**1. Dictionaries:**

A **dictionary** is a data structure that stores a collection of key-value pairs, where each key is unique and used to access its corresponding value. It is a very useful structure for mapping information efficiently, and in many programming languages, dictionaries are commonly referred to as **maps**, **hash maps**, or **associative arrays**.

* **Key**: The identifier used to access the value in the dictionary.
* **Value**: The data or information associated with the key.

**Properties of a Dictionary:**

* **Uniqueness**: Each key in a dictionary is unique, meaning no two entries can have the same key.
* **Fast Look-up**: Dictionary operations, like searching for a key, insertion of a new key-value pair, or deletion, are expected to be fast (often O(1) on average).
* **Dynamic Size**: The dictionary size can grow or shrink based on the number of elements added or removed.
* **Unordered**: In most implementations (e.g., hash-based), dictionaries don't maintain a specific order of key-value pairs.

**Applications of Dictionaries:**

* **Database Indexing**: Dictionaries are widely used for indexing in databases.
* **Caching**: Used in caching mechanisms to store frequently accessed data.
* **Symbol Tables**: Used in compilers to maintain variable names and values.
* **Counting Frequency**: Tracking frequencies of words, characters, etc.

**2. Dictionary Abstract Data Type (ADT):**

The **Dictionary Abstract Data Type (ADT)** is an abstraction for the collection of key-value pairs. It defines a set of operations that can be performed on the dictionary without specifying the internal implementation details. It provides a general interface for manipulating key-value pairs.

**Key Operations:**

1. **Insert(key, value):**
   * Adds a new key-value pair to the dictionary.
   * If the key already exists, it updates the value associated with the key.
   * **Time Complexity**: Depends on the underlying implementation (usually O(1) for hash tables).
2. **Search(key):**
   * Retrieves the value associated with the specified key.
   * If the key is not present, it returns an indication of absence (e.g., None).
   * **Time Complexity**: Usually O(1) for hash tables.
3. **Delete(key):**
   * Removes the key-value pair associated with the specified key from the dictionary.
   * **Time Complexity**: O(1) in hash tables (on average).
4. **Update(key, value):**
   * Changes the value associated with a specific key.
   * If the key is not present, some implementations may insert it as a new key-value pair.
   * **Time Complexity**: O(1).
5. **Size():**
   * Returns the number of key-value pairs currently in the dictionary.
   * **Time Complexity**: O(1).
6. **IsEmpty():**
   * Checks if the dictionary is empty.
   * **Time Complexity**: O(1).
7. **Keys():**
   * Returns a collection of all keys in the dictionary.
   * **Time Complexity**: O(n), where n is the number of key-value pairs.
8. **Values():**
   * Returns a collection of all values in the dictionary.
   * **Time Complexity**: O(n).
9. **Contains(key):**
   * Checks if a specific key is present in the dictionary.
   * **Time Complexity**: O(1) for hash-based implementations.

**Dictionary ADT Operations Complexity (Typical Implementations):**

| **Operation** | **Time Complexity (Average)** | **Time Complexity (Worst)** |
| --- | --- | --- |
| **Insert(key, value)** | O(1) | O(n) (if resizing is needed) |
| **Search(key)** | O(1) | O(n) (in case of collisions) |
| **Delete(key)** | O(1) | O(n) (due to collisions) |
| **Update(key, value)** | O(1) | O(n) (if resizing or collision happens) |

**Important Considerations:**

* **Collision Handling**: In case two keys hash to the same index, the dictionary must have a mechanism for resolving this collision (e.g., chaining or open addressing).
* **Load Factor**: The load factor indicates how full the hash table is. If it exceeds a certain threshold, the dictionary needs resizing.
* **Efficiency**: Hash-based dictionaries are typically very efficient with average-case time complexities of O(1) for insert, delete, and search.

**3. Implementation of Dictionaries:**

**a. Using Arrays or Lists:**

In a simple array-based implementation, the dictionary stores key-value pairs as elements in the array.

