3. Load Factor and Rehashing

3.1. Load Factor (?)

The **Load Factor (λ)** is a measure of how full the hash table is. It is a critical metric for performance.

- **Formula:** `λ = m/n`
  - `n` = total number of items stored in the table.
  - `m` = total size of the hash table (number of buckets).
- **Impact on Performance:**
  - **Separate Chaining:** λ can be greater than 1. A higher λ means longer linked lists, and search time degrades towards `O(λ)` or `O(n)` in the worst case.
  - **Open Addressing:** λ cannot be greater than 1. As λ approaches 1, it becomes extremely difficult and slow to find an empty slot, and performance grinds to a halt

3.2. Rehashing

To maintain good performance (close to `O(1)`), we must keep the load factor low. When the load factor exceeds a certain threshold (e.g., **λ > 0.75**), the table is "rehashed."

Rehashing is the process of creating a new, larger hash table and re-inserting all existing elements into it. Here is the **process** of rehashing:

- **Trigger:** The load factor exceeds a predefined threshold (e.g., 0.75).
- **Allocate:** Create a new, larger hash table (e.g., `2 * m`, or the next prime number).
- **Re-insert:** Iterate through **every element** in the old table.
- **Re-hash**: For each element, compute its **new hash value** based on the **new, larger table size**.
- **Insert**: Place the element in its new correct slot in the new table.
- **Deallocate:** Free the memory of the old table.

Rehashing is an `O(n)` operation. It is slow and computationally expensive, but it happens infrequently. Its cost is "amortized" over many `O(1)` insertions, so the average insertion time remains `O(1)`.

3.3 Rehashing Implementation

```
// We must track item count 'n' and table size 'm'

int n = 0; // Number of items
int m = 10; // Initial table size
std::vector<std::list<KeyValuePair>> table;
const float MAX_LOAD_FACTOR = 0.75;

void rehash() {
    // Get old table and size
    std::vector<std::list<KeyValuePair>> oldTable = table;
    int oldSize = m;

    // Allocate new, larger table
    m = m * 2;
    table.clear();
    table.resize(m);
    n = 0; // Reset item count

    // Re-insert all elements from old table
    for (int i = 0; i < oldSize; ++i) {
        for (auto& pair : oldTable[i]) {
            // Re-hash and insert into new table
            insert(pair.key, pair.value);
        }
    }
}

void insert(std::string key, int value) {
    // Check load factor before inserting
    if ((float)n / m > MAX_LOAD_FACTOR) {
        rehash();
    }

    // Hash the key (uses the new 'm' if rehashed)
    int index = hashFunction(key);

    // Traverse chain to find or update
    for (auto& pair : table[index]) {
        if (pair.key == key) {
            pair.value = value;
```

```
            return;
        }
    }

    // Not found, add new pair and increment item count
    table[index].push_back(KeyValuePair(key, value));
    n++;
}
```

Revision #1
Created 2025-11-04 15:25:17 UTC by RE
Updated 2025-11-04 15:25:53 UTC by RE