Hash Table

Mai Dahshan

September 23, 2024

Optimal Lookup

- Imagine you have an arbitrary collection of numbers and want to store them in an data structure for efficient search operations.

  Two Searching Algorithms: Linear Search (O(n)) & Binary Search (O(logn))
  What if we need to find a better way for searching elements that has a better time complexity in the best and worst cases compared to what we already know.

  Which data structure and searching algorithm would you use?

  We can do this using hash tables

"Magical" Data Structure

- What if we could locate an item in a list without needing to search?

- We enter our search key, and the algorithm directs us straight to the item we're seeking.

- No need to search through all the items (linear search) or even half of them(binary search)

# Learning Objectives

Hashing! woop, woop

- Understand the concept of hashing and how a hash table stores and retrieves data
- Learn how to implement basic operations in a hash table, including insertion, deletion, and searching
- Understand techniques for handling collisions in hash tables
- Analyze the time complexity of hash table operations in best, average, and worst cases
- Apply hash tables in solving real-world computational problems

🏛 UVA DATA SCIENCE

Direct Access Table

- A direct address table is a data structure that maps keys to indices in an array.

- Direct-address table representation:
  - An array T[0 . . . m - 1]
  - Each **slot**, or position, in T corresponds to a key in U
    - No two elements have the same key
  - For an element x with key k, a pointer to x will be placed in location T[k]
  - If there are no elements with key k in the set, T[k] is empty, represented by NULL
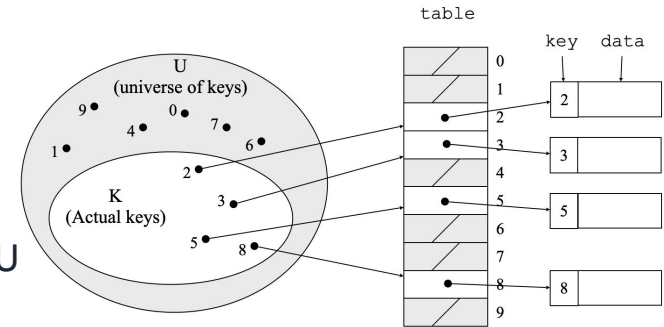
Each key is drawn from a universe
U = {0, 1, . . . , m - 1}

UVA DATA SCIENCE

5

Direct Access Table

- O(1) time complexity for insertions, deletions, and lookups

- This structure is particularly useful when the set of possible keys is small and known in advance.

Use Case of Direct Access Table

- Storing student IDs and their corresponding grades

- Manage product inventories in a warehouse using product IDs

- Store phone numbers indexed by area codes or specific digits

- Manage active user sessions in a web application using session IDs

UVA DATA SCIENCE

Issues with Direct Access Table

- Space Inefficiency: Requires a large array if the range of keys is vast but sparsely populated

- Fixed Universe of Keys: Not suitable for dynamic key sets or when keys are not integers

- Collision Handling: Does not inherently handle collisions; needs additional strategies if multiple items share a key

if two elements are being stored at the same index, AKA we can'd handle multiple elements with the same key. They have to be unique

Hash Table

- How can we store N keys in a table of size M, where no correspondence between keys and indices of table and values of keys are greater than M?
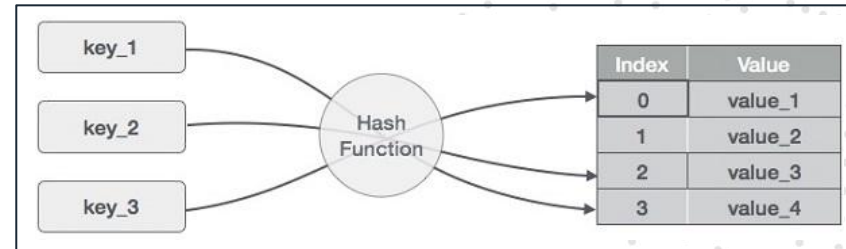
Remember, there is a 1:1 access between key and index (of the array)

Hash Table

- A hash table is a data structure that maps keys to values for efficient data retrieval

- A Hash Table consists of three things:

