# Week 7 - Hashing

AD 325 - 2021

## Contents

***Learning Outcomes***

- overview of hashing
- hash functions
- index hashing
- collision handling

***Reading & Videos***

- Carrano & Henry: Chapters 22, 23
- https://www.coursera.org/learn/algorithms-part1/home/week/6

***Reference***

- https://algs4.cs.princeton.edu/34hash/
- https://www.geeksforgeeks.org/hashing-data-structure/

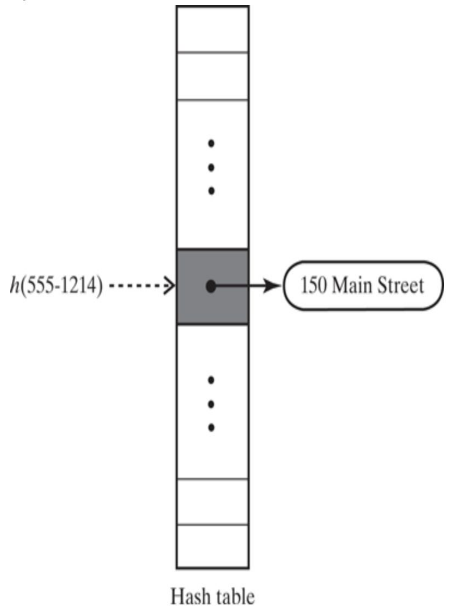
## Overview

**Hashing** is a technique for implementing a dictionary that can ideally result in O(1) (constant) search times.

Hashing determines the **array index** of an entry based on its search key. Entries can then be accessed directly according to their index, rather than by traversing the array.

Arrays constructed through hashing are called **hash tables.** In most cases, hash tables are **sparse**, with only a few positions used.

While useful for searching, hashing cannot provide traversal of search keys in sorted order.



## Hash Functions

A hash function produces an integer (**hash index**) based on the search key to determine an entry's array **index** position.

A **perfect** hash function should produce a unique index for each unique search key, but this is rarely achieved.

Good hash functions should at least:

- minimize **collisions (**where different search keys map to the same array index)
- be fast to compute

Hashing typically involves:

1. Converting the search key to an integer (**hash code**)
2. Compressing the hash code into the range of indices for the hash table

Hash code computation should always return the same code for the same input data.

## Hashing Methods

Some common approaches for hashing:

- Use all or part of the search-key data (e.g. part of a phone number)
- **modular hashing** - choose a prime number M for array size. For any positive integer k, compute the remainder (modulus) when dividing k by M. For floating-point numbers, use binary representation of the key.
- **Strings** - compute a modular hash value based on value & position of each character in the string:

```
int hash = 0;
for (int i = 0; i < s.length(); i++)
     hash = (R * hash + s.charAt(i)) % M;
```

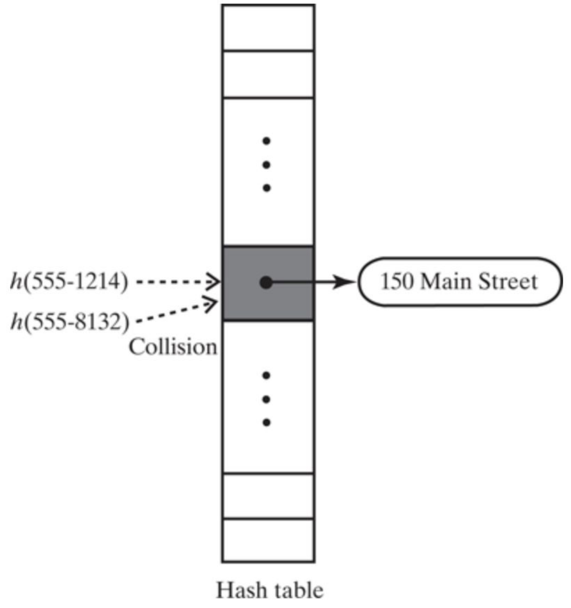
Modular hashing reduces chance of collisions by distributing entries uniformly throughout the hash table.

