Hashing

Joy Mukherjee

Searching vs. Hashing

- Linear Search: $O(n)$
- Binary Search: $O(\log_2 n)$
- Searching: Comparing a given key with existing key values in the array
- Hashing: Location/Address can be calculated from key value

Hash Function

- Suppose, n elements stored in a hash table of size M ($M \geq n$)
- A key k is stored in j-th location of the hash table, if $j = h(k)$
- The function h is called hash function

- Example:
- $h(k) = k \% M$, where $0 \leq h(k) \leq M-1$

Collision & Resolution

- **Collision**: $h(k_1) = h(k_2)$ (Mapping of different keys into same location)

- The keys k1 and k2 are called synonyms.

- **Collision Resolution Strategy**: How to resolve address / location conflict? (Where do we insert k_2 when the location $h(k_2)$ is occupied?)

Challenges

- Design of hash function that minimize collisions

- Design of efficient collision resolution strategy that locate a vacant position once a collision occurs.

- Ideal hash function distributes the keys over the range [0, M-1], where M is the size of the hash table.

- Evaluation of a hash function should also be efficient.

- The probability of generation of address x is 1/M.

Design of Hash Functions

Design of Hash Functions

- Division Method
- Mid-Square Method
- Folding

The Division Method

- **$h(k) = k \% M$, where $0 \leq h(k) \leq M-1$**
- How to choose M? (**Open Question**)
- Prime number that is not related to the base of the number system by some simple relation may be a good choice.

- **Example**: 4999, 7001, or 7999 are bad choices since they are related to 10 by $q.10^p - c$ or $q.10^p + c$
- Use of such prime numbers would have effects on the last p-digits of the key.
- If M = 4999, then 0234, 5233, 10232, 15231 etc. map to the address 234.

The Mid-Square Method

- A p-digit key is squared to have a 2p-digit value
- From middle portion, a q-digit number is chosen as its address.

- **Example**:
- Key 3271 (Here p=4 and q=3)
- Key^2 = 10699441
- 699 or 994 may be used as address.

Folding

- **Goal**: Generate a q-digit address from a p-digit key
- Partition p-digit key into groups of q-digits from the right.
- Sum up the groups
- Select the rightmost q-digits of the sum
- **Example**: p = 8 q = 3
- Key = 12345678
- Partition: 12/345/678
- Sum: 1035 (12 + 345 + 678)
- Address: 035 i.e. 35

Collision Resolution Algorithms

Collision Resolution Algorithms

- **Open Addressing**: If location of k_2 is collided with location of k_1, then an empty location is found out **within the hash table**, and k_2 is inserted in that location.
  - Linear Probe
  - Quadratic Probe
  - Double Hashing
- **Chaining**: **Linked List** is used to store keys. If location of k_2 is collided with location of k_1, then k_2 is inserted somewhere in the linked list at that location.
  - Separate Chaining
  - Coalesced Chaining

Collision Resolution Algorithms

- **Open Addressing**: If location of k_2 is collided with location of k_1, then an empty location is found out **within the hash table**, and k_2 is inserted in that location.
  - Linear Probe
  - Quadratic Probe
  - Double Hashing

Linear Probe

- Size of the hash table is M.
- The key k1 is inserted at h(k1), where h is the hash function.
- Now, the key k2 is map to h(k2) = h(k1)
- It will be placed in the next vacant location in the hash table. It will search (h(k2) + 1)% M, (h(k2) + 2)% M, (h(k2) + 3)% M, and so on.
- This search will continue till the next vacant location is found or the hash table is full.

