Fall 2019

Hashing

CMPE 250 - Data Structures & Algorithms

Presenter: Meriç Turan

24 OCTOBER 2019

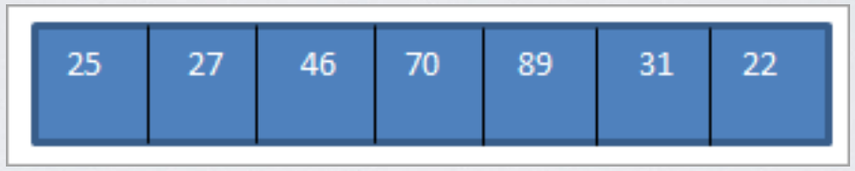
General Idea of Hashing

- The implementation of hash tables is frequently called hashing.
- Hashing is a technique used for performing **insertions**, **deletions**, and **finds** in constant average time.
- Tree operations that require any ordering information among the elements are not supported efficiently.
- Thus, operations such as **findMin**, **findMax**, and the printing of the entire table in **sorted** order in linear time are not supported.
- The ideal hash table data structure is merely **an array of some fixed size** containing the items.

General Idea of Hashing

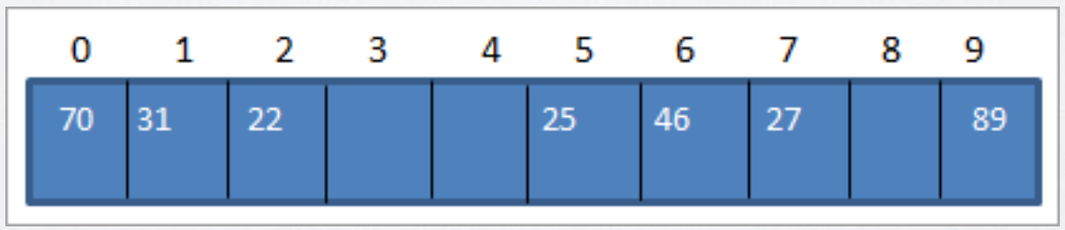
- For instance, an item could consist of a string (**key**) and additional data members (**value**) (e.g., a name that is part of a large employee structure).
- Each key is mapped into some number in the range 0 to *tableSize-1* and placed in the appropriate cell.
- The mapping is called a **hash function**, which ideally should be simple to compute and should ensure that **any two distinct keys get different cells**.
- Since there are a **finite number of cells** and a virtually **inexhaustible supply of keys**, this is clearly impossible, and thus we seek a hash function that **distributes the keys evenly** among the cells.
- In another words, we need to come up with a function that **decides what to do when two keys hash to the same value** (this is known as a collision).

Sample Hash Table

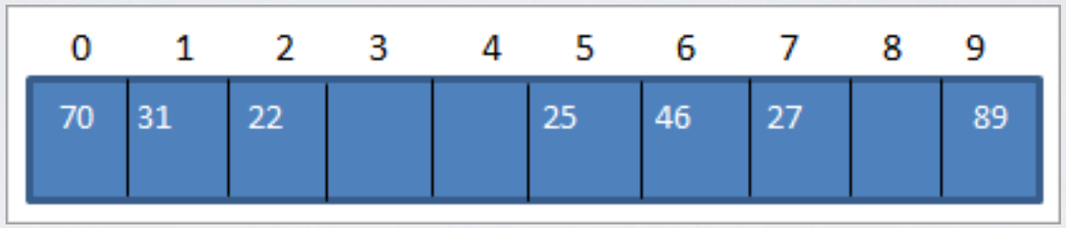


$$hash(x) = x \bmod 10$$

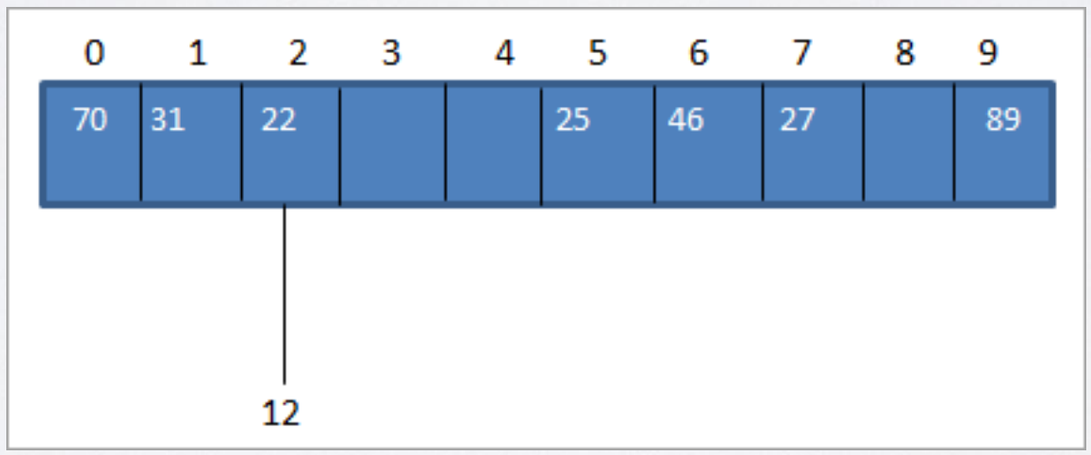
Data item	Hash function	Hash_code
25	25%10 = 5	5
27	27%10 = 7	7
46	46%10 = 6	6
70	70%10 = 0	0
89	89%10 = 9	9
31	31%10 = 1	1
22	22%10 = 2	2



