**Chaining in Hash Tables: A Strategy for Collision Resolution**

**Chaining** is one of the simplest and most effective methods to handle collisions in hash tables. Rather than storing the key-value pair directly in the array, chaining involves storing a **linked list** at each index of the array. When multiple keys hash to the same index (collision), they are added to the corresponding linked list.

**How Chaining Works:**

1. **Array of Linked Lists**:
   * Every index in the hash table contains a linked list. Initially, all lists are empty.
   * When a key-value pair is inserted into the hash table, the hash function calculates the **hash index**.
   * The key-value pair is then added to the linked list at that index.
2. **Handling Collisions**:
   * If multiple keys map to the same index, they are inserted into the same linked list. This avoids overflowing the array.
   * To search for a specific key, the linked list at the corresponding index is traversed.

**Example of Chaining in Action:**

Suppose you have 6 keys: k1,k2,k3,k4,k5,k6k\_1, k\_2, k\_3, k\_4, k\_5, k\_6k1​,k2​,k3​,k4​,k5​,k6​. These keys are inserted into a hash table using chaining, with the hash function producing the following results:

* hashIndex(k1,m)=0\text{hashIndex}(k\_1, m) = 0hashIndex(k1​,m)=0
* hashIndex(k2,m)=m−3\text{hashIndex}(k\_2, m) = m - 3hashIndex(k2​,m)=m−3
* hashIndex(k3,m)=m−1\text{hashIndex}(k\_3, m) = m - 1hashIndex(k3​,m)=m−1
* hashIndex(k4,m)=m−3\text{hashIndex}(k\_4, m) = m - 3hashIndex(k4​,m)=m−3
* hashIndex(k5,m)=0\text{hashIndex}(k\_5, m) = 0hashIndex(k5​,m)=0
* hashIndex(k6,m)=m−3\text{hashIndex}(k\_6, m) = m - 3hashIndex(k6​,m)=m−3

The table using chaining would look like this:

| **Index** | **Linked List (Keys)** |
| --- | --- |
| 0 | k1k\_1k1​, k5k\_5k5​ |
| m−3m-3m−3 | k2k\_2k2​, k4k\_4k4​, k6k\_6k6​ |
| m−1m-1m−1 | k3k\_3k3​ |
| Other indices | (empty) |

In this case, multiple keys hash to the same index, so they are added to the linked list at that index.

**Runtime Analysis:**

1. **Insert Operation** insert(k,v)\text{insert}(k, v)insert(k,v):
   * **Cost**: To insert a new key-value pair:
     1. Calculate the hash index θ(1)\theta(1)θ(1).
     2. Traverse the linked list at that index θ(λ)\theta(\lambda)θ(λ), where λ\lambdaλ is the average length of the chain (i.e., the load factor).
     3. Insert the key-value pair at the end of the list θ(1)\theta(1)θ(1).
   * **Worst Case Runtime**: θ(n)\theta(n)θ(n), where all keys hash to the same index.
   * **Average Case Runtime**: θ(1+λ)\theta(1 + \lambda)θ(1+λ), assuming the hash function distributes keys uniformly across the table.
2. **Search Operation** search(k)\text{search}(k)search(k):
   * **Cost**: To find a key:
     1. Calculate the hash index θ(1)\theta(1)θ(1).
     2. Traverse the linked list at that index θ(λ)\theta(\lambda)θ(λ), where λ\lambdaλ is the average length of the chain.
   * **Worst Case Runtime**: θ(n)\theta(n)θ(n), where all keys hash to the same index.
   * **Average Case Runtime**: θ(1+λ)\theta(1 + \lambda)θ(1+λ).
3. **Delete Operation** delete(k)\text{delete}(k)delete(k):
   * **Cost**: To delete a key:
     1. Calculate the hash index θ(1)\theta(1)θ(1).
     2. Traverse the linked list θ(λ)\theta(\lambda)θ(λ) to find the key.
     3. Remove the key from the list θ(1)\theta(1)θ(1).
   * **Worst Case Runtime**: θ(n)\theta(n)θ(n), where all keys hash to the same index.
   * **Average Case Runtime**: θ(1+λ)\theta(1 + \lambda)θ(1+λ).

**Load Factor (λ) and Its Impact on Performance:**

The **load factor** λ=nm\lambda = \frac{n}{m}λ=mn​, where nnn is the number of items in the table and mmm is the number of slots (array size), directly impacts the performance of chaining.

* **When λ<1\lambda < 1λ<1**: On average, the chains are short, and operations (insert, search, delete) are efficient.
* **When λ>1\lambda > 1λ>1**: The chains become longer, slowing down operations as they require more time to traverse linked lists.

In practice, keeping λ\lambdaλ close to 1 ensures optimal performance, where the average length of the chain remains manageable, and operations can still be performed in constant time θ(1+λ)\theta(1 + \lambda)θ(1+λ).

**Worst-Case Runtime:**

* In the **worst case**, all keys hash to the same index, and the entire hash table is reduced to a single linked list. In this scenario:
  + **Insert**: θ(n)\theta(n)θ(n)
  + **Search**: θ(n)\theta(n)θ(n)
  + **Delete**: θ(n)\theta(n)θ(n)

While this is a theoretical possibility, in practice, a well-designed hash function minimizes the likelihood of such a scenario.

**Average-Case Runtime:**

The **average-case runtime** assumes the **Simple Uniform Hash Assumption (SUHA)**, which states that any key is equally likely to hash to any slot.

* For a well-distributed hash function, the **average-case runtime** is θ(1+λ)\theta(1 + \lambda)θ(1+λ), where λ=nm\lambda = \frac{n}{m}λ=mn​.
* With a low load factor, most linked lists will be short, keeping operations efficient.

**Advantages and Disadvantages of Chaining:**

**Advantages:**

* **No Limit on Table Size**: The hash table will not overflow since chains (linked lists) can grow as long as needed.
* **Easy to Implement**: The concept is simple and works well for moderate table sizes and load factors.
* **Handles High Load Factors**: Chaining can handle a higher load factor than open addressing.

**Disadvantages:**

* **Increased Memory Usage**: Each linked list node requires additional memory for storing pointers.
* **Potentially Slower Operations**: If the load factor becomes large, operations (especially search) may slow down due to long chains.

**Summary:**

Chaining is a straightforward and robust way to handle collisions in hash tables. By storing keys in linked lists at each index, it avoids table overflow and ensures that operations remain efficient for moderate load factors. However, performance degrades as the chains grow longer, which is why maintaining an optimal load factor is essential for efficiency.