Data Structures and Algorithms Course Using Python

HashMaps: [**Hash table - Wikipedia**](https://en.wikipedia.org/wiki/Hash_table)

HashMaps Visualization: [**Open Hashing Visualization (usfca.edu)**](https://www.cs.usfca.edu/~galles/visualization/OpenHash.html)

Hash tables, also known as hash maps, are a type of data structure that offers very efficient data retrieval. They work by mapping keys to values, allowing for quick access to values when you know the corresponding key. Here's an overview of how they work:

**1. Hash Function**

* **Purpose**: Converts a key into an integer index.
* **How It Works**: The hash function takes a key as input and computes an integer. Ideally, this function should distribute keys uniformly across the available space to minimize collisions (where two keys hash to the same index).

**2. Array of Buckets or Slots**

* **Storage**: The hash table uses an array where each element is a "bucket" or "slot" where data can be stored.
* **Accessing Data**: The index computed by the hash function determines which bucket a key-value pair will be stored in.

**3. Handling Collisions**

* **Issue**: Two different keys might produce the same index from the hash function.
* **Solutions**:
  + **Chaining**: Each bucket holds a list of entries that hashed to that bucket. If a collision occurs, the new item is added to the list.
  + **Open Addressing**: If a collision occurs, the hash table searches for the next empty slot using a process like **linear probing, quadratic probing, or double hashing**.

**4. Insertion, Lookup, and Deletion**

* **Insertion**: Apply the hash function to the key to find the index, then store the key-value pair in the corresponding bucket.
* **Lookup**: Apply the hash function to the key to find the index, then search that bucket (and possibly others, in case of collisions) to find the desired entry.
* **Deletion**: Similar to lookup, but instead of retrieving the value, the entry is removed.

**5. Resizing**

* **Why Necessary**: As more elements are added, the hash table might become too crowded, leading to many collisions.
* **Process**: The hash table is resized (usually doubled), and existing entries are rehashed (re-inserted using the hash function) into the new, larger array of buckets.

**Efficiency**

* **Best Case**: O(1) for insertion, lookup, and deletion when there are no or few collisions.
* **Worst Case**: O(n) in scenarios where many items hash to the same bucket, especially with poor hash functions or a small number of buckets.

**Applications**

Hash tables are widely used in various applications like database indexing, caching, symbol tables in compilers, and associative arrays in programming languages.

Their efficiency and flexibility make them an essential tool in a software developer's arsenal.

Idempotency and Collisions

The idempotency of a hash function and the occurrence of collisions in a hash table are related to different aspects of how hash tables work.

1. **Idempotency of a Hash Function**:
   * **Meaning**: A hash function is idempotent when it consistently returns the same hash value for the same input. In other words, every time you pass the same key to the hash function, it will produce the same hash value.
   * **Purpose**: This is crucial for the functionality of a hash table. If a hash function produced different values for the same input on different occasions, it would be impossible to reliably retrieve values from the hash table.
2. **Collisions in Hash Tables**:
   * **Cause**: Collisions occur when two different keys produce the same hash value, or when they are mapped to the same bucket in the array. This is an inherent limitation of any hash function due to the Pigeonhole Principle: if you have more keys than hash values (which is often the case), some keys must end up with the same hash value.
   * **Example**: Imagine a simple hash function for strings where the function sums the ASCII values of the characters and then takes the remainder when divided by 100. Different strings like "abc" and "acb" will produce the same sum, and hence, the same hash value.
3. **Implications**:
   * **Hash Table Design**: A good hash function aims to minimize collisions by uniformly distributing keys across the available hash values. However, no hash function can entirely prevent collisions, especially as the number of keys grows large relative to the size of the array.
   * **Collision Resolution**: Hash tables must implement a method to handle collisions, such as chaining (storing all entries with the same hash value in a list) or open addressing (finding another spot in the table for the new key).

In summary, the idempotency of a hash function ensures consistent mapping of a key to a hash value, which is essential for retrieving data from a hash table. Collisions, on the other hand, are an unavoidable occurrence when different keys map to the same hash value or bucket, and they must be accounted for in the design of the hash table.