Collusion in Hash Table



Let's insert 12 to above hash table.



Collision Resolution Techniques

- There are two basic approaches to solve collisions
  - **Separate chaining (open hashing)**:
    - Collisions are stored outside of the table.
    - Chaining system can be implemented using linked list, binary search tree, hash table etc..
- **Open addressing (closed hashing)**:
  - Collisions are stored at another slot in the table.
  - Linear probing
  - Quadratic probing
  - Double hashing

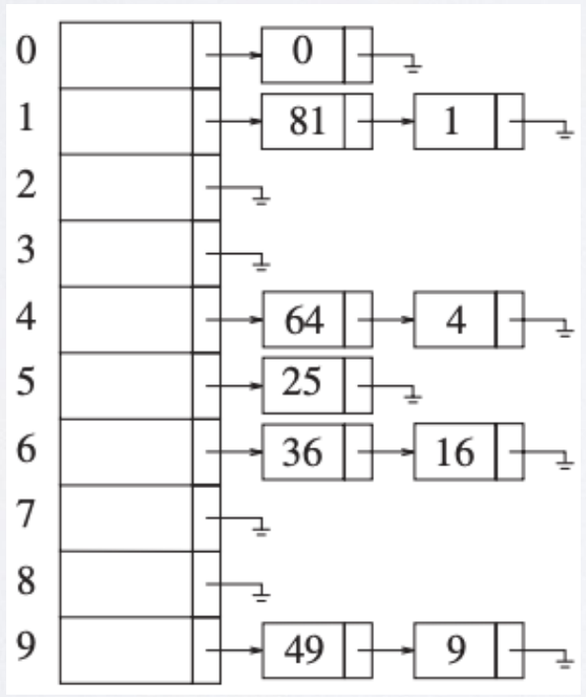
Separate Chaining (Open Hashing)

- *hash(x)* = *x* mod 10

- To perform a **search**, we use the hash function to determine which list to traverse. We then search the appropriate list.

- To perform an **insert**, we check the appropriate list to see whether the element is already in place (if **duplicates** are expected, an **extra data** member is usually kept, and this data member would be incremented in the event of a match).

- If the element turns out to be new, it can be inserted at the front of the list, since it is convenient and also because frequently it happens that recently inserted elements are the most likely to be accessed in the near future.



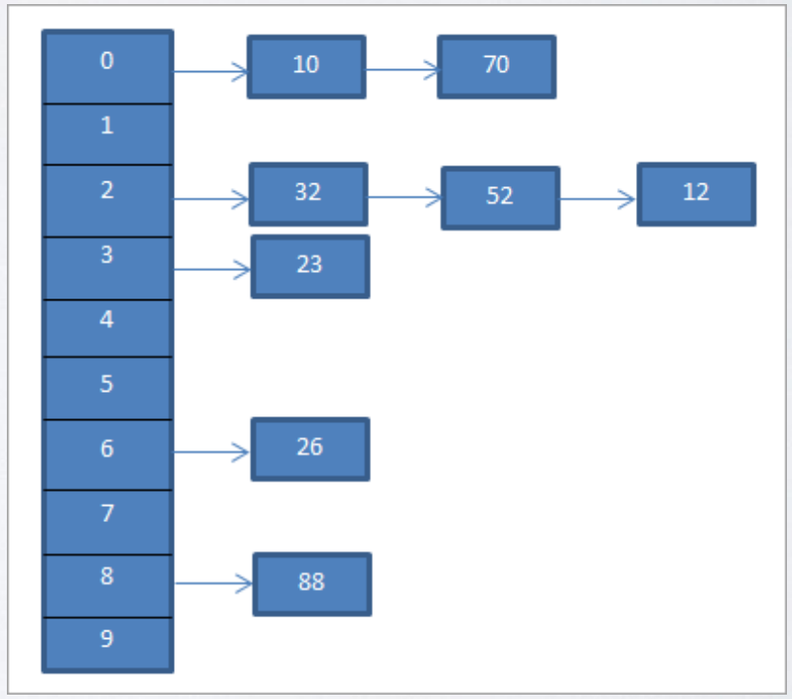


Figure: Sample separate chaining hash tables

Separate Chaining (Open Hashing)

