Data Structure & Algorithms

Lec 18: Hashing

Fall 2017 - University of Windsor
Dr. Sherif Saad

Outlines

- Hashing Motivation
- Hashtable Implementation
- Hash Functions
- Collision Resolution Method

Associative Array: Motivation

An associative array, map, symbol table, or dictionary is an abstract data type composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.

Common Operations are:

- Insert a pair to the collection
- Delete a pair from the collection
- Update an existing pair
- Lookup of a value associated with a particular key

Associative Array: Example

What are the possible implementation methods for associative array?

- Arrays
- LinkedList
- Binary Search Tree
- Hashing

Key	Value
reddit.com	72.247.244.88
google.com	172.217.11.174
youtube.com	74.125.65.91
hotmail.com	65.55.72.135
digg.com	64.191.203.30
bing.com	65.55.175.254

Hashing: Definition

Hashing is a technique used for storing and retrieving information as quickly as possible.

In general, a balanced binary tree we can insert, delete, search the data in O(log n).

What if we want to insert, delete, search to O(1), What does O(1) mean?

Hashing: Components

Any hashing technique or system has the following components:

1. Hash Table
2. Hash Function
3. Collision Resolution Method

Hash Table

Is a data structure which implements an associative array

A hash table uses a ***hash function to compute an index into an array*** of buckets or slots, from which the desired value can be found.

With an array, we store the record whose key is k at position k of the array.

Given a key k, we find the element whose key is k by just looking in the kth position of the array (direct addressing)

Note: direct addressing is possible when we could allocate an array with one position for every possible key

Hash Table

```
1. #define MAX 20
2. typedef struct Record{
3.     string  key;
4.     Data data;
5. } Record;
6. Record websites[MAX];
```

0	reddit.com 72.247.244.88
1	google.com 172.217.11.174
2	Youtube.com 74.125.65.91
3	Hotmail.com 65.55.72.135
4	Digg.com 64.191.203.30
5	Bing.com 65.55.175.254

Hash Function

The hash function is used to transfer a key into an array index.

The hash function should map every unique key into a unique slot (bucket) index (perfect hash function) but it is difficult to design a hash function that guarantee that.

A collision is a condition that occurs when the hash function can not map every unique key into a unique slot index. This means two different records with two different keys are mapped to the same index.

Hash Function

How to Choose Hash Function?

- Minimum number of collisions
- Distributes the keys evenly across the table.
- Calculation cost when mapping the key into index is minimum
- Use all the information provided in the key
- Have a high load factor for a given set of keys.

Hash Function: Example 1

hash(k)

return k mod n

where k is the key and n is the size of the table

```
1. int Hash (char *key, int n){
2.     int k = ConvertToInt(key);
3.     return k % n;
4. }
```

Hash Function Example 2

Use all letters of key

h(k) = (sum of ASCII values in Key) mod m

So,

$$h(k) = \left(\sum_{i=0}^{keysize-1} (int)k[i] \right) \bmod m$$

Hash Function Example 3

Use first three letters of a key & multiplier

```
h( k ) =
    ( (int) k[0] +
      (int) k[1] * 27 +
      (int) k[2] * 729 ) mod m
```

Note: 27 is number of letters in English + space 729 is 27^2 Using 3 letters, so 26^3 = 17, 576 possible keys

Load Factor

The load factor of a non-empty hash table is the number of items stored in the table divided by the size of the table.

$$loadfactor = \frac{M}{N}$$

Describes the amount of capacity which is to be exhausted for the Hashtable to increase its capacity.

For example in Java, ***HashMap has a lost factor of 0.75. 25%*** of the buckets will be free before there is an increase in the capacity.