* **Inserting Key-Value Pairs**: Traverse the array to find the correct position and insert the new pair.
* **Searching for a Key**: Perform a linear search through the array to find the key.
* **Deleting a Key-Value Pair**: Locate the key through linear search and remove the corresponding entry.

**Issues**:

* **Inefficiency**: Both searching and deleting take O(n) time.
* **Limited Performance**: Not suitable for large datasets.

**b. Using Linked Lists:**

A **linked list** implementation involves creating a linked list of nodes, where each node stores a key-value pair and a reference to the next node.

* **Insertion**: Add new key-value pairs by creating a new node.
* **Search**: Traverse the linked list and compare keys.
* **Deletion**: Find and remove the corresponding node.

**Issues**:

* **Still inefficient**: Searching and deleting require O(n) time in the worst case.

**c. Using Hash Tables:**

A **hash table** is a more efficient way to implement a dictionary, where a hash function converts the key into an index, which points to the corresponding value.

**Working of Hash Tables:**

1. **Hash Function**: The hash function is responsible for converting a key into an index. The idea is to distribute the keys uniformly across the array.
2. **Collision Resolution**: When two keys hash to the same index, a **collision** occurs. There are several strategies to handle collisions:
   * **Chaining**: Maintain a linked list at each index to store multiple key-value pairs.
   * **Open Addressing**: Find the next available slot for the key if a collision occurs (e.g., linear probing or quadratic probing).

**Key Operations:**

* **Insert(key, value)**:
  + Compute the hash of the key.
  + Insert the key-value pair at the index corresponding to the hash value.
  + Handle collisions if any.
* **Search(key)**:
  + Compute the hash of the key.
  + Retrieve the value associated with the key from the corresponding index.
* **Delete(key)**:
  + Compute the hash of the key.
  + Remove the key-value pair from the corresponding index.

**Advantages of Hash Tables**:

* **Efficiency**: Average time complexity for insert, delete, and search operations is O(1).
* **Scalability**: Hash tables can efficiently handle large numbers of key-value pairs.

**Disadvantages of Hash Tables**:

* **Collisions**: If too many keys hash to the same index, performance degrades.
* **Resizing**: Hash tables need resizing when the load factor becomes too high.
* **Space Complexity**: Hash tables can waste memory, especially if they are not resized properly.

**Example of Hash Table Implementation in Python:**

In Python, the built-in dict type is a hash table implementation.

# Creating a dictionary

my\_dict = {}

# Inserting key-value pairs

my\_dict["name"] = "Alice"

my\_dict["age"] = 25

# Searching for a key-value pair

if "name" in my\_dict:

print(my\_dict["name"]) # Output: Alice

# Deleting a key-value pair

del my\_dict["age"]

# Checking the dictionary

print(my\_dict)

**Key Functions of Python Dictionary**:

* my\_dict.get(key): Retrieves the value for the given key, returning None if the key is not found.
* my\_dict.items(): Returns a list of all key-value pairs in the dictionary.
* my\_dict.keys(): Returns all the keys in the dictionary.
* my\_dict.values(): Returns all the values in the dictionary.

**d. Complexity of Hash Tables:**

* **Average-case Complexity**:
  + Insert: O(1)
  + Search: O(1)
  + Delete: O(1)
* **Worst-case Complexity** (with collisions):
  + Insert: O(n) (due to resizing or long chains of collisions)
  + Search: O(n) (if all elements hash to the same index)
  + Delete: O(n)

**Hashing**

**Hashing** is a technique for efficiently mapping keys to a fixed-size array or table using a **hash function**. Hashing allows for fast data retrieval and storage operations, such as insertion, deletion, and searching. It is widely used in data structures like **hash tables** and **hash maps**, where data is stored in key-value pairs.

* **Hash Table**: A hash table is a data structure that stores data in an array format, where each element is stored at an index based on its hash value.
* **Hash Map**: A dictionary-like structure in programming languages, which uses hashing for fast lookups and is essentially a **hash table** that stores key-value pairs.