  - A hash function

  - A data storage structure

  - A method of addressing hash collisions

    WIth a good method, you could actually incnlude non-unique keys to translate it into a new index in the table
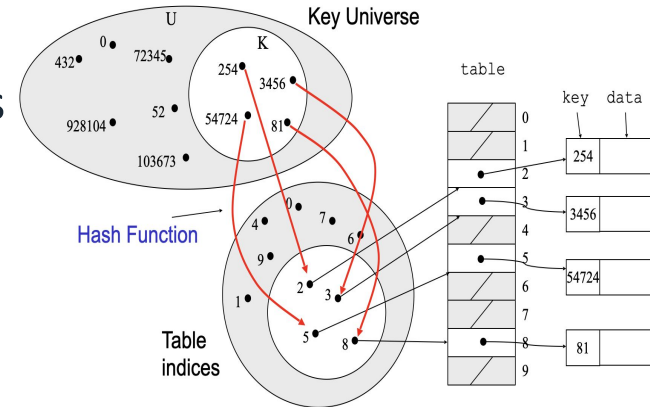
Hash Table

Index	Value
0	value_1
1	value_2
2	value_3
3	value_4

key_1
key_2
key_3

Hash Function

Hash Function

- A hash function h transforms a key into an index in a hash table T[0…m-1]: $h : U \rightarrow \{0, 1, . . . , m - 1\}$

- The values returned by a hash function are called hash values, hash codes, hash sums, or hash integers

- Implementation of a hash function is two steps
  - Creating the hash code for the hash key
  - Transferring hash code to an index

In some cases, this will happen, but if there's a collision, the hash code will be different to the index

🏛 UVA DATA SCIENCE

Hash Function

- **Ideal case:** Every key maps to a **unique** location in the hash table

- Typically not possible because:

  - All of the keys are not known in advance

    - flight numbers mapping to actual flights

  - Only a small percentage of the possible key combinations are used.

    - A company with 500 employees would not create a hash table mapped to 1 billion combinations of SIN numbers

Hash Function

your hash funtion should genereate different hash codes for different keys!

- There is a very large number of available hash functions. A function may be suitable for one type of object and not the other

- Characteristics of a good hash function

  - Efficient: Computing the hash code should be fast and efficient

  - Deterministic: The same input will always produce the same output
    The hash function should generate the same hash code for the same key, always

  - Uniform Distribution: A good hash function minimizes collisions by distributing keys uniformly across the hash table

  - In Python, hash functions can work with various data types (strings, integers, tuples, etc.).

Hash Function

- Examples of hash functions include:
  - Simple modulo arithmetic   Key mod size of table
  - Mid-Square Method
  - Binning   If you have a string val, how will you use modulo arith.?
   You take the aski representation of the letters.
  - and more   ex. 'abc' mod 10 (size of table)
                                (97 + 98 + 99) mod 10 = your index key
  - ..

Hash Function

- Simple modulo arithmetic

  - The hash function uses the modulus operation to map keys to indices within a fixed-size hash table.

  - The formula is typically: $\text{hash}(key) = key \mod table\_size$

  - This ensures that the result is always within the bounds of the array indices (0 to table_size−1)

Hash Function

- Example Simple modulo arithmetic

  - Integer key
    - Key: 42
    - Table Size: 10
    - Hash code hash(42)=42 mod 10=2

  - String Key
    - Key: "abc"
    - Table Size: 7
    - Hash code
      - Sum the ASCII values: 97+98+99=294
      - 294 mod 7 = 0

Hash Table
0
9
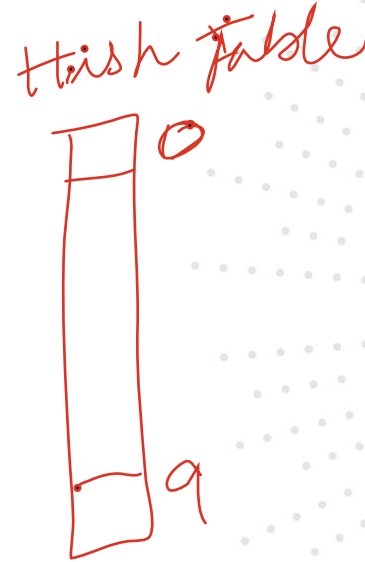
Hash Function

- The Mid-Square Method square the key and using a specific portion of the resulting digits as the hash value

  There are some other hash functions that exist, AND you can also create your own has functions. just make sure it meets the 3 Criteria!