Lower load factor → more free slots →less collision → high performance → high space requirement

Collision Resolution Methods

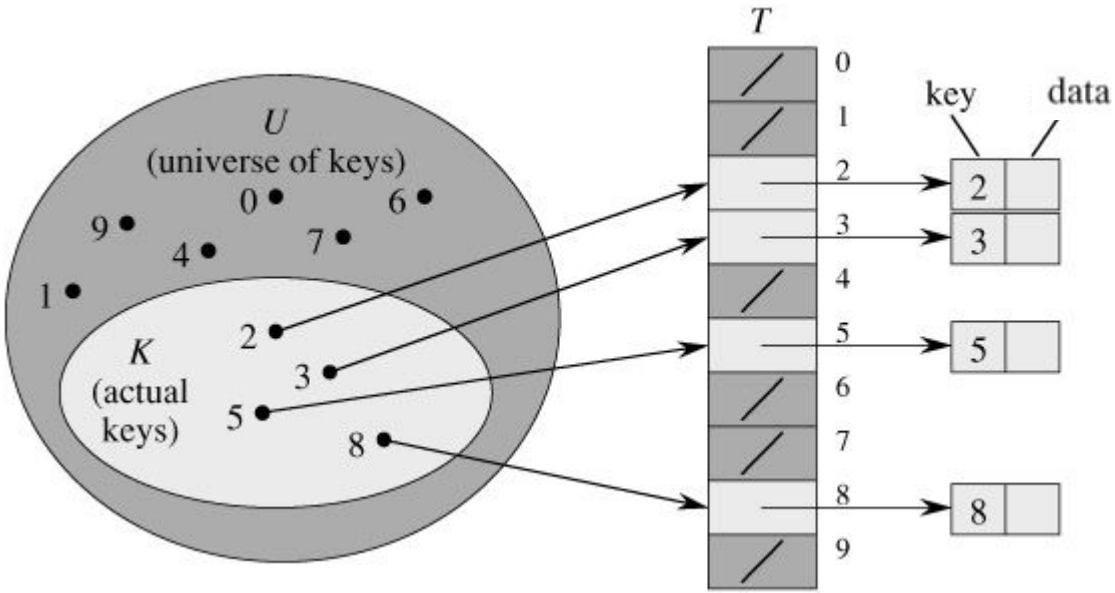
Collision is the state in which two different keys produce the same address. Collision needs to resolved. Chaining and open addressing are the basic methods of resolving collision.

Direct Chaining: An array of linked list application (Separate Chaining)

Open Addressing: Array-based implementation

- Linear Probing
- Quadratic Probing
- Double Hashing

Collision & Hash Table



Separate Chaining

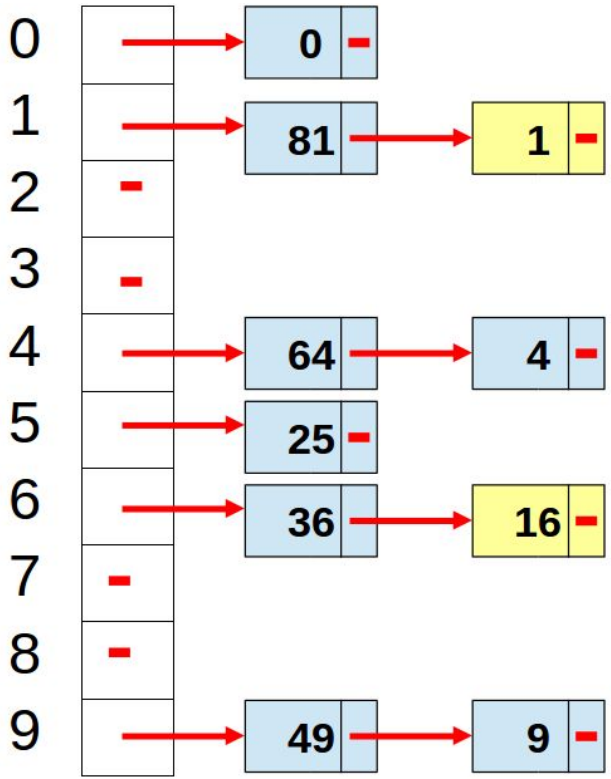
The hash table is an array of linked lists

Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

As before, elements would be associated with the keys

We are using the hash function:

h(k) = k mod n



Separate Chaining Operations

```
1. Chained-Hash-Insert( T, x )
2.    insert x at the head of list T[ h( key[x] ) ]
3.
4. Chained-Hash-Search( T, k )
5.    search for an element with key k
6.    in list T[ h(k) ]
7.
8. Chained-Hash-Delete( T, x )
9.    delete x from the list T[ h( key[x] ) ]
```

Open Addressing

All records are stored in the hash table itself (the array). If a collision occurs, try alternate cells until empty cell is found.