**2. Hash Function**

A **hash function** is a mathematical function that takes an input (the key) and computes an integer that represents the position or index in the hash table where the value associated with that key should be stored. The output is called a **hash value** or **hash code**.

The main goal of a good hash function is to distribute keys uniformly across the available slots in the hash table. This minimizes collisions, where two keys produce the same hash value and are assigned to the same position in the hash table.

**Properties of a Good Hash Function:**

1. **Deterministic**: For the same input key, the hash function should always return the same hash value.
2. **Uniform Distribution**: A good hash function distributes keys uniformly across the available range of indices to reduce the chance of collisions.
3. **Efficiency**: The hash function should be computationally efficient.
4. **Minimize Collisions**: A good hash function reduces the likelihood of two keys hashing to the same value (a collision).

**Common Hash Functions:**

1. **Division Method**:

**1. Review of Hashing**

**Hashing** is a technique for efficiently mapping keys to a fixed-size array or table using a **hash function**. Hashing allows for fast data retrieval and storage operations, such as insertion, deletion, and searching. It is widely used in data structures like **hash tables** and **hash maps**, where data is stored in key-value pairs.

* **Hash Table**: A hash table is a data structure that stores data in an array format, where each element is stored at an index based on its hash value.
* **Hash Map**: A dictionary-like structure in programming languages, which uses hashing for fast lookups and is essentially a **hash table** that stores key-value pairs.

**2. Hash Function**

A **hash function** is a mathematical function that takes an input (the key) and computes an integer that represents the position or index in the hash table where the value associated with that key should be stored. The output is called a **hash value** or **hash code**.

The main goal of a good hash function is to distribute keys uniformly across the available slots in the hash table. This minimizes collisions, where two keys produce the same hash value and are assigned to the same position in the hash table.

**Properties of a Good Hash Function:**

1. **Deterministic**: For the same input key, the hash function should always return the same hash value.
2. **Uniform Distribution**: A good hash function distributes keys uniformly across the available range of indices to reduce the chance of collisions.
3. **Efficiency**: The hash function should be computationally efficient.
4. **Minimize Collisions**: A good hash function reduces the likelihood of two keys hashing to the same value (a collision).

**Common Hash Functions:**

1. **Division Method**:

hash(k) = k % table\_size

Where k is the key, and table\_size is the size of the hash table.

1. **Multiplication Method**:

python

Copy code

hash(k) = floor(table\_size \* (k \* A % 1))

Where A is an irrational constant (usually 0.6180339887), and the fractional part of k \* A is used to determine the hash.

1. **Cryptographic Hash Functions** (e.g., SHA-256, MD5): These functions are more complex and are used for security, but are slower than simpler hash functions.

**3. Collision Resolution Techniques**

When two different keys hash to the same index, a **collision** occurs. There are several techniques to handle collisions:

**a. Chaining (Separate Chaining)**

Separate Chaining is a collision resolution technique used in hash tables. In this method, each index of the hash table points to a collection (usually a linked list or another data structure) of all the key-value pairs that hash to the same index. This way, even if multiple keys hash to the same index, they can be stored separately in the linked list or another collection at that index.

**How Separate Chaining Works:**

1. Hash Function: A hash function is used to compute an index from the key. This index is where the key-value pair will be stored in the hash table.
2. Collision Handling: When two or more keys hash to the same index (collision), each key is stored in a linked list (or another collection) at that index.
3. Insertion: When a key is inserted, it is hashed to find an index. If there is already a key at that index, the new key-value pair is added to the linked list at that index.
4. Search: To search for a key, the hash function is applied to compute the index, and then the list at that index is traversed to find the desired key.
5. Deletion: To delete a key, the hash function is used to compute the index, and then the linked list at that index is searched to find and remove the key-value pair.