- Example

  - Key: 123

  - Square the Key: 123^2= 15129

  - Extract middle digits: If we want to extract 2 digits, we take the middle digits from 15129, which could be '51'

  - Calculate Hash Index: If the hash table size is, say, 10, then the index would be 51 mod 10 = 1

🏛 UVA DATA SCIENCE

Hash Function

- In Python, the built-in **hash()** function returns a hash value for a given object

- You can define your own hash functions, especially when using custom objects as keys. You would override the __hash__() method in your class

- The default implementation of __hash__() for instances of user-defined classes returns a unique integer for each instance, based on its identity (memory address).

again, you can hash tuples and other IMMUTABLE data types. Int, float, stirng

You can't hash lists or sets

Hash Table in Python

- **Python's dict** is a built-in data structure that allows for the storage of key-value pairs, enabling efficient data retrieval.

- A dictionary in Python is implemented using a hash table, which means it uses a hash function to convert keys into indices for fast access.

In PYTHON, if you are inserting a key into a dictionary, and that key already exists, then that key's value will just be updated. generally, a hash table would create a new key with the unique value, but again, this is python implementation.

Ex.hash_table['apple'] = 1
hash_table['apple'] = 5

🏛 UVA DATA SCIENCE
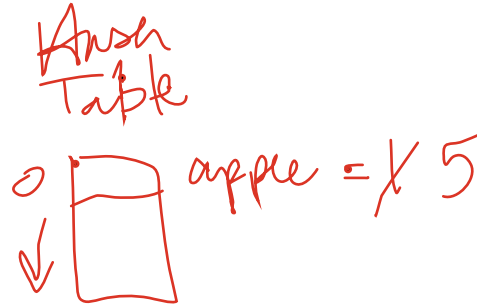
Collision

- Collisions occur when different keys have the same hash value

- Two strategies to handle collision:

  - Open addressing, a.k.a. closed hashing

  - Separate chaining, a.k.a. open hashing

- Difference has to do with whether collisions are stored *outside the hash table* (open hashing) or whether collisions result in storing one of the records at *another slot in the table* (closed hashing)
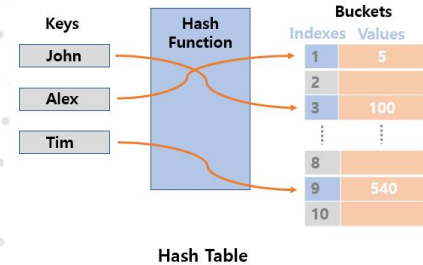
Open Addressing

- Open addressing is a collision resolution schemes that probe for an empty, or open, location in the hash table

- The sequence of locations that are examined is the probe sequence

- Probing Methods

  - Linear Probing

  - Quadratic Probing

  - Double Hashing

Linear Probing

- Linear probing searches the hash table sequentially, starting from the original location specified by the hash function
  - $h(k,i) = (h_1(k) + i) \bmod m$ for i=0,1,2,...
    - i i is the probe number and m is size of hash table
- When there is a collision, check the next available position in the table (i.e., probing)
- Probe Sequence
  - $h_1(k)$ -> First slot probed
  - $h_1(k) + 1 \bmod m$ -> Second slot probed
  - $h_1(k) + 2 \bmod m$ -> third slot probed
  - ....

🏛 UVA DATA SCIENCE

Linear Probing Example

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Linear Probing Visualization: https://visualgo.net/en/hashtable

Linear Probing

- Primary clustering occurs when multiple keys hash to the same index (resulting in a collision) and are subsequently placed in adjacent slots due to linear probing. This creates large contiguous blocks of filled slots

Primary Clustering

Quadratic Probing

- Quadratic probing uses a quadratic function to determine the next slot to check after a collision occurs.

$$h(k, i) = (h'(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$$

c1 and c2 are constants and h'(k) is ordinary hash function
c1 can be zero while $c_2 \neq 0$

- Probe Sequence
  - $h_1(k)$ -> First slot probed
  - $h_1(k) + 1 \bmod m$ -> Second slot probed
  - $h_1(k) + 4 \bmod m$ -> third slot probed
 - ….

