

Objectives

- To understand what hashing is and for what hashing is used.
- To obtain the hash code for an object and design the hash function to map a key to an index.
- To handle collisions using open addressing.
- To know the differences among linear probing, quadratic probing, and double hashing.
- To handle collisions using separate chaining.
- To understand the load factor and the need for rehashing.

Why Hashing?

The preceding chapters introduced search trees. An element can be found in $O(\log n)$ time in a well-balanced search tree.

Is there a more efficient way to search for an element in a container? This chapter introduces a technique called **hashing**. You can use hashing to implement a map or a set to search, insert, and delete an element in $O(1)$ time.

Map

A *map* is a data structure that stores entries. Each entry contains two parts: *key* and *value*. The key is also called a *search key*, which is used to search for the corresponding value. For example, a dictionary can be stored in a map, where the words are the keys and the definitions of the words are the values.

A map is also called a *dictionary*, a *hash table*, or an associative array. The new trend is to use the term map.



Map key to an index in HashTable

What Is Hashing?

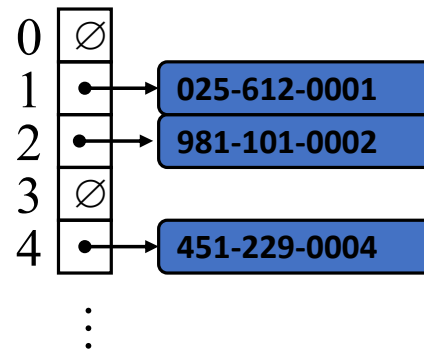
If you know the index of an element in the array, you can retrieve the element using the index in $O(1)$ time. So, can we store the values in an array and use the key as the index to find the value? The answer is yes if you can map a key to an index.

The array that stores the values is called a **hash table**. The function that maps a key to an index in the hash table is called a **hash function**.

Hashing is a technique that retrieves the value using the index obtained from key without performing a search.

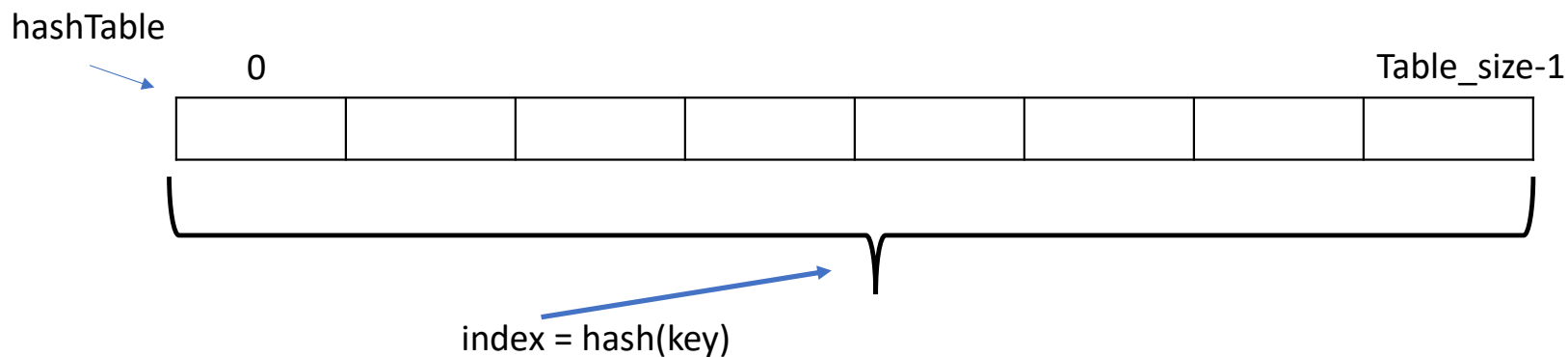
More General Kinds of Keys

- But what should we do if our keys are not integers in the range from 0 to $N - 1$?
 - Use a **hash function** to map general keys to corresponding indices in a table.
 - For instance, the last four digits of a Social Security number.