**Advantages of Separate Chaining:**

1. Handling Collisions: Separate chaining allows handling multiple keys that hash to the same index without needing to modify the overall structure of the hash table.
2. Dynamic Growth: As the number of keys grows, the linked lists at each index grow. This means that the hash table can handle more data without a significant performance hit as long as the hash function distributes keys well.
3. Simpler Rehashing: In separate chaining, the hash table does not need to be resized immediately when a collision occurs; the linked list grows dynamically as needed.

**Disadvantages of Separate Chaining:**

1. Memory Overhead: Using linked lists or other collections requires additional memory to store the linked nodes or elements.
2. Performance Degradation: As the load factor increases (more collisions occur), the linked lists grow longer, and the performance of search, insert, and delete operations may degrade.
3. Inefficient Space Utilization: If many slots in the table are empty and only a few have linked lists, there may be inefficient space usage in the hash table.

**Python code**

class Node:

# A node for the linked list to store the key-value pairs

def \_\_init\_\_(self, key, value):

self.key = key

self.value = value

self.next = None

class HashTable:

def \_\_init\_\_(self, size=10):

# Initialize the hash table with a given size

self.size = size

self.table = [None] \* self.size

def hash(self, key):

# Simple hash function (modulo operation)

return hash(key) % self.size

def insert(self, key, value):

# Insert a key-value pair into the hash table

index = self.hash(key)

new\_node = Node(key, value)

# If no collision, simply add the node at the index

if self.table[index] is None:

self.table[index] = new\_node

else:

# Collision: add the node to the linked list at this index

current\_node = self.table[index]

while current\_node:

if current\_node.key == key:

# Update the value if the key already exists

current\_node.value = value

return

if current\_node.next is None:

break

current\_node = current\_node.next

current\_node.next = new\_node

def search(self, key):

# Search for a key in the hash table

index = self.hash(key)

current\_node = self.table[index]

while current\_node:

if current\_node.key == key:

return current\_node.value

current\_node = current\_node.next

return None # Key not found

def delete(self, key):

# Delete a key-value pair by key

index = self.hash(key)

current\_node = self.table[index]

prev\_node = None

while current\_node:

if current\_node.key == key:

if prev\_node:

prev\_node.next = current\_node.next

else:

self.table[index] = current\_node.next

return

prev\_node = current\_node

current\_node = current\_node.next

raise KeyError(f"Key '{key}' not found.")

def \_\_str\_\_(self):

# String representation of the hash table

result = []

for i in range(self.size):

if self.table[i]:

current\_node = self.table[i]

while current\_node:

result.append(f"({current\_node.key}: {current\_node.value})")

current\_node = current\_node.next

return " -> ".join(result)

# Testing the HashTable with Separate Chaining

hash\_table = HashTable()

# Inserting elements

hash\_table.insert("name", "Alice")

hash\_table.insert("age", 25)

hash\_table.insert("city", "New York")

hash\_table.insert("email", "alice@example.com")

# Searching for values

print(f"Search 'name': {hash\_table.search('name')}")

print(f"Search 'age': {hash\_table.search('age')}")

print(f"Search 'country': {hash\_table.search('country')}") # Not found

# Deleting a key

hash\_table.delete("age")

print(f"Hash Table after deleting 'age': {hash\_table}")

# Inserting more elements

hash\_table.insert("phone", "123-456-7890")

print(f"Hash Table after more insertions: {hash\_table}")

**b. Open Addressing (Closed Hashing)**

1. **Open Addressing** is a method of collision resolution in a hash table where, instead of using separate data structures (like linked lists in separate chaining), the hash table itself is used to store all the keys. When a collision occurs (i.e., two keys hash to the same index), open addressing resolves the collision by probing for other available slots within the table.
2. In open addressing, all elements are stored directly in the hash table, and if a slot is already occupied, the algorithm searches for the next available slot using a predefined sequence of probe steps.

**How Open Addressing Works:**

1. **Hash Function**: A hash function computes the index for each key. The index is the primary position where the key-value pair will be inserted into the table.
2. **Collision Handling**: When a key hashes to an index that is already occupied by another key, open addressing uses **probing** to find the next available slot. The probing can follow several strategies.
3. **Insertion**: If the computed index is empty, the key is inserted. If the computed index is occupied, the algorithm looks for the next available slot based on a probing strategy.
4. **Search**: To search for a key, the hash table computes the index using the hash function and follows the probing sequence to find the key.
5. **Deletion**: To delete a key, the hash table computes the index using the hash function and follows the probing sequence to locate and remove the key.