## Collision Handling

Where a hash function maps a search key to an array index already in use, programs can either:

- find another open position (**open addressing**)

- change hash table structure to allow more than one value per index (**separate chaining**)



## Open Addressing

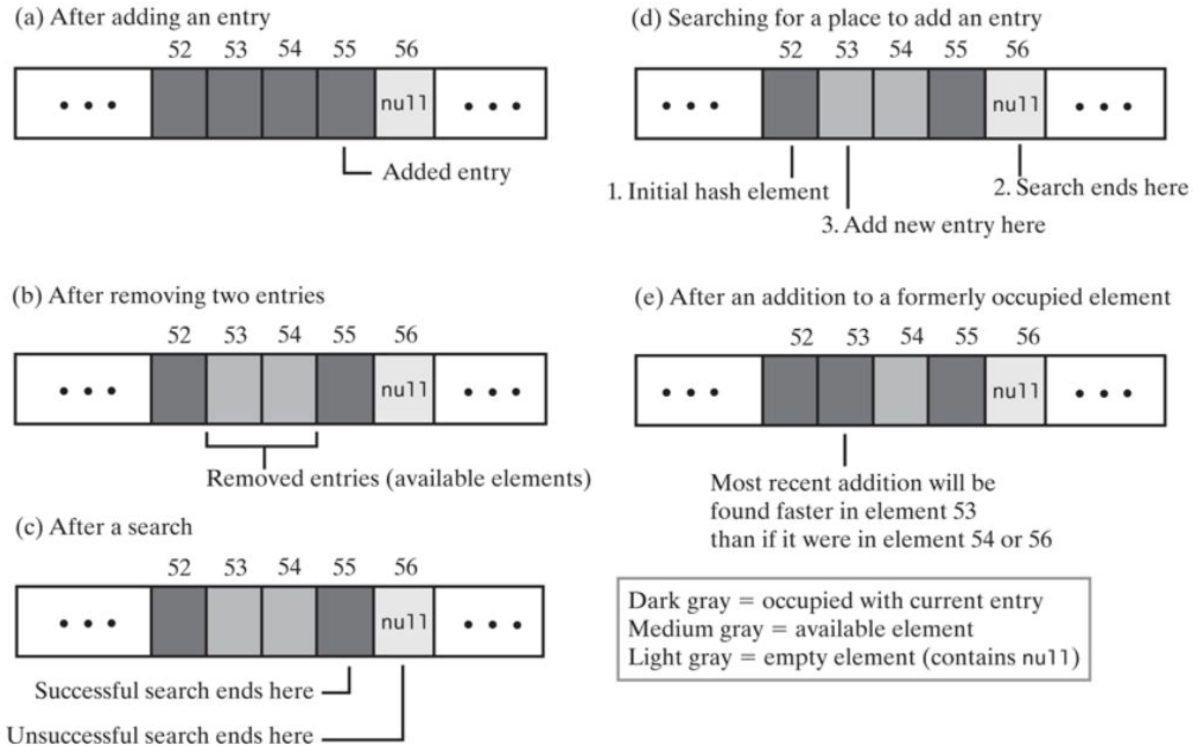
Open addressing is the process of finding an unused position in the hash table through **probing**.

A **Linear Probing** algorithm looks at each consecutive table element after the collision, for an unused position. Probing continues until an open position is found.

When items are removed from the hash table, their position is marked as **available** to distinguish from **empty** positions and ensure searches aren't impacted.

Linear probing can examine every table element, but can result in **primary clustering** that slows searches.