Linear Probe

- **h(k) = k%7**
- **L = {706, 18, 38, 102, 351, 146, 7}**

Index	Key
0	146
1	351
2	7
3	38
4	18
5	102
6	706

Linear Probe: Insert

```
int insertLinearProbe (int key) {
int i = 0, j = hash(key), k = j;
do {
    if(T[j] == false) { /* No key at A[j] */
        A[j] = key; T[j] = true; return j;
    }
    i = i + 1;
    j = (k + i) % m; // m = size of the hash table
} while(i < m);
return -1; // Hash table is full
}
```

Linear Probe: Search

```
int searchLinearProbe (int key) {
int i = 0, j = hash(key), k = j;
do {
    if(A[j] == key) {
        return j;
    }
    i = i + 1;
    j = (k + i) % m;
} while(T[j]==true && i < m);
return -1;
}
```

Problem of Linear Probing

- Primary Clustering: It would not distribute the keys uniformly in the hash table.

- All keys that map to a location i will be clustered around that location, thereby creating long run of occupied locations.

- Search and Insertion time is increased.

Quadratic Probe

- Size of the hash table is M.
- The key k1 is inserted at h(k1), where h is the hash function.
- Now, the key k2 is map to h(k2) = h(k1)
- It will be placed in the next vacant location in the hash table. It will search (h(k2) + 1)% M, (h(k2) + 4)% M, (h(k2) + 9)% M, and so on.
- This search will continue till the next vacant location is found.

Quadratic Probe

- **h(k) = k%7**
- **L = {706, 18, 38, 102, 32, 146, 7}**

Index	Key
0	146
1	32
2	7
3	38
4	18
5	102
6	706

(7+1)%7 = 1
(7 + 4)%7= 4
(7+9)%7 =2

Quadratic Probe: Insert

```
int insertQuadraticProbe (int key) {
int i = 0, j = hash(key), k = j;
do {
    if(T[j] == false) { /* No key at A[j] */
        A[j] = key; T[j] = true; return j;
    }
    i = i + 1;
    j = (k + i * i) % m;
} while(i < m);
return -1;
}
```

Quadratic Probe: Search

```
int searchQuadraticProbe (int key) {
int i = 0, j = hash(key), k = j;
do {
    if(A[j] == key) {
        return j;
    }
    i = i + 1;
    j = (k + i * i) % m;
} while(T[j]==true && i < m);
return -1;
}
```

Problem of Quadratic Probing

- A key may not be inserted even if the hash table is not full.
- 0 1 2 3 4 M = 5
- k3 k1 k2
- I % M, (I + 1) % M, (I + 4) %M, (I + 9) % M, (I + 16) % M
- The key k4 can’t be placed if h(k4) = 1.
- Note that there are still vacant places in the hash table, yet we are not able to place k4.

Double hashing: Insert

```
int insertDoubleHashing (int key) {
int i = 0, j = h1(key), l = h2(key), k = j;
do {
    if(T[j] == false) { /* No key at A[j] */
        A[j] = key; T[j] = true; return j;
    }
    i = i + 1;
    j = (k + i * l) % m;
} while(i < m);
return -1;
}
```

Double hashing: Search

```
int searchDoubleHashing (int key) {
int i = 0, j = h1(key), l = h2(key), k = j;
do {
    if(A[j] == key) {
        return j;
    }
    i = i + 1;
    j = (k + i * l) % m;
} while(T[j]==true && i < m);
return -1;
}
```

Collision Resolution Algorithms

- **Chaining**: **Linked List** is used to store keys. If location of k_2 is collided with location of k_1, then k_2 is inserted somewhere in the linked list at that location.
  - Separate Chaining (Keys are stored outside the hash table)
  - Coalesced Chaining (Keys are stored inside the hash table)