**Probing Strategies in Open Addressing:**

There are several types of probing strategies to resolve collisions in open addressing:

1. **Linear Probing:-**

**Linear Probing** is a collision resolution technique used in **open addressing** within hash tables. When a collision occurs (i.e., two keys hash to the same index), **linear probing** searches for the next available slot by checking subsequent slots in the hash table in a sequential manner. If the slot is occupied, it continues probing in a linear fashion until an empty slot is found.

**How Linear Probing Works:**

* **Hash Function**: A hash function is used to compute an index for a key. If two or more keys hash to the same index, a collision occurs.
* **Collision Handling**: In the case of a collision, linear probing attempts to resolve the conflict by searching linearly for the next available empty slot in the table.
* **Probing Process**: When a collision occurs at index i, linear probing checks the next slot, i.e., index (i+1) % table\_size, then the next index (i+2) % table\_size, and so on, wrapping around to the beginning of the table if necessary, until an empty slot is found.
* **Insertion**: If an empty slot is found during probing, the key-value pair is inserted there.
* **Search**: To search for a key, the hash function is applied to compute the initial index. If the key at that index does not match, linear probing is used to check subsequent slots until the key is found or the table is searched completely.
* **Deletion**: When deleting a key, the algorithm must ensure that the deletion doesn't break the chain of probed elements that are part of the same bucket. After deleting a key, the subsequent elements in the probe chain may need to be reinserted to avoid losing them.

**Linear Probing Formula:**

The formula for computing the index in linear probing is:

**hash(k, i) = (hash(k) + i) % table\_size**

Where:

* k is the key being hashed,
* i is the number of probes (i.e., how many positions we've moved),
* table\_size is the total number of slots in the table.

**Advantages of Linear Probing:**

1. **Simplicity**: Linear probing is easy to implement and understand.
2. **Cache Efficiency**: Since the search is sequential, it can take advantage of cache locality.
3. **Space Efficiency**: Unlike separate chaining, linear probing does not require additional data structures (like linked lists).

**Disadvantages of Linear Probing:**

1. **Primary Clustering**: As elements fill the table, sequences of consecutive occupied slots (clusters) can form, leading to inefficient probing as the distance between consecutive empty slots increases.
2. **Performance Degradation**: As the load factor increases (i.e., the table becomes more full), the number of probes required to find an empty slot increases, leading to slower performance.
3. **Table Resizing**: To maintain efficient operations, the table may need to be resized (rehashing) when the load factor exceeds a threshold, which can be costly.
4. **Linear Probing**:
   * If a collision occurs at index i, the algorithm checks the next slot, i.e., index i+1, i+2, etc., until an empty slot is found.
   * Formula:

hash(k, i) = (hash(k) + i) % table\_size

* + Where i is the number of attempts or collisions.

1. **Quadratic Probing**:
   * Instead of checking consecutive slots, quadratic probing uses a quadratic function to find the next slot, which reduces clustering.
   * Formula:

hash(k, i) = (hash(k) + i^2) % table\_size

1. **Double Hashing**:
   * A second hash function is used to determine the step size when a collision occurs.
   * Formula:

hash(k, i) = (hash1(k) + i \* hash2(k)) % table\_size

**c. Rehashing (Dynamic Resizing)**

When the load factor (the number of elements divided by the table size) exceeds a certain threshold (usually 0.7), the hash table is **rehashes**. This involves creating a larger table and rehashing all existing elements into the new table.

**d. Perfect Hashing**

**Perfect Hashing** is a collision-free approach where a hash function is constructed so that no collisions occur. This is typically used when the set of keys is static, and a perfect hash function can be created in advance. It's an ideal but more complex solution.