Hashing

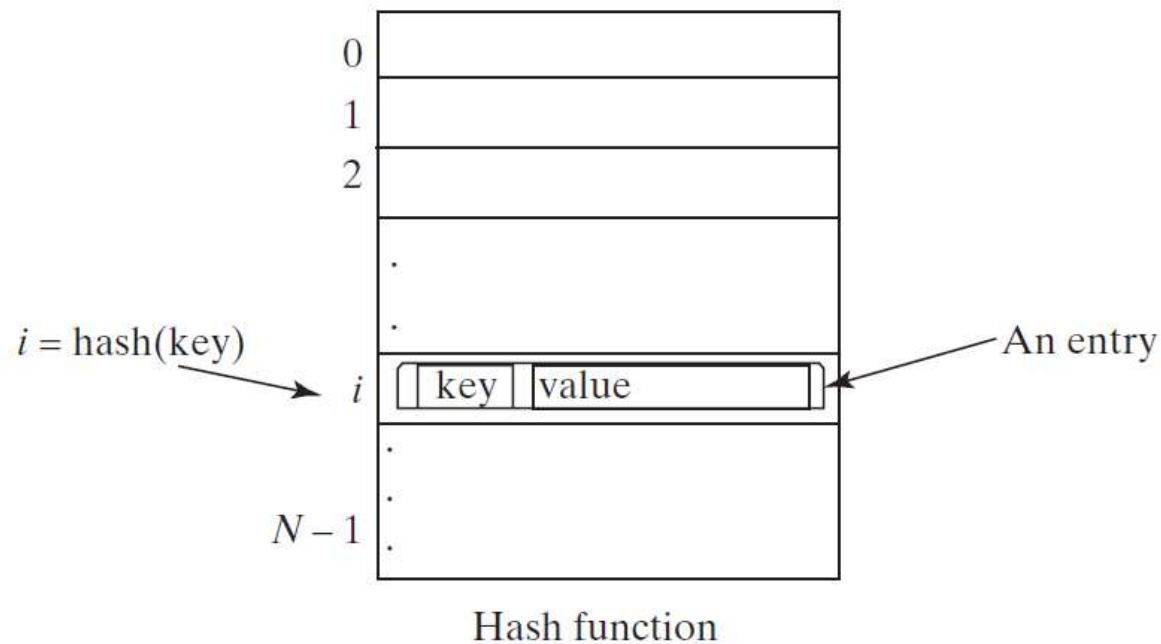
- A technique that determines an index into a table using only an entry's search key
- Hash function
 - Takes a search key and produces the integer index of an element in the hash table
 - Search key is mapped, or hashed, to the index



Hash Function and Hash Codes

A typical hash function:

1. first converts a search key to an integer value called a *hash code*,
2. Compresses the hash code into an index to the hash table.



Ideal Hashing

- Simple algorithms for the dictionary operations that add and retrieve

□ `put(k, v):`

insert entry (k, v) into the map M; if key k is not already in M,
then return null; else, return old value associated with k

□ `get(k):` if the map M has an entry with key k, return its associated value; else, return null

Hash Functions and Hash Tables



- A **hash function** h maps keys of a given type to integers in a fixed interval $[0, N - 1]$
- Example:
$$h(x) = x \bmod N$$

is a hash function for integer keys
- The integer $h(x)$ is called the **hash value** of key x
- A **hash table** for a given key type consists of
 - Hash function h
 - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

Typical Hashing

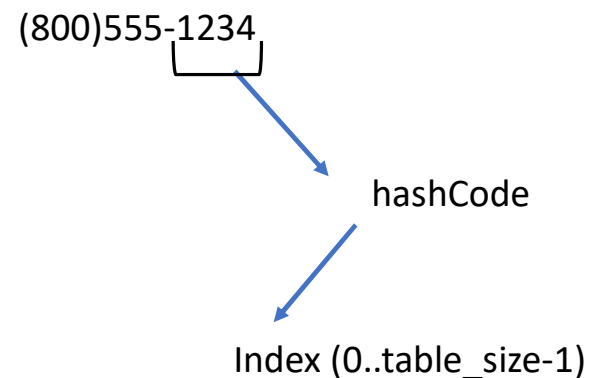
- Typical hash functions perform two steps:
 - Convert search key to an integer
 - Called the hash code.
 - Compress hash code into the range of indices for hash table.

Algorithm `getHashIndex(phoneNumber)`

// Returns an index to an array of tableSize elements.

