * Use a visited array to keep track of visited vertices. * Start BFS from a specified vertex:   + Dequeue a vertex, mark it as visited, and record its level.   + Enqueue all unvisited adjacent vertices. * Print the BFS traversal order and levels of each vertex.    **DFS Traversal:**   * Initialize arrays to keep track of visited vertices, start times, and end times of each vertex. * Start DFS from a specified vertex:   + Mark the vertex as visited, record its start time, and store it in the DFS traversal order.   + Recursively visit all unvisited adjacent vertices.   + Record the finish time for each vertex upon returning from the recursive call. * Print the DFS traversal order, start times, and end times for each vertex.    **Input Handling:**   * Get the number of vertices and edges from the user. * Read edges from the user to build the graph and adjacency matrix for BFS. * Ask the user for the starting vertex for both DFS and BFS.    **Output:**   * Print the adjacency list representation of the graph. * Print the BFS and DFS traversal orders and relevant timing information.    **Memory Cleanup:**   * Free allocated memory for the adjacency lists and the graph structure at the end of the program. |
| PROBLEM SOLVING: |  |
| PROGRAM: | #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define TABLE\_SIZE 10  #define PRIME 7  typedef struct {  int \*table;  bool \*occupied;  } HashTable;  int hash(int key) {  return key % TABLE\_SIZE;  }  HashTable\* createHashTable() {  HashTable\* ht = (HashTable\*)malloc(sizeof(HashTable));  ht->table = (int\*)malloc(sizeof(int) \* TABLE\_SIZE);  ht->occupied = (bool\*)malloc(sizeof(bool) \* TABLE\_SIZE);  for (int i = 0; i < TABLE\_SIZE; i++) {  ht->table[i] = -1;  ht->occupied[i] = false;  }  return ht;  }  // Linear Probing  bool linearProbingInsert(HashTable\* ht, int key) {  int idx = hash(key);  int original\_idx = idx;  int step = -1;    // Show initial hash calculation  printf("%d %% %d = %d\n", key, TABLE\_SIZE, key % TABLE\_SIZE);    while (true) {  // After collision, calculate next position using linear probing  step++;  idx = (original\_idx + step) % TABLE\_SIZE;  printf("Step %d: (h(%d) + %d) mod %d = %d\n",  step, key, step, TABLE\_SIZE, idx);    // Check if current position is empty  if (!ht->occupied[idx]) {  ht->table[idx] = key;  ht->occupied[idx] = true;  printf("Inserted %d at index %d\n", key, idx);  return true;  }    // Position is occupied - show collision  printf("Collision occurred: position %d is occupied by %d\n",  idx, ht->table[idx]);    // Check if table is full  if (step >= TABLE\_SIZE - 1) {  printf("Table is full! Cannot insert %d\n", key);  return false;  }  }  }  // Quadratic Probing with detailed index calculation steps  bool quadraticProbingInsert(HashTable\* ht, int key) {  int idx = hash(key);  int original\_idx = idx;  int i = 0;  while (ht->occupied[idx]) {  printf("Collision occurred for %d at index %d\n", key, idx);  printf("Step %d: Original index = %d\n", i, original\_idx);  int probe\_value = (original\_idx + i \* i) % TABLE\_SIZE;  printf("Step %d: (%d + %d^2) mod %d = %d (new index)\n", i, original\_idx, i, TABLE\_SIZE, probe\_value);  idx = probe\_value;  i++;  if (i >= TABLE\_SIZE) {  return false;  }  }  ht->table[idx] = key;  ht->occupied[idx] = true;  printf("Inserted %d at index %d\n", key, idx);  return true;  }  // Double Hashing  int secondHash(int key) {  return PRIME - (key % PRIME);  }  bool doubleHashingInsert(HashTable\* ht, int key) {  int idx = hash(key);  int original\_idx = idx;  int step = secondHash(key);  int i = 0;  // Display initial hashing values  printf("Inserting %d:\n", key);  printf("h1(%d) = %d\n", key, idx);  while (ht->occupied[idx]) {  printf("Collision occurred for %d at index %d\n", key, idx);    printf("h2(%d) = %d - (%d mod %d) = %d\n", key, PRIME, key, PRIME, step);    // Calculate the new index  idx = (original\_idx + (i \* step)) % TABLE\_SIZE;    // Show new index calculation  printf("Step %d: (h(%d) + %d \* %d) mod %d = %d\n", i + 1, original\_idx, i, step, TABLE\_SIZE, idx);  // Increment the probe count  i++;  // Check if we have looped back to the original index    }  // If the slot is empty, insert the key  ht->table[idx] = key;  ht->occupied[idx] = true;  printf("Inserted %d at index %d\n", key, idx); // Print the final index  return true;  }  // Display the hash table with a title  void displayHashTable(HashTable\* ht, const char\* title) {  printf("%s:\n", title);  for (int i = 0; i < TABLE\_SIZE; i++) {  if (ht->occupied[i]) {  printf("Index %d: %d\n", i, ht->table[i]);  } else {  printf("Index %d: Empty\n", i);  }  }  printf("\n");  }  // Free the hash table  void freeHashTable(HashTable\* ht) {  free(ht->table);  free(ht->occupied);  free(ht);  }  int main() {  HashTable\* ht = createHashTable();  int choice, numKeys, key;  while (1) {  printf("Menu:\n");  printf("1. Insert elements into the hash table\n");  printf("2. Print the hash table\n");  printf("3. Exit\n");  printf("Enter your choice: ");  scanf("%d", &choice);  switch (choice) {  case 1:  printf("Enter the number of keys to insert (max %d): ", TABLE\_SIZE);  scanf("%d", &numKeys);  if (numKeys > TABLE\_SIZE) {  printf("Number of keys exceeds table size.\n");  break;  }    printf("Choose probing method:\n");  printf("1. Linear Probing\n");  printf("2. Quadratic Probing\n");  printf("3. Double Hashing\n");  printf("Enter your choice: ");  int probingChoice;  scanf("%d", &probingChoice);  if (probingChoice < 1 || probingChoice > 3) {  printf("Invalid probing method.\n");  break;  }    // Prompt user for keys  for (int i = 0; i < numKeys; i++) {  printf("Enter key %d: ", i + 1);  scanf("%d", &key);  switch (probingChoice) {  case 1:  linearProbingInsert(ht, key);  break;  case 2:  quadraticProbingInsert(ht, key);  break;  case 3:  doubleHashingInsert(ht, key);  break;  }  }  break;  case 2:  displayHashTable(ht, "Hash Table");  break;  case 3:  printf("Exist from loop..\n");  return(0);  break;  default:  printf("Invalid choice. Please try again.\n");  }  }  return 0;  } |
| RESULT :- | 1.**Linear Probing..**            2.**Quadratic Probing…**              3.**Double Probing..** |
| CONCLUSION :- | After implementing hashing with linear probing, quadratic probing, and double hashing in C, I learned how each technique addresses collisions and affects performance. Linear probing is simple but can cause clustering; quadratic probing reduces clustering but may leave some spots unused; double hashing provides the best distribution with minimal clustering but requires an extra hash function. Overall, each method balances trade-offs between simplicity, speed, and even data distribution in hash tables. |