|  |
| --- |
| A blue and white logo  Description automatically generated |
| **AVL, B-TREE, HASH TABLE**  **Name:** *MUHAMMAD IBTISAM AFZAL*  **Registration No:** *FA22-BCS-073*  **Submitted To:** *Mam Tahreem Saeed* |
|  |

**ASSIGNMENT NO. 4**

**Question 1:**

**AVL Tree:**

An AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one, ensuring that the tree remains balanced after every insertion and deletion operation. The balance is achieved through rotations.

**Insertion in AVL Tree:**

**Perform Standard BST Insertion:**

* Insert the new node in the AVL tree using standard binary search tree insertion.

**Update Heights:**

* Update the height of the current node.

**Rebalance the Tree:**

* Check the balance factor of each node in the path from the inserted node to the root.
* If the balance factor is violated (greater than 1 or less than -1), perform rotations to restore balance.

**Deletion in AVL Tree:**

**Perform Standard BST Deletion:**

* Delete the node as in a regular binary search tree.

**Update Heights:**

* Update the height of the current node.

**Rebalance the Tree:**

* Check the balance factor of each node in the path from the deleted node to the root.
* If the balance factor is violated, perform rotations to restore balance.

**B-tree:**

A B-tree is a self-balancing search tree that maintains sorted data and allows searches, insertions, deletions, and sequential access in logarithmic time. B-trees are commonly used in database systems and file systems.

**Insertion in B-tree:**

**Search and Insert:**

* Start at the root and traverse the tree to find the appropriate leaf node for insertion.

**Insert Key:**

* Insert the key into the leaf node. If the node overflows split it and promote the middle key to the parent.

**Update Parent Nodes:**

* If a split occurs, update the parent nodes. If the root is split, create a new root.

**Maintain B-tree Properties:**

Ensure that the B-tree properties (e.g., minimum, and maximum number of keys per node) are maintained after the insertion.

**Deletion in B-tree:**

**Search and Delete:**

* Start at the root and traverse the tree to find the key to be deleted.

**Delete Key:**

* If the key is in a leaf node, simply delete it. If it's in an internal node, replace it with its predecessor or successor and delete the predecessor or successor.

**Adjust Nodes:**

* If the deletion causes a node to have too few keys, borrow from or merge with its siblings to maintain the minimum number of keys.

**Update Parent Nodes:**

* If merging occurs, update the parent nodes. If the root has only one child, make that child the new root.

B-trees are designed to be more efficient for storage systems, especially in situations where data needs to be retrieved sequentially or in large blocks, making them well-suited for use in databases and file systems. They provide a good balance between read and write operations.

**Question 2:**

**Hashing:**

Hashing is a technique used in computer science to map data of arbitrary size to fixed-size values. The output of this process is often called a hash code or hash value. Hashing is widely used in various applications, such as data retrieval, password storage, and hash tables for efficient data storage and retrieval.

**Hash Function:**

A hash function is a mathematical function that takes an input (or 'key') and produces a fixed-size string of characters, which is typically a hash code. The primary goals of a good hash function are:

1. **Deterministic:** The same input should always produce the same hash code.
2. **Efficient:** The computation of the hash code should be fast.
3. **Uniform Distribution:** The hash codes should be distributed as evenly as possible across the output space.

**Hash Tables:**

A hash table is a data structure that uses a hash function to map keys to index locations, allowing for efficient insertion, deletion, and retrieval of data. The hash function determines where to store and look for the data. Each index in the hash table is often referred to as a "bucket," and each bucket can hold multiple items.

**Collision:**

A collision occurs when two different keys produce the same hash code, leading to a situation where two keys want to be stored in the same bucket of the hash table.

**Collision Resolution Strategies:**

**1. Separate Chaining:**

* **Description:**
* Each bucket in the hash table is associated with a linked list.
* When a collision occurs, the colliding elements are stored in the same bucket using a linked list.
* **Advantages:**
* Simple to implement.
* Handles any number of collisions.
* **Disadvantages:**
* Requires extra memory for linked lists.
* Access time can degrade in the case of a long-linked list.

**2. Open Addressing:**

* **Description:**
  + When a collision occurs, search for the next available slot in the hash table (linear probing, quadratic probing, or double hashing).
  + The idea is to find the next open slot in the hash table and insert the item there.
* **Advantages:**
  + No need for additional data structures (like linked lists).
  + More cache-friendly compared to separate chaining.
* **Disadvantages:**
  + Can lead to clustering, where consecutive slots are filled, causing more collisions.
  + May require careful tuning of probing strategies for optimal performance.

|  |  |
| --- | --- |
| **Example Code** | |
| #include <iostream>  #include <list>  Using namespace std;  const int TABLE\_SIZE = 10;  class HashTable {  private:  list<int> table[TABLE\_SIZE];  // Hash function  int hash(int key) {  return key % TABLE\_SIZE; }  public:  // Insert key into the hash table  void insert(int key) {  int index = hash(key);  table[index].push\_back(key); }  // Search for key in the hash table  bool search(int key) {  if (item == key) { | int index = hash(key);  for (int item : table[index]) {  return true; } }  return false; } };  int main() {  HashTable hashTable;  // Inserting keys into the hash table  hashTable.insert(5);  hashTable.insert(15);  hashTable.insert(25);  // Searching for keys in the hash table  cout << "Key 15 " << (hashTable.search(15) ? "found" : "not found") << endl;  cout << "Key 10 " << (hashTable.search(10) ? "found" : "not found") << endl;  return 0; } |