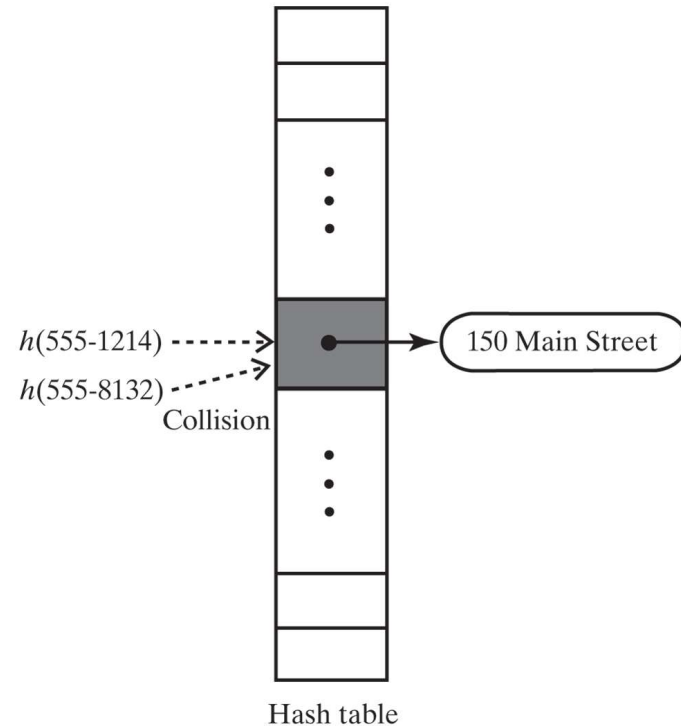
i = last four digits of phoneNumber

return $i \% \text{tableSize}$



Typical Hashing

- Most hash functions are not perfect,
 - Can allow more than one search key to map into a single index
 - Causes a collision in the hash table
- Consider **tableSize = 101**
- **getHashIndex(555-1264) = 52**
- **getHashIndex(555-8132) = 52**
also!!!



Hashing

- A perfect hash function
 - Maps each search key into a unique location of the hash table
 - Possible if all the search keys are known
- Collisions
 - Occur when the hash function maps more than one item into the same array location
- Collision-resolution schemes
 - Assign locations in the hash table to items with different search keys when the items are involved in a collision
- Requirements for a hash function
 - Be easy and fast to compute
 - Place items evenly throughout the hash table

Computing Hash Codes

- Java's base class **Object** has a method **hashCode** that returns an integer hash code
 - By default returns memory address of the object
- A class should define its own version of **hashCode**
 - Override: whenever the equals() method is overridden to ensure two equal objects return the same hashcode
 - Multiple execution of a program with same key should give same integer
 - Two unequal objects may have the same hashcode, must avoid too many such cases

A hash code for a string

➤ Approach 1:

- Using a character's Unicode integer is common

➤ Approach 2:

- Generate a hash code that takes the position of characters into consideration.
- Let the hash code be:

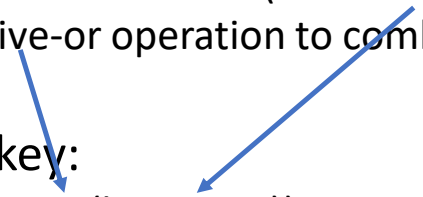
$$s_0 * b^{(n-1)} + s_1 * b^{(n-2)} + \dots + s_{n-1}, \text{ where } s_i \text{ is } s.\text{charAt}(i)$$

- The expression is a polynomial for some positive value (b).

▪ Polynomial hash code

$$(\dots((s_0 * b + s_1) * b + s_2) * b + \dots + s_{n-2}) * b + s_{n-1}$$

Hash Code for a Primitive type

- If data type is **int (32 bits)** → `Integer.hashCode()`
 - Use the key itself
 - For **byte (8 bits), short (16 bits), char (16 bits)**:
 - Cast as **int**
 - **For long (64 bits)**
 - Casting from long to int (not good idea)
 - All keys that differ in only the first 32 bits will have the same hash code.
 - To take the first 32 bits into consideration (folding)
 - divide the 64 bits into two halves (bit-wise right-shift operator)
 - perform the exclusive-or operation to combine the two halves
 - Hash code for long key:
`int hashCode = (int)(key ^ (key >> 32));`
- 

➤ Other primitive types (Manipulate internal binary representations)

- Hash code for float key (32 bits) → `Float.hashCode()`:

`Float.floatToIntBits(key)` → `Float.floatToIntBits(12.2f)` → 1094923059

`Float.floatToIntBits(2.2f)` → 1074580685

- Hash code for double key (64 bits) → `Double.hashCode()`:

`long bits = Double.doubleToLongBits(key);`

`int hashCode = (int)(bits ^ (bits >> 32));`

Compressing a Hash Code

- Common way to scale an integer
 - Use Java mod operator %: `hashCode % n`
- Best to use a prime number for n
- Prime numbers often give good distribution of hash values
- Java API implementation for `java.util.Map`
 - n is set to an integer power of 2
 - Use logical bit-wise and (&) compress the hashCode
Index = `hashCode & (n-1)` → index will be in range of 0..(n-1)