- If the **keys are uniformly distributed** across the hash table then the **average cost** of looking up for the particular key depends on the **average number of keys** in that linked list. Thus separate chaining remains effective even when there is an increase in the number of entries than the slots.
    - For insert, find, and remove operations: **O(1+n/tableSize)**.
- The **worst-case** for separate chaining is when all the keys equate to the same hash code and thus are **inserted in one linked list** only. Hence, we need to look up for all the entries in the hash table and the cost which are proportional to the number of keys in the table.
    - For insert, find, and remove operations: **O(n)**.

Open Addressing (Closed Hashing)

- An alternative way to resolve collusions is to **find alternative cells until an empty cell is found**.
- More formally, cells $h_0(x)$, $h_1(x)$, $h_2(x)$ are tried in succession, where **$h_i(x)$ = (hash(x) + f(i)) mod tableSize**, where f(0) = 0.
- Function ***f*** is the **collusion resolution strategy**.
- There are three basic collusion resolution strategies.
    - Linear probing
    - Quadratic probing
    - Double hashing

Linear Probing

- In linear probing, *f* is a linear function of i, typically *f*(i) = i.

hash = key % tableSize , if occupied, then

hash = (key + 1) % tableSize , if occupied, then

hash = (key + 2) % tableSize , if occupied, then

hash = (key + 3) % tableSize , if occupied, then

.

.

.

hash = (key + tableSize - 1) % tableSize, if occupied, then

hash table is full

Linear Probing

- Let's insert keys {10, 70} into a hash table using the hash function, *hash(x)* = *x* mod 10, and the collision resolution strategy, *f*(i) = i.