Quadratic Probing Example

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	After insert 18	After insert 49	After insert 58	After insert 9
0			49	49	49
1					
2				58	58
3					9
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Quadratic Probing Visualization: https://visualgo.net/en/hashtable

Quadratic Probing

- Secondary clustering occurs when keys that have collided are distributed unevenly across the hash table, causing empty spaces or gaps between clusters
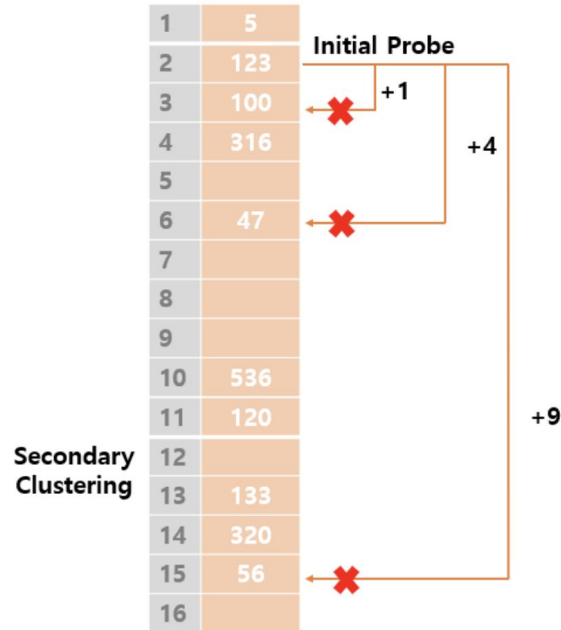
Double Hashing

- **Double hashing** is a collision resolution approach that combines two hash functions to determine the index for probing when a collision occurs.

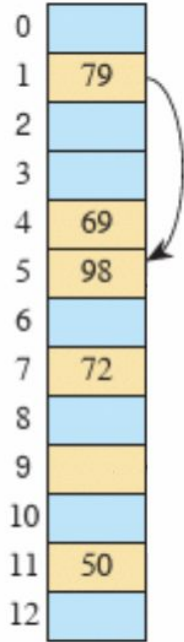$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m, \quad i=0,1,\dots$$

i i is the probe number and m is size of hash table and h1(k) + h2(k) are auxiliary hash functions

- Double hashing uses one hash function to determine the first slot and a second hash function to determine the increment for the probe sequence
- Initial probe: $h_1(k)$
- Second probe is offset by $h_2(k) \bmod m$, so on …
- It is particularly effective at reducing clustering compared to other methods, such as linear or quadratic probing.

Double Hashing

- Probe sequence

  - $h_1(k) \mod m$

  - $(h_1(k) + 1\, h_2(k)) \mod m$

  - $(h_1(k) + 2\, h_2(k)) \mod m$

  - …

Double Hashing Example

0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	
10	
11	50
12	

insert 14

$h_1(k)= k \bmod 13 \rightarrow 14 \bmod 13 = 1$

$h_2(k)=1+ k \bmod 11 \rightarrow 1+ 14 \bmod 11 = 4$

hash (14,13) = 1 (collision)

probe to 5 (collision)

probe to 9 (empty) -> insert 14 at index 9

Open Addressing

Feature	Linear Probing	Quadratic Probing	Double Hashing
Collision Resolution	Uses a fixed increment (next slot)	Uses a quadratic increment j^2	Uses a secondary hash function to determine the increment
Probing Sequence	h(k)+j	$h(k)+j2$	h'(k)+j · h2(k)
Clustering	Suffers from Primary Clustering	Reduces Primary Clustering, can still have secondary clustering	Reduces both primary and secondary clustering
Table Size Considerations	Can work with any size, but larger sizes help	Often requires a prime number for optimal performance	Ideally uses a prime number to ensure effective probing

Common Flags in Hash Table

- **Empty**: Indicates that the cell is unoccupied and does not contain any key-value pair.

- **Deleted**: Marks a cell that previously held a key-value pair but has been deleted. This flag helps in managing collisions, as it allows probing to continue even if the cell is not currently occupied.

- **Occupied**:Indicates that the cell contains a valid key-value pair.

- **Full**: This flag may indicate that the cell is filled and cannot accommodate new entries until a deletion occurs or resizing happens.