`n=4 , hashCode=11`

Resolving Collisions

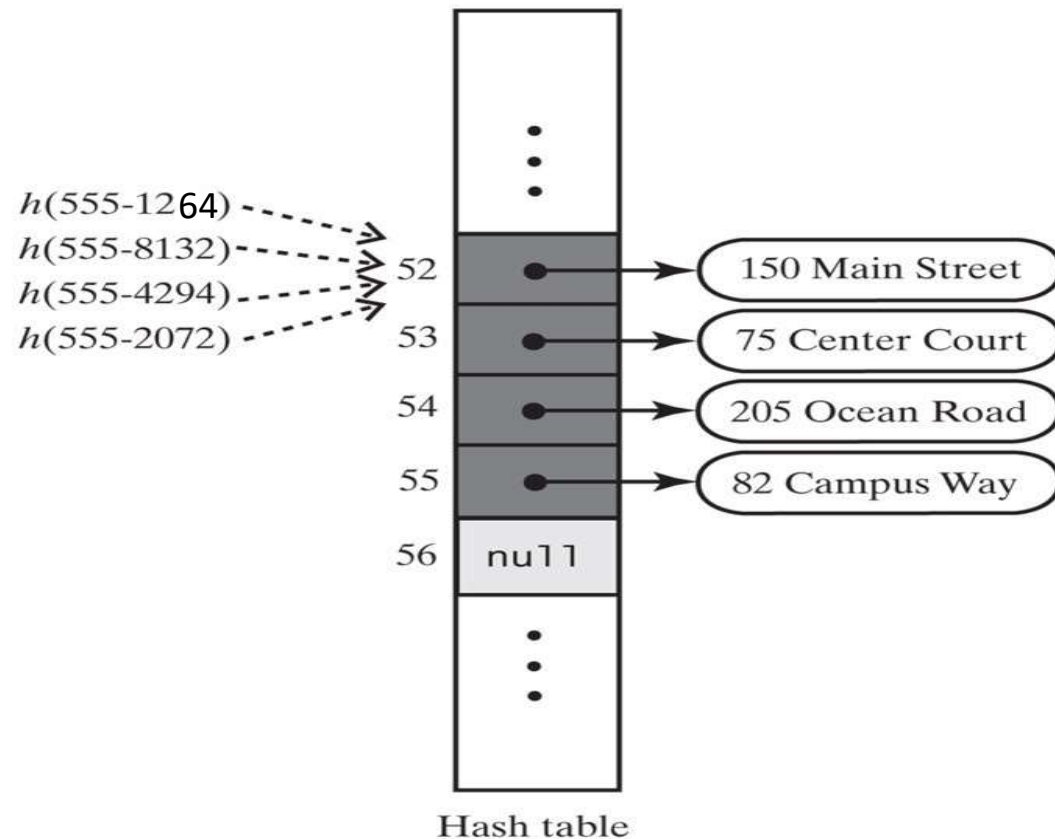
- Collision:
 - Hash function maps search key into a location in hash table already in use
- Two choices:
 - Use another location in the hash table
 - Change the structure of the hash table so that each array location can represent more than one value

Resolving Collisions

- **Linear probing**

- Resolves a collision during hashing by examining consecutive locations in hash table
 - Beginning at original hash index
 - Find the next available one
- Table locations checked make up probe sequence
- If probe sequence reaches end of table, go to beginning of table (circular hash table)
 - Possible problem
 - Primary clustering

Linear Probing



© 2019 Pearson Education, Inc.