**Python Code: Hash Table with Open Addressing (Linear Probing)**

The following Python code implements a hash table using **open addressing** and **linear probing** for collision resolution:

class HashTable:

def \_\_init\_\_(self, size=10):

# Initialize the hash table with a given size

self.size = size

self.table = [None] \* self.size

self.num\_elements = 0

self.load\_factor\_threshold = 0.7 # Load factor threshold for rehashing

def hash(self, key):

# Hash function: simple modulo operation

return hash(key) % self.size

def load\_factor(self):

# Calculate load factor

return self.num\_elements / self.size

def rehash(self):

# Rehashing procedure when load factor exceeds threshold

old\_table = self.table

self.size \*= 2 # Double the size

self.table = [None] \* self.size

self.num\_elements = 0 # Reset the number of elements

for item in old\_table:

if item is not None:

key, value = item

self.insert(key, value)

def insert(self, key, value):

# Insert a key-value pair into the hash table

if self.load\_factor() > self.load\_factor\_threshold:

self.rehash()

index = self.hash(key)

start\_index = index

# Linear probing for collision resolution

while self.table[index] is not None:

if self.table[index][0] == key: # Update value if key exists

self.table[index] = (key, value)

return

index = (index + 1) % self.size # Check next slot

if index == start\_index: # Full circle, table is full

raise Exception("Hash Table is full")

# Insert the key-value pair

self.table[index] = (key, value)

self.num\_elements += 1

def search(self, key):

# Search for a key in the hash table

index = self.hash(key)

start\_index = index

while self.table[index] is not None:

if self.table[index][0] == key:

return self.table[index][1] # Return the value

index = (index + 1) % self.size # Check next slot

if index == start\_index: # We circled back, key not found

break

return None

def delete(self, key):

# Delete a key-value pair by key

index = self.hash(key)

start\_index = index

while self.table[index] is not None:

if self.table[index][0] == key:

self.table[index] = None # Remove the key-value pair

self.num\_elements -= 1

return

index = (index + 1) % self.size # Check next slot

if index == start\_index: # We circled back, key not found

break

raise KeyError(f"Key '{key}' not found.")

def \_\_str\_\_(self):

# String representation of the hash table

return str([item for item in self.table if item is not None])

# Testing the HashTable class

hash\_table = HashTable()

# Inserting elements

hash\_table.insert("name", "Alice")

hash\_table.insert("age", 25)

hash\_table.insert("city", "New York")

# Searching for values

print(f"Search 'name': {hash\_table.search('name')}")

print(f"Search 'age': {hash\_table.search('age')}")

print(f"Search 'country': {hash\_table.search('country')}") # Not found

# Deleting a key

hash\_table.delete("age")

# Checking the table after deletion

print(f"Hash Table: {hash\_table}")

# Inserting more elements to check rehashing

hash\_table.insert("email", "alice@example.com")

hash\_table.insert("phone", "123-456-7890")

print(f"Hash Table after more insertions: {hash\_table}")

**Explanation of the Code:**

1. **\_\_init\_\_**: Initializes the hash table with a given size (default is 10), and an empty table of that size. A threshold for rehashing is also set to 0.7.
2. **hash**: A simple hash function using Python’s built-in hash() function, followed by the modulo operation to ensure the hash fits within the table's size.
3. **insert**: Adds a key-value pair to the table, using **linear probing** to handle collisions. If the load factor exceeds the threshold, the table is **rehashes** to a larger size.
4. **search**: Searches for a key and returns the associated value if found.
5. **delete**: Deletes a key-value pair from the hash table.
6. **rehash**: Doubles the table's size and rehashes all existing key-value pairs when the load factor exceeds the threshold.

**Output Example:**

Search 'name': Alice

Search 'age': 25

Search 'country': None

Hash Table: [('name', 'Alice'), ('city', 'New York')]

Hash Table after more insertions: [('name', 'Alice'), ('city', 'New York'), ('email', 'alice@example.com'), ('phone', '123-456-7890')]