HashTables vs Arrays

Hash tables offer several advantages over arrays, particularly in scenarios involving large datasets and when the primary operations include searching, insertion, and deletion of data. Here are some of the key advantages:

**1. Time Complexity for Common Operations**

* **Search, Insert, Delete**:
  + In hash tables, these operations can often be performed in *O*(1) average time complexity, assuming a good hash function with minimal collisions.
  + In contrast, for unsorted arrays, search operations take *O*(*n*) time, insertions at specific positions are *O*(*n*), and deletion requires *O*(*n*) time (since elements need to be shifted).
* **Sorted Arrays**:
  + While binary search in sorted arrays allows for *O*(log*n*) search time, insertions and deletions remain *O*(*n*) due to the need to maintain order.

**2. Direct Access via Keys**

* **Hash Tables**: Allow direct access to data through keys. You don't need to know an element's position; the hash function maps the key directly to the location where the data is stored.
* **Arrays**: Access is index-based. To retrieve an element, you need to know its exact index, and for searching, you often have to iterate through elements sequentially.

**3. Flexibility in Key Types**

* **Hash Tables**: Can handle a wide variety of data types as keys, including integers, strings, objects, etc.
* **Arrays**: Typically, indexes are integer-based.

**4. Handling Large Data Sets Efficiently**

* **Hash Tables**: Particularly effective in scenarios where there are large numbers of search, insert, and delete operations, as they maintain efficiency even with large datasets.
* **Arrays**: Efficiency degrades, especially with operations that require traversing or shifting many elements.

**5. No Requirement for Order**

* **Hash Tables**: Do not maintain any order of elements, which can be an advantage when order is not necessary, as maintaining order can be an overhead.
* **Arrays**: Maintaining a sorted order in arrays involves additional computational effort.

**6. Dynamic Resizing**

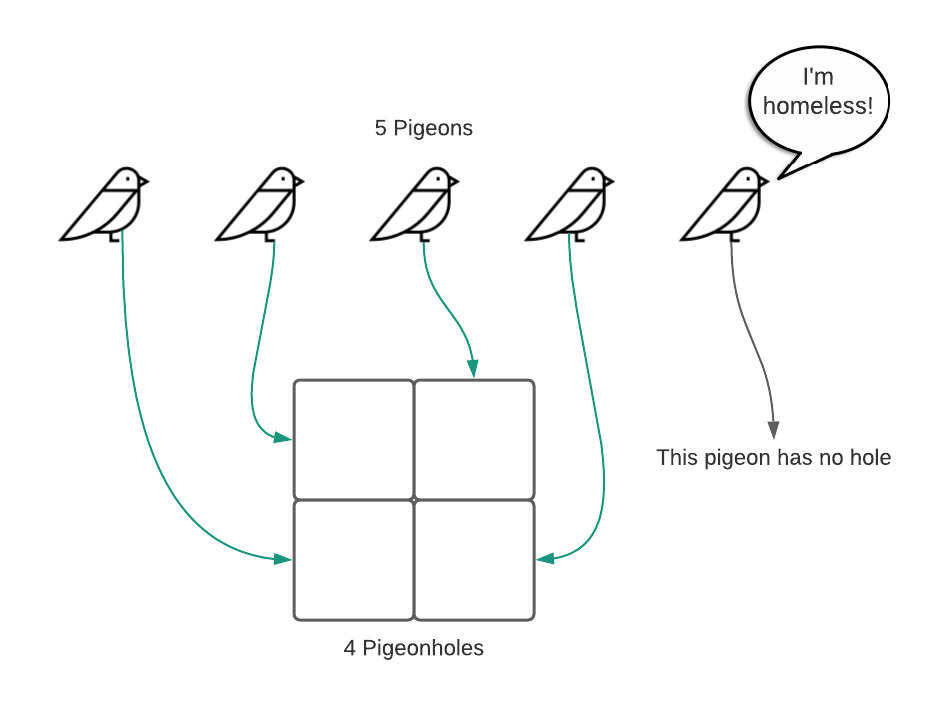
* **Hash Tables**: Many implementations dynamically resize to accommodate more elements, thus handling growing data efficiently.
* **Arrays**: Static arrays have a fixed size. While dynamic arrays (like vectors in C++ or ArrayLists in Java) can resize, the resizing process (which involves creating a new array and copying elements) can be expensive.