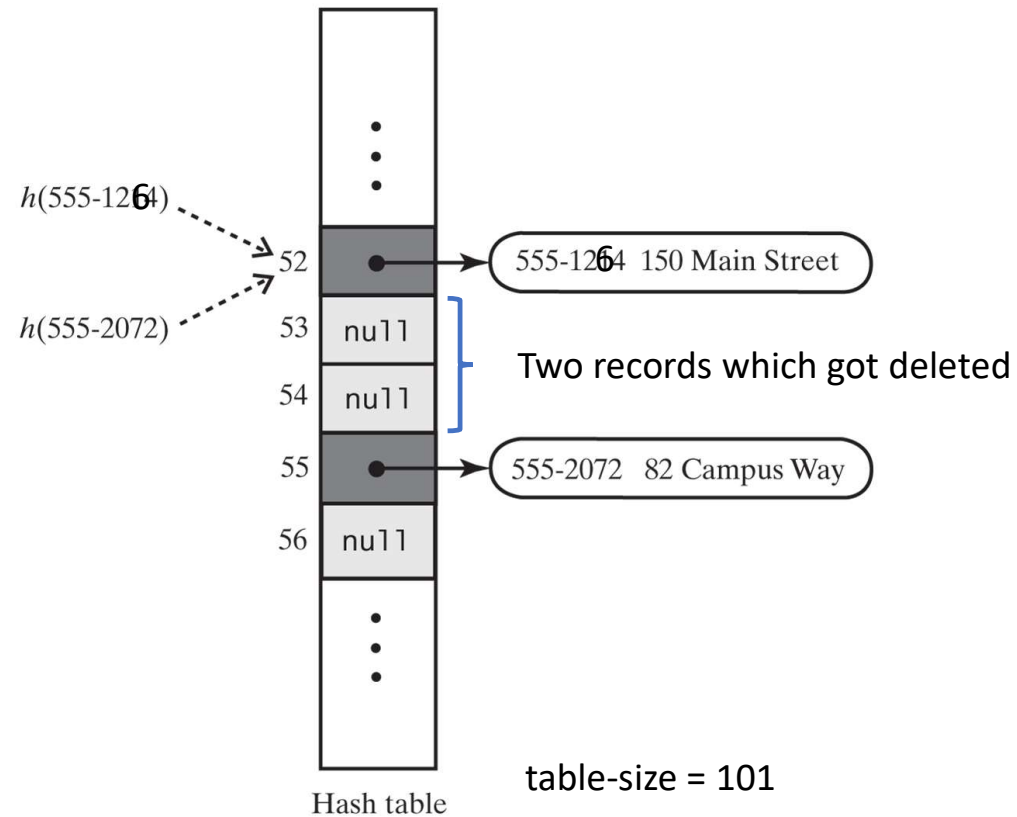
- The effect of linear probing after adding four entries whose search keys hash to the same index

Linear Probing

Insert 44, 4, 16, 28, 21, 26 into a table of size 11

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Linear Probing



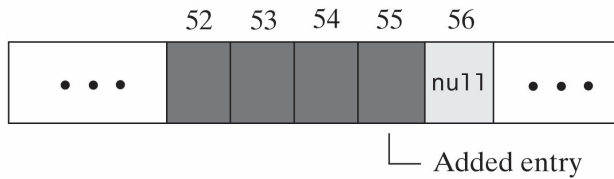
© 2019 Pearson Education, Inc.

- A hash table if remove used `null` to remove entries

Linear Probing

Table-size=101

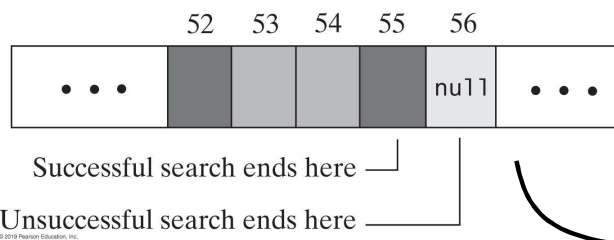
(a) After adding an entry



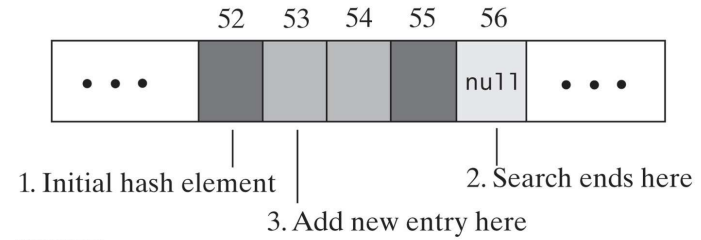
(b) After removing two entries



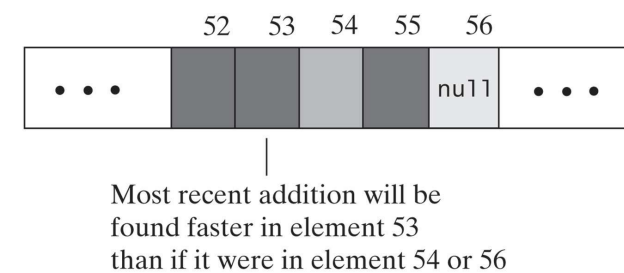
(c) After a search



(d) Searching for a place to add an entry



(e) After an addition to a formerly occupied element



Dark gray = occupied with current entry
Medium gray = available element
Light gray = empty element (contains null)

