Hash Table with Linear Probing
Alexis Aguilar
CSCE 3110

Design of the Hash Table

The hash table is designed to efficiently store and retrieve key-value pairs while handling collisions using linear probing. The key features include:

Hash Function: Computes an index based on the key using a hash formula involving a prime multiplier (31).

Collision Handling: Uses linear probing to find the next available slot when collisions occur.

Dynamic Resizing: Automatically resizes the table when the load factor exceeds 0.7 to maintain performance.

Node Structure: Each slot in the table is represented by a HashNode, which stores the key, value, and a deletion marker (isDeleted) for logical deletion.

How Linear Probing Was Implemented

Linear probing is used to handle collisions, ensuring every key-value pair can find an appropriate slot without overwriting existing data:

Insert Operation: After calculating the hash index, if the slot is occupied, the program increments the index sequentially ```((index + 1) % capacity)``` until an empty slot is found.
If the key already exists in the table, its value is updated.

```
while (table[index] && !table[index]->isDeleted && table[index]->key != key) {
    index = (index + 1) % capacity; // Probe to the next index
}
```

Retrieve Operation: Similar to insertion, the program probes sequentially starting from the calculated index until the desired key is found or an empty slot is encountered.
If the key is found at an index, its associated value is returned.

```
for (int i = 0; i < capacity; ++i) {
    int probeIndex = (index + i) % capacity; // Probe sequentially
    if (!table[probeIndex]) break; // Stop search if slot is empty
    if (table[probeIndex]->key == key && !table[probeIndex]->isDeleted) {
        return table[probeIndex]->value; // Key found
    }
}
```

Delete Operation: The program uses linear probing to locate the key and marks the slot as deleted ```(isDeleted = true)```.
Marking the slot as deleted allows the table to maintain search continuity during linear probing while preventing incorrect results.

```
```

```
for (int i = 0; i < capacity; ++i) {
    int probeIndex = (index + i) % capacity;
    if (!table[probeIndex]) return; // Key not found
    if (table[probeIndex]->key == key && !table[probeIndex]->isDeleted) {
        table[probeIndex]->isDeleted = true; // Mark as deleted
        --size;
        return;
    }
}
```

How Resizing and Load Factor Were Handled

Load Factor: The load factor is defined as the ratio of the number of active elements to the total capacity of the table:
> Load Factor = size / capacity

If the load factor exceeds 0.7, the table is resized to maintain efficient insertion and retrieval operations.

Resizing: The table's capacity is doubled when resizing is triggered.

During resizing: A new table with double the capacity is created.
All active elements are reinserted into the new table using the hash function.
Deleted slots are ignored during reinsertion, ensuring that the table only contains valid key-value pairs.

```
void resizeTable() {
    int oldCapacity = capacity;      // Store the old capacity
    capacity *= 2;                   // Double the capacity
    vector<HashNode*> oldTable = table; // Backup the old table

    // Clear and resize the new table
    table.clear();
    table.resize(capacity, nullptr);
    size = 0; // Reset size and reinsert active elements

    // Rehash and reinsert non-deleted elements
    for (int i = 0; i < oldCapacity; ++i) {
        if (oldTable[i] && !oldTable[i]->isDeleted) {
            insert(oldTable[i]->key, oldTable[i]->value);
        }
        delete oldTable[i]; // Free memory for old nodes
    }
}
```