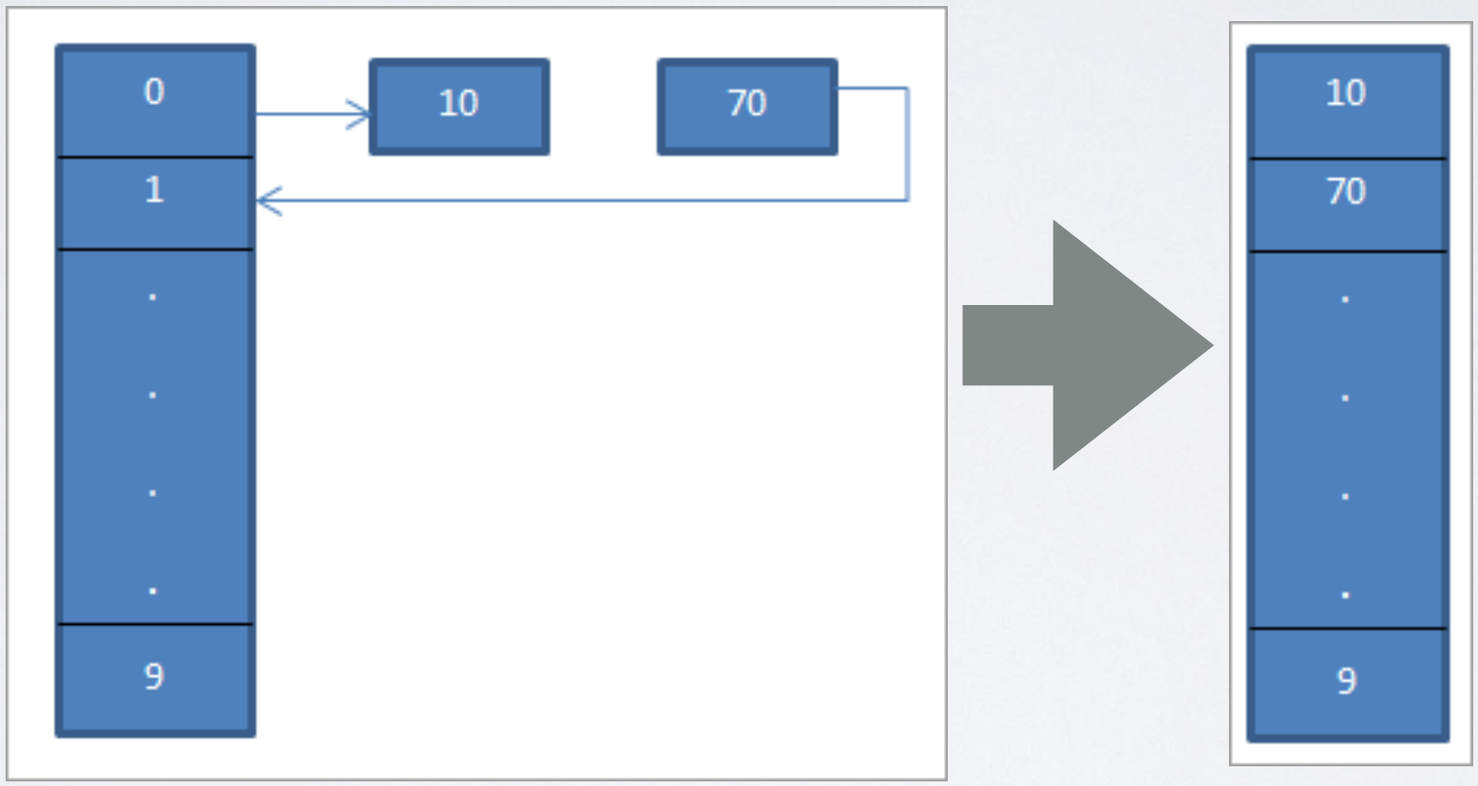


Figure: Hash table with linear probing, after inserting 10 and 70.

Linear Probing

- Let's insert keys {89, 18, 49, 58, 69} into a hash table using the hash function, *hash(x)* = *x* mod 10, and the collision resolution strategy, *f*(i) = i.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Figure: Hash table with linear probing, after each insertion.

Linear Probing

- Linear probing may suffer from the problem of **primary clustering** in which there is a chance that the consecutive cells may get occupied and the probability of inserting a new element gets reduced.
- In another words, any key that hashes into the cluster will require several attempts to resolve collusion before it will be added.
- As in other hash tables, worst case running time is O(n), where n is the number of elements in the table.

Quadratic Probing

- Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing.
- In quadratic probing, as the name suggests, the collision function is quadratic.
- The popular choice is $f(i) = i^2$.

hash = key % tableSize , if occupied, then

hash = (key + 1^2) % tableSize , if occupied, then

hash = (key + 2^2) % tableSize , if occupied, then

hash = (key + 3^2) % tableSize , if occupied, then

.

.

.

until finding an empty location, or detecting a loop

Quadratic Probing

- No guarantee that an empty cell will be found if table is more than half full, or even before that if table size is not prime (loop).
- Elements that hash to the same position will probe to the same alternative cells, which is known as **secondary clustering**.

Double Hashing

- The last collision resolution method we will examine is **double hashing**.
- For double hashing, one popular collusion resolution strategy is $f(i) = i*hash_2(x)$.

$h_i(x) = (hash_1(x) + i*hash_2(x))$ mod tableSize,

where

$hash_1(x) = x$ mod tableSize

$hash_2(x) = R - (x$ mod $R)$, where R is a prime number smaller than tableSize.

Double Hashing

- $h_i(x) = (x + i*(7 - (x \bmod 7))) \bmod 10$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Figure: Hash table with double hashing, after each insertion.

Rehashing

- If the table gets too full, the running time for the operations will start taking too long.
- Insertions might fail for open addressing hashing with quadratic resolution (this can happen if there are too many removals intermixed with insertions).
- A solution, then, is to build another table that is about twice as big (with an associated new hash function) and scan down the entire original hash table, computing the new hash value for each (non deleted) element and inserting it in the new table.

Rehashing

- Suppose the elements 13, 15, 24, and 6 are inserted into a linear probing hash table of size 7. The hash function is h(x) = x mod 7.
- If 23 is inserted into the table, the resulting table will be over 70 percent full.
- Since the table is so full, a new table is created. The size of this table is 17, because this is the first prime that is twice as large as the old table size.
- The new hash function is then h(x) = x mod 17.

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

0	6
1	15
2	
3	24
4	
5	
6	13

0	6
1	15
2	23
3	24
4	
5	
6	13

Rehashing

- This entire operation is called rehashing.
- This is obviously a very expensive operation; the running time is **O(n)**, since there are n elements to rehash and the table size is roughly 2n, but it is actually not all that bad, because it happens very **infrequently**.
- In particular, there must have been n/2 insertions prior to the last rehash, so it essentially adds a constant cost to each insertion.

Deletion

- What if we delete 89, and then search for 49?

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

Example

- Let's insert keys {13, 27, 38, 49, 15, 73, 19} into a hash table with size 11 using the hash function *hash(x)* = *x* mod 11, and
  - collision resolution strategy, *f*(i) = i.
  - collision resolution strategy, *f*(i) = i^2.
  - collusion resolution strategy, *f*(i) = i*(7 - (x mod 7))