Hash Table Main Operations

- Insert: $h[k] = Value$

- Search: $h[k]$

- Delete: $Delete(h[k])$

Hash Table Main Operations

- Insertion

  ○ Compute the hash value of the key

  ○ Check the computed value

    ■ If the slot associated with index is marked empty

      - Insert the key at this index and mark slot as occupied

    ■ If the slot associated with index is occupied (collision occurs)

      - Use a probing strategy (e.g., linear probing, quadratic probing, or double hashing) to find the next available slot

Hash Table Main Operations

- Search/Lookup
  - Compute the hash value of the key
  - Check the computed index
    - If the key matches the value at that index, return the associated value
    - If the slot associated with index is empty (indicating no entry), conclude that the key is not present in the table.
    - If the slot associated with index is occupied but does not contain the key (collision occurs)
      - continue checking subsequent indices based on the probing method until:
        - The key is found, then return the associated value
        - An empty slot is encountered, indicating the key is not in the hash table.

Hash Table Main Operations

- Deletion
  - Compute the hash value of the key
  - Check the computed index
    - If slot associated with index is marked empty, the key is not in the table. Terminate the search
    - If the slot associated with index contains the key, delete the key from this index and mark the slot as deleted
    - If the slot associated with index is occupied but does not contain the key (collision occurs)
      - continue checking subsequent indices based on the probing method until:
        - The key is found, then delete it and mark the slot as deleted
        - An empty slot is encountered, indicating the key is not in the hash table.

🏛UVA DATA SCIENCE

Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion			
Search/Lookup			
Deletion			

UVA DATA SCIENCE

Time Complexity

worst case is every time you're accessing, you are having a collision, so you will have to linearly search

Operation	Best Case	Average Case	Worst Case
Insertion	O(1)	O(1)	O(n)
Search/Lookup	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)

- In best case, every key hashes to an index that is empty

- In the average case, the hash function distributes keys fairly uniformly across the hash table, leading to some keys mapping to same index but not too many

- In the worst case, every key hashes to the same index, resulting in a collision

Pros and Cons of Open Addressing

- Pros
  - Uses a single array, so lower memory overhead
  - Better cache locality due to contiguous memory usage
  - Easier to implement for simple cases
 - All data stored in the hash table array

- Cons
  - Can lead to clustering
  - Requires careful management of load factor and rehashing

Separate Chaining

- Separate chaining is a collision resolution method where each index in the hash table contains a linked list of entries that hash to the same index

your separate chain is storing the reference to a linked list in each index of the array

Cons, you will need extra memory if your linked lists are long
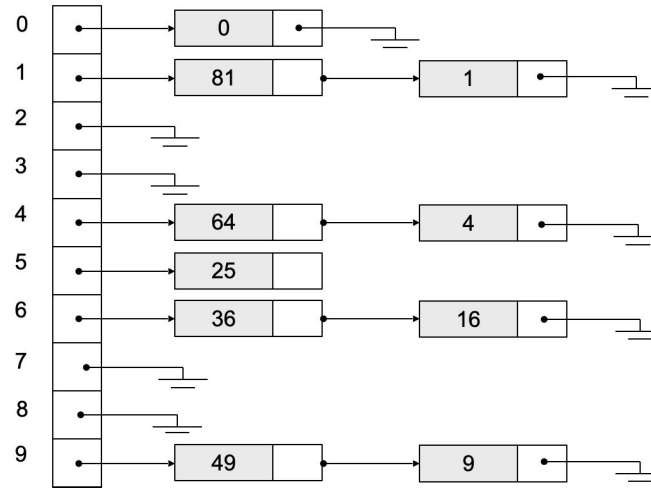
Separate Chaining Example
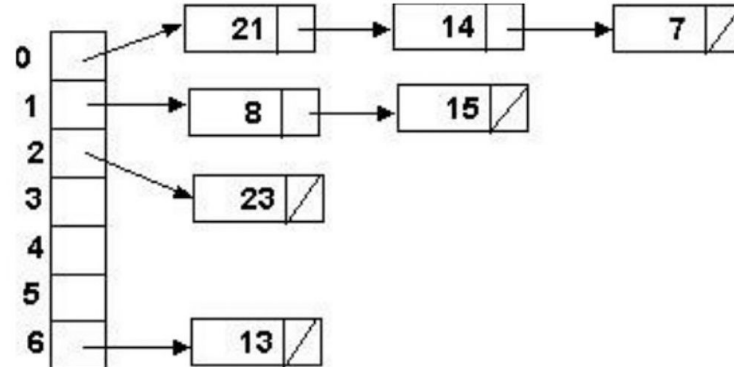
$h(23) = 23 \% 7 = 2$
$h(13) = 13 \% 7 = 6$
$h(21) = 21 \% 7 = 0$
$h(14) = 14 \% 7 = 0$ collision
$h(7) = 7 \% 7 = 0$ collision
$h(8) = 8 \% 7 = 1$
$h(15) = 15 \% 7 = 1$ collision