Three main techniques:

1. Linear Probing.
2. Quadratic Probing.
3. Double Hashing.

Open Addressing: Linear Probing

It was invented in 1954 by Gene Amdahl and colleague and first analyzed in 1963 by Donald Knuth.

Each slot of a hash table stores a single key–value pair.

When the hash function causes a collision, linear probing searches the table for the closest following free location and inserts the new key there.

It sensitive to the ***quality of its hash function*** than some other collision resolution schemes

Open Addressing: Linear Probing

Linear Probing is easy to implement, but it suffers from the problem of primary clustering. Where one part of the table become more dense.

Hashing several times in one area results in a cluster of occupied spaces in that area. Long runs of occupied spaces build up and the average search time increases.

Open Addressing: Linear Probing Operations

Searching: To search for a given key k the slots of hash table are examined, beginning with the cell at index hash(k) and continuing to the adjacent cells hash(x) + 1, hash(x) + 2, ..., until finding either an empty cell or a cell whose stored key is x.

What about insert and delete?

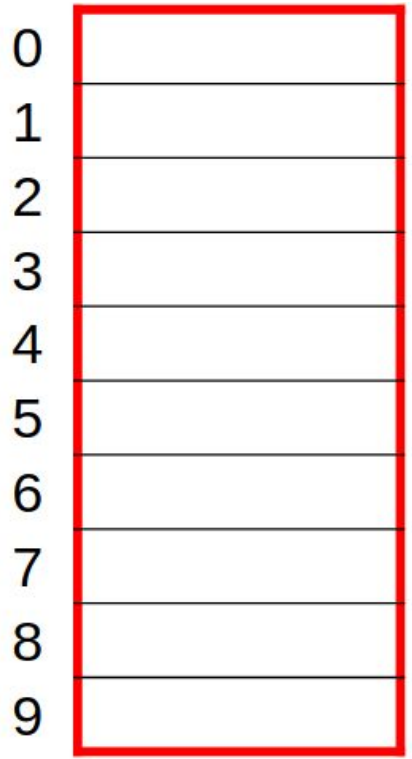
Open Addressing: Linear Probing

What does the hash table looks like after the following insertions?

Insert Keys: 0, 1, 4, 9, 16, 25, 36, 49, 64, 81

We are using the hash function:

h(k) = k mod n



Open Addressing: Linear Probing

```
1. LinearProbing-Search( T, k, N )
2.     i ← 0
3.     repeat
4.         j ← h( k, i )
5.         if T[ j ] = k
6.             then return j
7.         i ← i + 1
8.     until T[ j ] = NIL or i = N
9.     return NIL
```

Open Addressing: Linear Probing

```
1. LinearProbSearch( T, k )
2.    i ← 0
3.       j ← hash( k )
4.    repeat
5.       if T[ j ] = k
6.          then return j
7.       i ← i + 1
8.    until T[ j ] = NIL or i = m
9.    return NIL
```

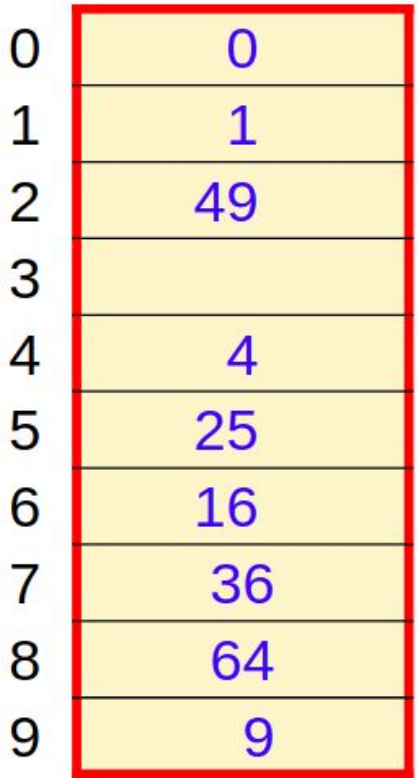
Open Addressing: Linear Probing

How do we delete 9?

How do we find 49 after deleting 9?

