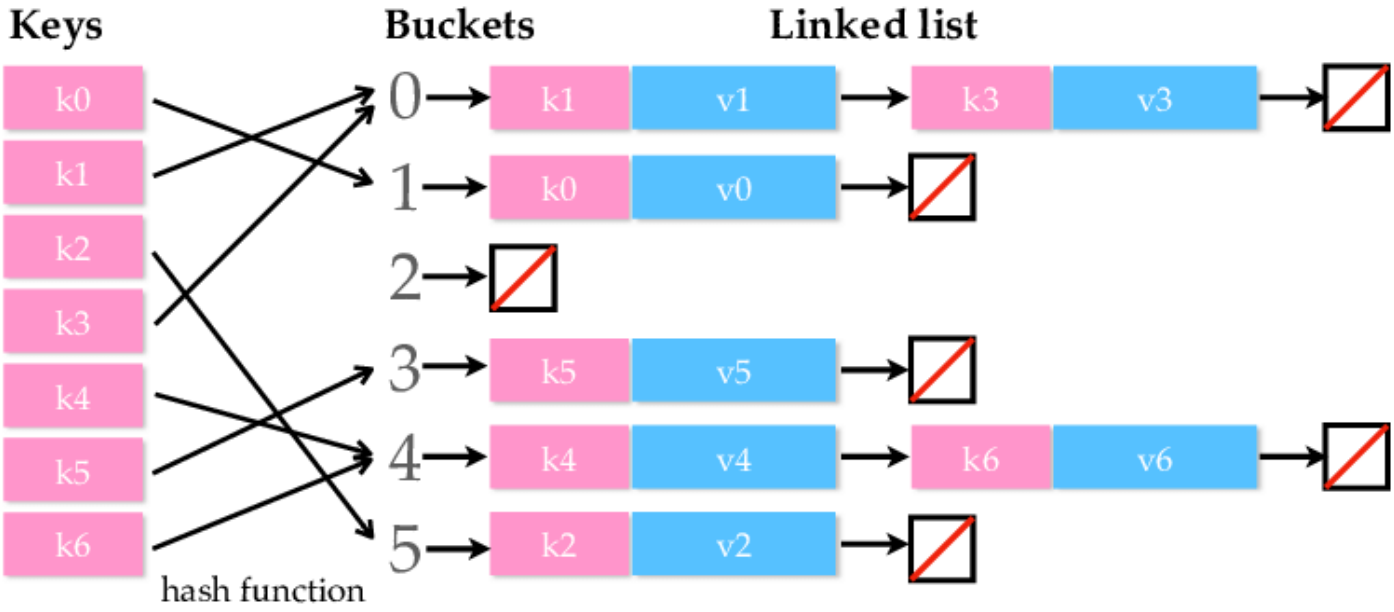
2. Collision Handling

A **Collision** occurs when two or more different keys produce **the same hash value** (index). For example, "Budi" and "Dina" are both hashed to **row #5**. We can't overwrite Budi's data. We must have a strategy to handle this.

2.1. Separate Chaining



This is the most common strategy to avoid collisions.

- **Concept:** Instead of each table row storing a single value, each row stores a pointer to another data structure, usually a **Linked List**.
- **How it Works:**
  1. "Budi" hashes to row 5. We create a linked list at row 5 and add "Budi".
  2. "Dina" hashes to row 5. We add "Dina" to the linked list that already exists at row 5.
  3. Row 5 now contains: `[Budi] -> [Dina] -> NULL`
- **Search:** To find "Dina," we compute its hash (5), go to row 5, and then traverse the linked list at that row until we find "Dina." The worst-case search time becomes `O(k)` where `k` is the number of elements in the chain.

2.1.1. Manual Implementation: Insertion

The `insert` function handles adding a new key-value pair. Its logic is:

1. Hash the `key` to find the correct bucket `index`.
2. Get the linked list (the chain) at that `index`.

3. Traverse the list:
    - If the `key` is already in the list, **update** its `value` and stop.
    - If the end of the list is reached and the `key` was not found, **add** the new key-value pair to the end of the list.

```
void insert(std::string key, int value) {
    // Hash the key to get the bucket index
    int index = hashFunction(key);

    // Get the chain (linked list) at that index
    std::list<KeyValuePair>& chain = table[index];

    // Traverse the chain
    for (auto& pair : chain) {
        if (pair.key == key) {
            // Key found: update the value and return
            pair.value = value;
            return;
        }
    }

    // Key not found: add new pair to the end of the list
    chain.push_back(KeyValuePair(key, value));
}
```

2.1.2. Manual Implementation: Searching

The `search` function finds the value associated with a key. Its logic is:

1. Hash the `key` to find the bucket `index`.
2. Get the linked list at that `index`.
3. Traverse the list:
    - If the `key` is found, return its `value`.
    - If the end of the list is reached and the `key` was not found, return a sentinel value (like -1) or throw an exception.

```
int search(std::string key) {
    // Hash the key to get the bucket index
    int index = hashFunction(key);

    // Get the chain at that index
```

```
    std::list<KeyValuePair>& chain = table[index];

    // Traverse the chain
    for (auto& pair : chain) {
        if (pair.key == key) {
            // Key found: return the value
            return pair.value;
        }
    }

    // Key not found
    return -1;
}
```

2.1.3. Manual Implementation: Deletion

The remove function deletes a key-value pair. Its logic is:

1. Hash the `key` to find the bucket `index`.
2. Get the linked list at that `index`.
3. Traverse the list using an **iterator**. We must use an iterator because we need to know the position of the element to delete it.
4. If the `key` is found:
   - Call the list's `erase()` method, passing the iterator.
   - Stop and return.

```
void remove(std::string key) {
    // Hash the key to get the bucket index
    int index = hashFunction(key);

    // Get a reference to the chain
    auto& chain = table[index];

    // Traverse the chain using an iterator
    for (auto it = chain.begin(); it != chain.end(); ++it) {
        // 'it' is an iterator (position pointer)
        // 'it->key' accesses the key of the element at that position
        if (it->key == key) {
            // Key found: erase the element at this position
            chain.erase(it);
```

```
            return;
        }
    }
    // If loop finishes, key was not found. Do nothing.
}
```

2.2. Open Addressing (Probing)

This is an alternative strategy where all data is stored within the table itself. No linked lists are used.

- **Concept:** If a collision occurs (the slot is full), we "probe" for the next empty slot according to a specific rule and place the item there.
- **Types of Probing:**
  1. **Linear Probing:** This is the simplest rule. If slot `h(key)` is full, try `h(key) + 1`, then `h(key) + 2`, `h(key) + 3`, and so on, wrapping around the table if necessary. The **problem** of this type is it tends to **create clusters of occupied slots**, which degrades performance as the table fills up.
  2. **Quadratic Probing:** If slot `h(key)` is full, try `h(key) + 1^2`, then `h(key) + 2^2`, `h(key) + 3^2`, and so on. This helps spread out elements better than linear probing.
  3. **Double Hashing:** Uses a second hash function to determine the "step size" for probing, which is the most effective at avoiding clusters.

Revision #2
Created 2025-11-04 15:23:05 UTC by RE
Updated 2025-11-04 16:05:59 UTC by RE