**Limitations and Considerations**

* **Memory Usage**: Hash tables generally use more memory than arrays due to the storage of keys, values, and additional structures for collision handling.
* **Hash Function Quality**: The efficiency of a hash table heavily relies on the quality of its hash function. Poor hash functions can lead to many collisions, degrading performance.
* **Order and Predictability**: Arrays provide ordered and predictable traversal, which is not the case with hash tables.

In summary, hash tables are advantageous when you need fast access and operations on data using keys, especially in large datasets where operations like search, insert, and delete are frequent. Arrays, while simpler and sometimes more memory-efficient, are generally slower for these operations, particularly when dealing with unsorted data or maintaining order is not a priority

Pigeonhole Principle: item size > container capacity

****

Collisions in hash functions are inevitable due to a fundamental principle known as the **Pigeonhole Principle**. This principle states that if you have more items than containers to put them in, at least one container will end up with more than one item. In the context of hash tables, this means if you have more keys than available hash values (which is often the case), some keys must inevitably hash to the same value.

**Reasons for Inevitability of Collisions**

1. **Limited Range of Hash Values**: Hash functions map a potentially infinite or very large set of keys to a finite set of hash values (or indices). The finite range of the hash table means there are fewer potential hash values than possible keys.
2. **Non-Uniform Key Distribution**: Even with a good hash function, it's challenging to perfectly distribute an arbitrary set of keys uniformly across the hash value space. Some clustering is likely, leading to collisions.
3. **Variability of Key Space**: Keys can be of various types and sizes (e.g., strings, integers, objects), making it virtually impossible to create a hash function that perfectly maps these keys to a unique index without collision.

**Managing Collisions**

To handle collisions in hash tables, several strategies are employed:

1. **Chaining (Separate Chaining)**
   * **How It Works**: Each bucket of the hash table points to a linked list of entries that hashed to the same index. If two keys hash to the same index, they are stored in a linked list at that bucket.
   * **Pros and Cons**: Simple to implement; however, it requires additional memory for pointers in the list, and lookup times can suffer if the list becomes long.
2. **Open Addressing**
   * **Types**: Includes methods like linear probing, quadratic probing, and double hashing.
   * **How It Works**: When a collision occurs, open addressing seeks the next available slot using a predefined probe sequence.
   * **Pros and Cons**: It avoids pointers, saving memory. However, it can suffer from clustering issues, where continuous blocks of slots get filled, leading to longer search times.
3. **Double Hashing**
   * **How It Works**: Uses a second hash function to determine the probe sequence after a collision.
   * **Pros and Cons**: Offers better performance by reducing clustering compared to linear or quadratic probing but is more complex to implement.
4. **Resizing the Hash Table**
   * **Trigger**: When the load factor (the ratio of the number of stored entries to the number of buckets) exceeds a certain threshold.
   * **Process**: The hash table is resized (typically doubled), and existing entries are rehashed into the new, larger table.
   * **Purpose**: Resizing helps maintain a balance between the efficient use of space and time.
5. **Using a Good Hash Function**
   * **Characteristics**: A good hash function is essential for minimizing collisions. It should distribute keys uniformly across the hash table and should be fast to compute.
   * **Customization**: For specific types of keys, designing a hash function that accounts for the key characteristics can help in reducing collisions.

In summary, collisions in hash functions are unavoidable due to the finite and often smaller range of hash values compared to the potentially infinite set of keys. Effective collision resolution strategies like chaining, open addressing, and resizing the hash table, along with a well-designed hash function, are crucial for maintaining the efficiency of hash tables.