Separate Chaining Visualization: https://visualgo.net/en/hashtable

Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion			
Search/Lookup			
Deletion			

Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion	O(1)	O(1)	O(n)
Search/Lookup	O(1)	O(1)	O(n)
Deletion	O(1)	O(1)	O(n)

- In best case, every key hashes to an index that is empty

- In the average case, the hash function distributes keys fairly uniformly across the hash table, leading to some keys mapping to same index but not too many

- In the worst case, every key hashes to the same index, resulting in a collision

🏛 UVA DATA SCIENCE

Pros and Cons of Separate Chaining

- Pros
  - Can accommodate an arbitrary number of entries without needing to resize the entire structure
  - Effectively handles a large number of collisions since each slot can store multiple entries
  - Deleting an entry is straightforward: just remove it from the linked list
- Cons
  - Performance can degrade if chains become long, leading to $O(n)$ lookup times in the worst case
  - Requires additional memory for linked lists, which can lead to higher overall memory usage

Collision Handling in Python

- Python's specific approach is a combination of quadratic probing with **perturbation**.

- Perturbation adds an extra value derived from the hash code that helps distribute keys more evenly when collisions occur

Load Factor

What is the hash table is full?

Load Factor

- The **load factor** of a hash table is the ratio of the number of elements in the table (n) to the total number of slots (size of table) (m).

- Load Factor(α)= $\dfrac{\text{Number of Elements}}{\text{Size of Table}}$

- Load factor indicates how full the hash table is

- As the load factor increases (i.e., the table becomes nearly full), the performance of the hash table degrades due to more collisions.

the hash table doesn't wait for the table to be full. It will rehash based on the load factor which will double the size and rehash all the elements in the table.

Linear probing with load factor: https://yongdanielliang.github.io/animation/web/LinearProbing.html

Load Factor

- To maintain efficiency, Python dynamically resizes the hash table when the load factor exceeds a certain threshold (typically around 2/3)

- When the hash table becomes too full, it needs to be **resized** to maintain efficient performance

- When resizing occurs:

  - A larger array is allocated (usually doubled in size)

  - All existing keys are rehashed and placed into the new array

  - Resizing happens in O(n) time but is infrequent, ensuring the overall performance remains efficient

Load Factor

- If the load factor (α) in a hash table becomes too high, several issues can arise:

  high load factor would be 0.9 - 1
  prefered to havea reasonable
  threshold for load factor

  - Increased Collisions

  - Decreased Performance

  - Frequent Resizing

UVA DATA SCIENCE

Load Factor

- If the load factor (α) in a hash table becomes too high, several issues can arise:

  - Increased Collisions

  - Decreased Performance

  - Frequent Resizing

Load Factor

- For an open-address hash table with a load factor α=(n/m) <1, the expected number of probes in a successful search can be approximated by the formula

$$\frac{1}{\alpha} \ln \left(\frac{1}{1 - \alpha} \right)$$

  assuming uniform hashing and assuming that each key in the table is equally likely to be searched for

- Given an open-address hash-table with load factor α = n/m < 1, the expected number of probes in an unsuccessful search is at most 1/(1−α), assuming uniform hashing

Applications of Hash Table

- **Caching**: Storing results of expensive function calls for quick retrieval
  facebook/instagram uses a hash table
- **Databases**: Implementing indexing systems to quickly look up records

- **Symbol Tables**: In compilers for tracking variable names and scopes

- **Implementing Sets**: Python's set is also built using a hash table