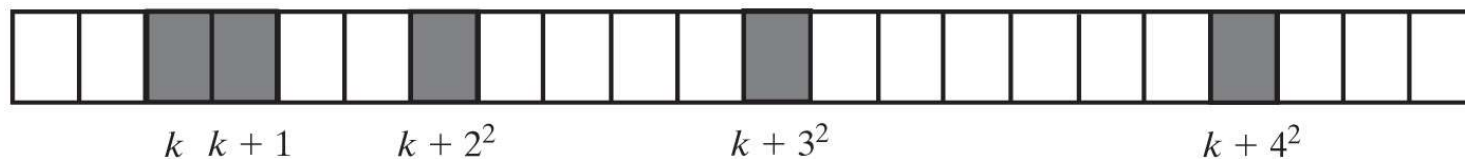
- The linear probe sequence in various situations

Clustering

- Collisions resolved with linear probing cause groups of consecutive locations in hash table to be occupied
 - Each group is called a ***cluster***
- Bigger clusters mean longer search times following collision

Open Addressing with Quadratic Probing

- Linear probing looks at consecutive locations beginning at index k
- Quadratic probing:
 - Considers the locations at indices $k + j^2$
 - Uses the indices $k, k + 1, k + 4, k + 9, \dots$



© 2019 Pearson Education, Inc.

A probe sequence of length five using quadratic probing

Possible problem:

Secondary clustering: entries that collide with an occupied entry use the same probe sequence

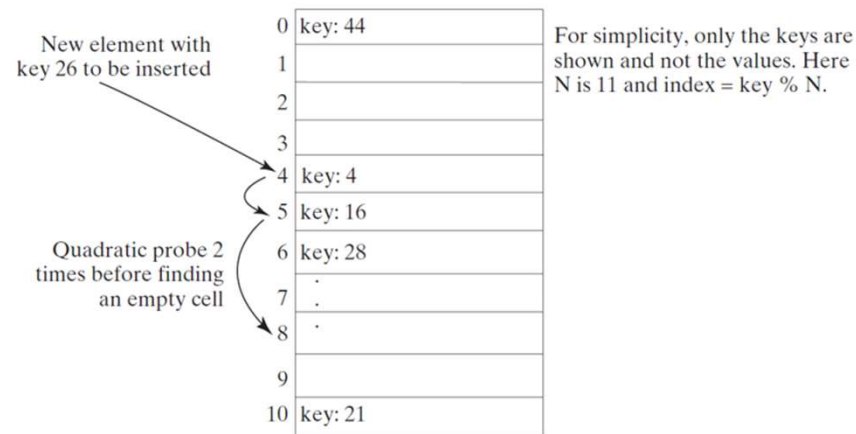
Quadratic Probing

<https://liveexample.pearsoncmg.com/dsanimation/QuadraticProbingBook.html>

Quadratic probing can avoid the clustering problem in linear probing. Linear probing looks at the consecutive cells beginning at index k .

Quadratic probing increases the index by j^2 for $j = 1, 2, 3, \dots$

The actual index searched are $k, k + 1, k + 4, \dots$



Open Addressing with Double Hashing

- Linear probing and quadratic probing add increments to k (*initial index*) to define a probe sequence
 - Both are independent of the search key
- **Double hashing** is an open-addressing collision resolution technique that uses 2 different hash functions to compute bucket indices. Using hash functions h_1 and h_2 , a key's index in the table is computed with the formula $(h_1(\text{key}) + i * h_2(\text{key})) \bmod (\text{tableSize})$. Inserting a key uses the formula, starting with $i = 0$, to repeatedly search hash table buckets until an empty bucket is found. Each time an empty bucket is not found, i is incremented by 1. Iterating through sequential i values to obtain the desired table index is called the **probing sequence**.
- This is a key-dependent method.

Double Hashing

Double hashing uses a secondary hash function on the keys to determine the increments to avoid the clustering problem.

$k = h(\text{key}) = \text{key} \% \text{table-size}$ (table-size=11)