Problem of Open Addressing

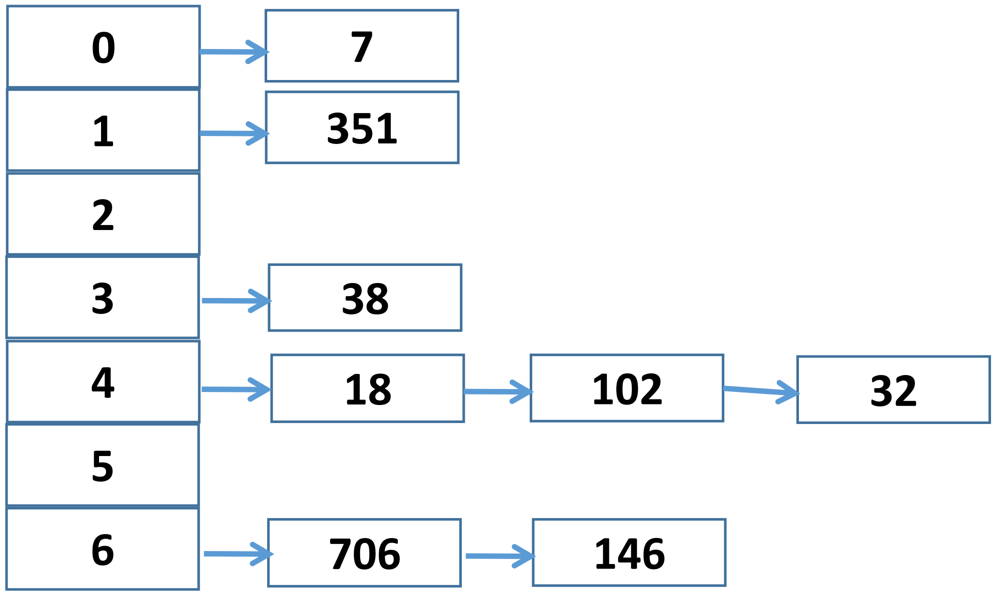
- Deletion of a key is not straight-forward
- When a key is deleted,

  - Option 1: The corresponding entry in the hash table is marked with a special symbol which leads to an inefficient search.

  - Option 2: Pack the synonyms in the hash table through shifting which leads to an inefficient delete. (Compaction)
  - 10 20 30 40 are synonyms placed at location 1 3 5 7 respectively. Delete 20. 10 30 40 are synonyms placed at location 1 3 5 respectively

Separate Chaining

- An array A of linked list is maintained
- A[i] is initially NULL
- If **h(k) = k % M, where 0 ≤ h(k) ≤ M-1,** then k is inserted in the linked list pointed by A[h(k)]

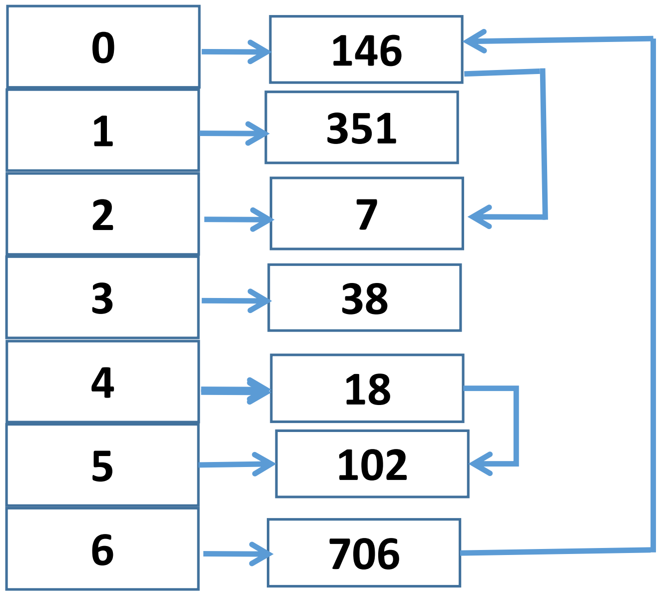


h(k) = k%7

L = {706, 18, 38, 102, 351, 146, 7, 32}

Coalesced Chaining

- Similar to linear probing
- All keys that map to the same address are linked together
- In coalesced chaining, chain consists of keys hashed to the same address and keys placed through linear probing



h(k) = k%7

L = {706, 18, 38, 102, 351, 146, 7}

Coalesced Chaining

Index	Key	Next Index
0	146	2
1	351	-1
2	7	-1
3	38	-1
4	18	5
5	102	-1
6	706	0

h(k) = k%7

L = {706, 18, 38, 102, 351, 146, 7}