We are using the hash function:

h(k) = k mod n



Open Addressing: Quadratic Probing

For a given hash value, the indices generated by linear probing are as follows:

$$H+1, H+2, H+3, H+4, \ldots, H+k$$

As we mentioned this results in primary clustering, where the search becomes less efficient.

An example sequence using quadratic probing is:

$$H+1^2, H+2^2, H+3^2, H+4^2, \ldots, H+k^2$$

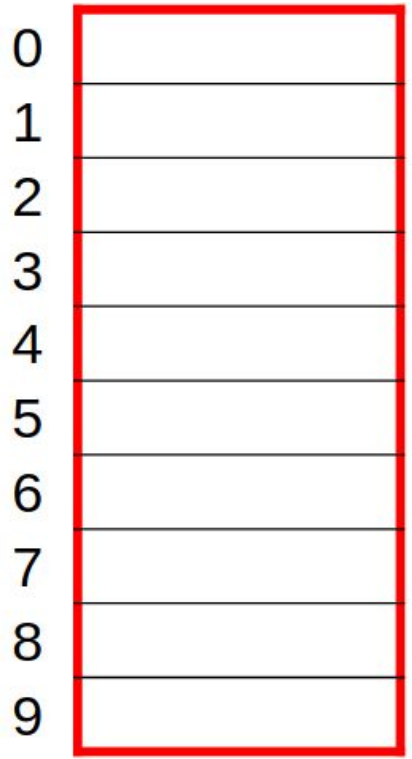
Open Addressing: Quadratic Probing

What does the hash table looks like after the following insertions?

Insert Keys: 10, 23, 14, 9, 16, 25, 36, 44, 33

We are using the hash function:

$$h(k) = k \bmod n$$



Open Addressing: Double Hashing

If a collision occurs when inserting, apply a second auxiliary hash function, h2(k), and probe at a distance.

In order for the entire table to be searched, the value of the second hash function, h2(k), must be relatively prime to the table size n.

One of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations

Open Addressing: Double Hashing

The interval between probes is computed by a second hash functions:

hash2(key) != 0 and hash2() != hash1()

We first probe the position with hash1(key), if the position is occupied, we probe for new position using hash2(key) as following

((hash1(key) + i * hash2(key)) mod M)

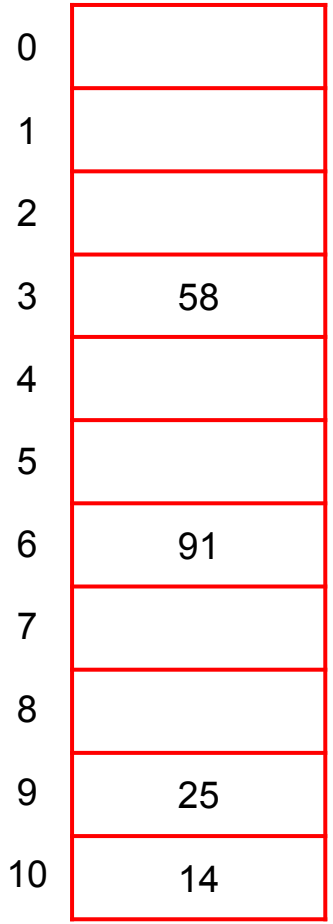
Open Addressing: Double Hashing

hash1(key) = key mod 11

hash2(key) = 7 - (key mod 7)

Insert keys: 58, 14, 91, 25

- 58 mod 11 = 3
- 14 mod 11 = 3 → (h1(14) + h2(14)) mod 11= 3+ 7 - 14 mod 7 = 10
- 91 mod 11 = 3 → 3 + 7, (3 + 2 *7) mod 11 = 6
- 25 mod 11 = 3 → 3 + 3, (3+ 2 * 3) mod 11 = 9



Collision Resolution Method-Design Consideration

1. Are we guaranteed to find an empty cell if there is one?
2. Are we guaranteed we would not be checking the same cell twice during one insertion?
3. What should the load factor be to obtain O(1) average-case insert, search, and delete?

Rehashing

Rehashing is required When the table gets too full, the average search time deteriorates from O(1) to O(n).

Create a larger table and then rehash all the elements into the new table