Hash Tables allows for efficient searching, inserting, and deletion of values. They use a hash function to map all of the value’s keys into an index in the storage array, which is the hash index.

Most hash functions determine the hash index through calculating a value’s key % with the table size. Collisions are bound to occur however, e.g.

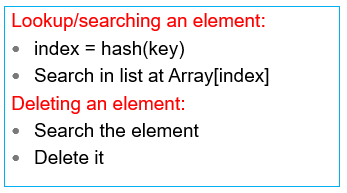
key 29 % 13 = 3

key 16 % 13 = 3

There are methods to address collision

Open Hashing/Separate Chaining

Create a linked list for each index in the array. If collisions occur at the hash index, append the element into the index as a linked list.



Best-case Time complexity:

Insert: O(1) – Can always add a new element (as a node) onto the linked list corresponding to the index.

Search: O(1) – Element is first element in the linked list

Delete: O(1) – Element to delete is first element in the linked list

Worst-case Time complexity:

Insert: O(1) – Can always add a new element onto the linked list

Search: O(N) - At worst need to traverse size of linked list of N elements

Delete: O(N) – At worst need to traverse size of linked list of N elements to delete an element.

Closed Hashing (Open Addressing)

Each index in the table contains one element. Hence if collisions occur, many methods to find a new index to insert:

1. Linear Probing

If collision, sequentially look at the next indices until you find an empty index OR return fail if the hash table is full. Make sure to point to starting indices if reached ending index, hence do (index + 1) % table\_size.

When probing, we want to increment our search by 1, hence we sequentially search with increment of a constant c like where:

index = (hash(key) + c\*i) % table\_size.

Problem is that collisions from nearby hash values tend to merge into big blocks, and lookup can degenerate into a linear O(N) search, called **primary clustering**.

Deletion – Search the element, delete it, and make sure we set a “deleted” value in the position otherwise may result in errors.

E.g. Searching for keys that have been probed and then probed into the empty index where the key was deleted, and incorrectly returning key not found even though the key is later on in the probe.

Worst-case Time Complexity

Search:

Delete:

1. Quadratic Probing

Like linear probing but collisions probe is in quadratic jumps where:

index = (hash(key) \* c\*I + d\*i^2) % table\_size.

Problem with this approach is that probing is not guaranteed to probe every location in the table. An insert could fail while there is still an empty location. Problem Is called **secondary clustering**.

If two elements have the same hash index, the jumps are the same for both elements.

1. Double Hashing

Resolves **secondary clustering** by using two different hash functions: one to determine the initial index and the other to determine the amount to jump where

index = (hash1(key) + i\*hash2(key)) & table\_size

E.g. suppose hash1() is key % 13 and hash2 is key % 7.

Insert 40 => hash1(40) = 1, hash2(40) => 5

So, check index 1, if collide, then jump by 5 positions until empty spot.

1. Cuckoo Hashing

The only one which guarantees in the worst case that

Searching – O(1)

Deletion – O(1)

Insertion may be significantly higher – but expected cost is O(1)

Main Idea: Use two different hash functions hash1() and hash2() and two different hash tables T1 and T2 of possibly different sizes.

Insertion

During insertion, insert key to its hash index in T1 using hash1().

If collision occurs, place new key into T1 and place the old key into T2 using hash2() on it to compute the hash index.

This process can keep repeating where:

* 1. Insert a key at T1 using hash1()
  2. Collision occurs, new key replaces old key
  3. old key goes to T2 using hash2
  4. Collision occurs, old key replaces the one inside
  5. replaced key in T2 goes to T1 using hash1()
  6. repeat until empty position found in T1 or T2 for the key.

**Problem**, this process may lead to an infinite loop. Hence, we want to define a **maxloop** number of iterations.

Search

A key is either at

index = hash1(key) in T1

index = hash2(key) in T2

Hence, we just need to look at two cells, one in T1 and one in T2. So very efficient O(1) time.

Delete

Same as Search. Just need to look at two cells. After finding, then delete the key.