$h'(\text{key}) = 7 - \text{key} \% 7;$

Insert 44, 4, 16, 28, 21, 26 into a table of size 11

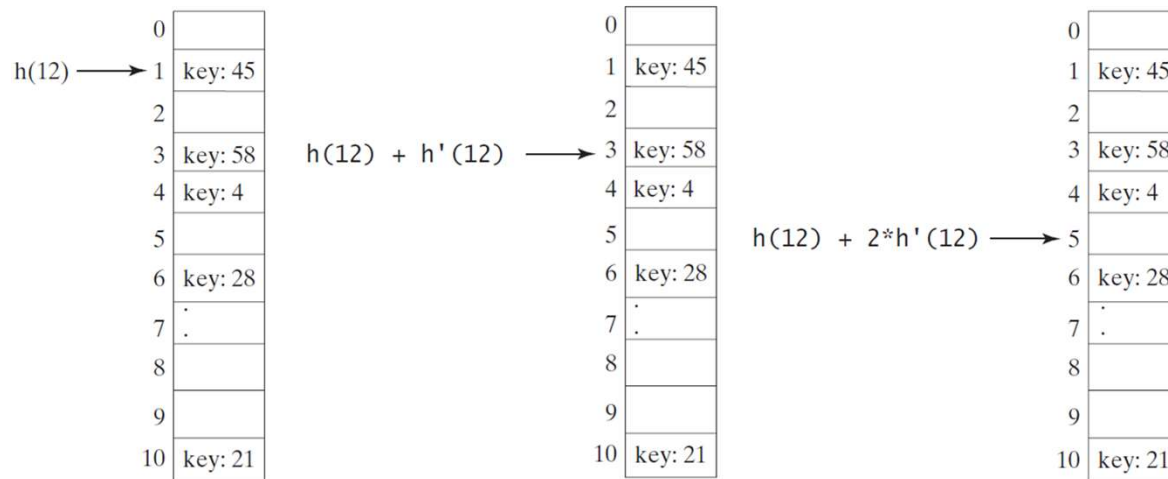
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Double Hashing

<https://liveexample.pearsoncmg.com/dsanimation/DoubleHashingeBook.html>

Double hashing uses a secondary hash function on the keys to determine the increments to avoid the clustering problem.

$$h'(k) = 7 - k \% 7;$$



Potential Problem with Open Addressing

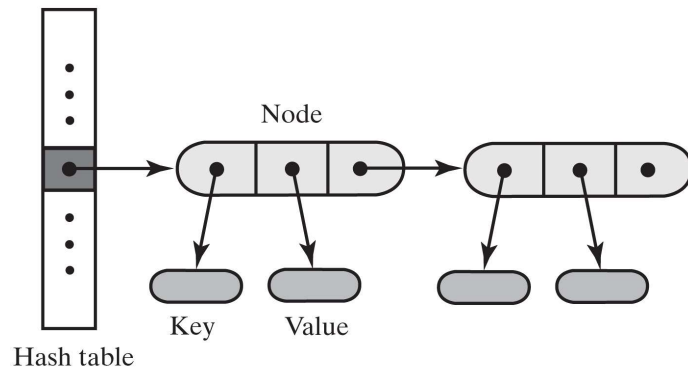
- Recall each location is either occupied, empty, or available
 - Frequent additions and removals can result in no locations that are null
- Thus searching a probe sequence will not work
- Consider separate chaining as a solution

Resolving Collisions

- Approach 2: Restructuring the hash table
 - Changes the structure of the hash table so that it can accommodate more than one item in the same location
 - Buckets
 - Each location in the hash table is itself an array called a bucket
 - Separate chaining
 - Each hash table location is a linked list

Separate Chaining

- Alter the structure of the hash table
 - Each location can represent more than one value.
 - Such a location is called a bucket
- Decide how to represent a bucket
 - **list, sorted list**
 - **array**
 - **linked nodes**
 - **vector**



A hash table for use with separate chaining; each bucket is a chain of linked nodes

Separate Chaining Animation

<https://liveexample.pearsoncmg.com/dsanimation/SeparateChainingBook.html>

Hashing Separate Chaining Animation by Y. Daniel Liang

Enter the table size and press the enter key to set the hash table size. Entering the load factor threshold factor and press the enter key to set a new load factor threshold. Enter an integer key and click the Search button to search the key in the hash set. Click the Insert button to insert the key into the hash set. Click the Remove button to remove the key from the hash set. Clicking the Remove All button to remove all entries in the hash set. For the best display, use integers between 0 and 99.

Current table size = 11. Number of keys = 4. Current load = 0.36. Load factor threshold = 0.5.