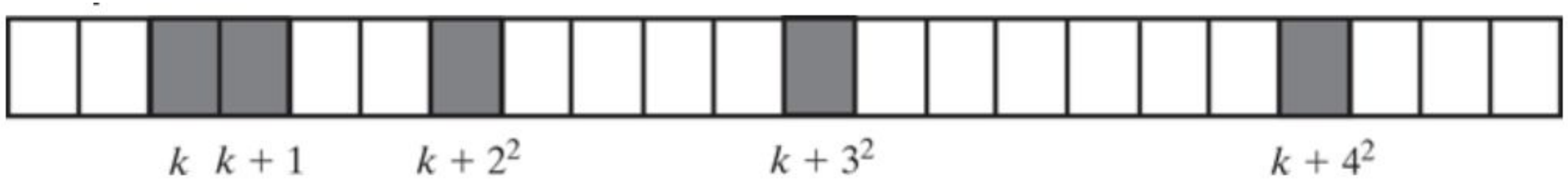
## Linear Probing Sequence


(a) After adding an entry
52 53 54 55 56
• • • null • • •
Added entry
(b) After removing two entries
52 53 54 55 56
• • • null • • •
Removed entries (available elements)
(c) After a search
52 53 54 55 56
• • • null • • •
Successful search ends here
Unsuccessful search ends here
(d) Searching for a place to add an entry
52 53 54 55 56
• • • null • • •
1. Initial hash element
2. Search ends here
3. Add new entry here
(e) After an addition to a formerly occupied element
52 53 54 55 56
• • • null • • •
Most recent addition will be found faster in element 53 than if it were in element 54 or 56
Dark gray = occupied with current entry
Medium gray = available element
Light gray = empty element (contains null)

## Quadratic Probing & Double Hashing

**Quadratic Probing** is similar to Linear probing, but uses a quadratic step value (k + j2 for j >= 0) - e.g. k +1, k + 4, k + 9 ...

Quadratic probing can result in secondary clustering where entries that collide with an existing entry use the same probe sequence.



**Double hashing** uses a second hash function to generate the increment for a probe sequence. This avoids both primary & secondary clustering, and can reach every element in the hash table.

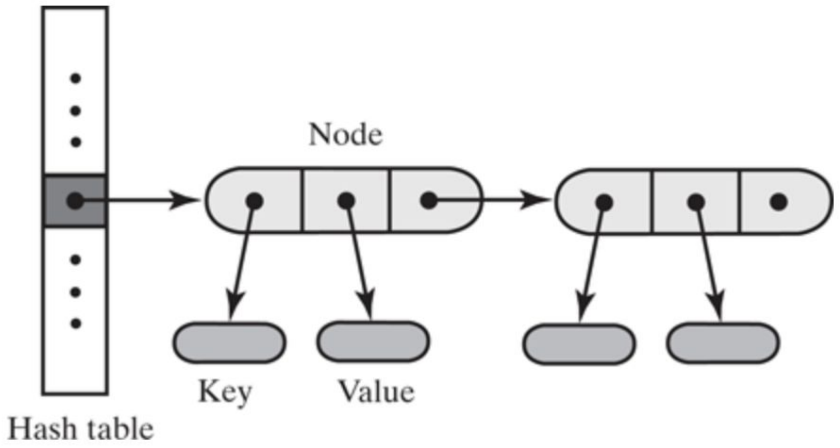
The second has function should differ from the first and return a non-zero value.

## Separate Chaining

Separate chaining treats each element in the hash table as a **bucket** that can hold more than one entry. This method avoids problems of clustering and endless searches.

When searching for a key, the algorithm searches key-value pairs in the bucket that matches the key.

Buckets can be represented with many different data structures - e.g. list, sorted list, array, linked nodes. Linked nodes are optimal since that approach does not require a fixed memory allocation.



## Hashing Efficiency

Resolving a collision takes more time than the hashing function & tends to be the prime contributor to the cost of hashing.

**load factor** measures the cost of collision resolution. It is the ratio of number of entries in the table to the size of the table. Restricting the size of load factor improves hashing performance. With separate chaining, load factor is independent of how many buckets are empty.

For linear probing, the number of comparisons increases as the table fills, and increases rapidly when the table is > 0.5 full.