[0]		
[1]	→	1
[2]		
[3]		
[4]	→	48
[5]		
[6]		
[7]		
[8]		
[9]	→	31
[10]	→	21

Enter Initial Table Size: Enter a Load Factor Threshold:

Enter a key:

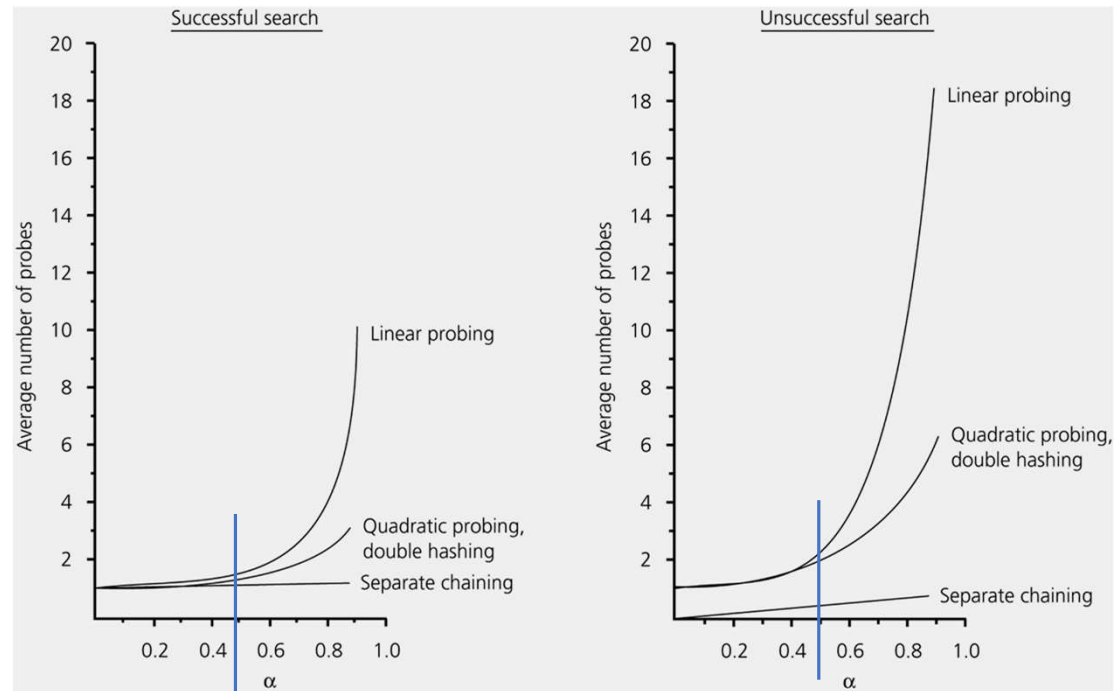
The Efficiency of Hashing

- An analysis of the average-case efficiency of hashing involves the load factor
 - Load factor α
 - Ratio of the current number of items in the table to the maximum size of the array table
 - Measures how full a hash table is
 - Should not exceed $2/3$
 - Hashing efficiency for a particular search also depends on whether the search is successful
 - Unsuccessful searches generally require more time than successful searches

The Efficiency of Hashing

- Linear probing
 - Successful search: $\frac{1}{2}[1 + \frac{1}{1-\alpha}]$
 - Unsuccessful search: $\frac{1}{2}[1 + \frac{1}{(1-\alpha)^2}]$
- Quadratic probing and double hashing
 - Successful search: $\frac{-\log_e(1-\alpha)}{\alpha}$
 - Unsuccessful search: $\frac{1}{1-\alpha}$
- Separate chaining
 - Insertion is $O(1)$
 - Retrievals and deletions
 - Successful search: $1 + \frac{\alpha}{2}$
 - Unsuccessful search: α

The Efficiency of Hashing



The relative efficiency of four collision-resolution methods

What Constitutes a Good Hash Function

- A good hash function should
 - Be easy and fast to compute (using % or bitwise operators)
 - Avoid clusters : Scatter the data evenly throughout the hash table

- Consider:

key \rightarrow nine digits , table-size \rightarrow 40, hashTable[0..39]

$H(\text{key}) \rightarrow (\text{first two digits} \% 40)$



hash code

- Assume keys are unique
- 60 keys will map to hashTable[0..19]
- 40 keys will map to hashTable[20..39]

- Java JCF HashMap provides

```
/** Hash function */
```

```
private int hash(int hashCode) {  
    return supplementalHash(hashCode) & (capacity - 1);  
}
```

```
/** Ensure the hashing is evenly distributed */
```

```
private static int supplementalHash(int h) {  
    h ^= (h >>> 20) ^ (h >>> 12);  
    return h ^ (h >>> 7) ^ (h >>> 4);  
}
```

Exclusive OR Unsigned shift right

For example, both **11100101 & 00000111** and **11001101 & 00000111** yield **00000101**.

But **supplementalHash(11100101) & 00000111** and **supplementalHash(11001101) & 00000111** will be different.

Using a supplemental function reduces this type of collision.