

# Lecture 27

## **Universal Hash Function**

### &

## Collision Resolution

*November 16, 2021*  
*Tuesday*

# UNIVERSAL HASHING

- When very little information is known about keys.
- Pick a hash function randomly in a way that is independent of the keys
  - Guarantees good performance on average, without depending upon keys.
  - Need a family of hash functions to choose from.

# UNIVERSAL HASH FUNCTIONS

- A class of functions is Universal when for any sample,
- A randomly chosen member of that class will be expected to distribute the sample evenly,
- Whereby members of that class guarantee low probability of Collisions (Carter & Wegman 1979)

# UNIVERSAL HASHING

- Let  $\mathbf{H}$  be a (finite) collection of hash functions
  - ... that map a given universe  $\mathbf{U}$  of keys ...
  - ... into the range  $\{0, 1, \dots, m - 1\}$ .
- $\mathbf{H}$  is said to be **Universal** if
  - for each pair of distinct keys  $x, y \in \mathbf{U}$ , the number of hash functions  $h$  in  $\mathbf{H}$  for which  $h(x) = h(y)$  equals  $|\mathbf{H}| / m$ .

# UNIVERSAL HASHING

- In other words, if no pair of distinct keys are mapped into the same index by a randomly chosen function  $h$  with a probability  $= 1/m$ .
- In Simpler words, there is only one possibility of TSize ( $m$ ) that two keys collide

# UNIVERSAL HASHING

- We begin by choosing a prime  $p$  large enough so that every possible key  $k$  is in the range  $0$  to  $p - 1$ .
- $Z_p$  denotes the set  $\{ 0, 1, \dots, p - 1 \}$ ,
- And  $Z_p^*$  let denote the set  $\{ 1, 2, \dots, p - 1 \}$ .
- We can define the hash function  $h_{a,b}$  for any  $a \in Z_p^*$  and any  $b \in Z_p$

$$h_{a,b}(k) = ( (ak + b) \bmod p ) \bmod m$$

# Lecture 27

## Universal Hash Function & **Collision Resolution**

*November 16, 2021*  
*Tuesday*

# COLLISION RESOLUTION

- Straight forward hashing is not without its problems.
- Almost all hash functions can assign one position to more than one keys.
- Consider  $h_1(\text{name}) = \text{name}[0]$  takes the ASCII value of the first character of the name. (TSize = 26 for all alphabets).
  - Almost all names starting with same alphabet will be hashed to same position.
- We can solve this problem by replacing the hash function with more sophisticated one, which distributes names more uniformly in the table.



# COLLISION RESOLUTION

- Consider  $h_2(\text{name}) = \text{name}[0] + \text{name}[1]$ .
  - Adds the first two letters of the name, will be better than  $h_1$
- The possibility of hashing different names to the same location still exists
- Consider  $h_3(\text{name}) = \text{name}[0] + \dots + \text{name}[\text{strlen}(\text{name}) - 1]$ 
  - Adds all the alphabet of the name
  - Better than  $h_1$  and  $h_2$ .
  - However, the table size needs to be increased from 26 to more in this case.

# COLLISION RESOLUTION

These two factors **Hash Function** and **Table Size** can minimize the number of Collisions But they cannot completely eliminate them.

# COLLISION RESOLUTION

Collision resolution techniques are divided into two broad categories

- Close Hashing (Open Addressing)
  - Linear Probing
  - Quadratic Probing
  - Double Hashing
  - Bucket Addressing
- Open Hashing (Close Addressing)
  - Chaining

# LINEAR PROBING

- When a key collides with another key, the collision occurs.
- The collision is resolved by finding an available table entry other than the position (address) to which the colliding key is originally hashed.
- If the position  $h(k)$  is occupied then, following probing sequence is tried.
  - Until an available cell is found **Or**
  - The same positions are tried repeatedly **Or**
  - The table is full.

$norm(h(K) + p(1)), norm(h(K) + p(2)), \dots, norm(h(K) + p(i)), \dots$

# LINEAR PROBING

- Function  $p$  is probing function,  $i$  is a probe and  $\text{norm}$  is a normalization function (usually the modulo the size of the table).
- In linear probing, a sequential search is performed to find the next available empty cell to store the current key.
- If the end of the table is reached and no empty cell has been found, the search is continued from the beginning of the table and stops—in the extreme case—in the cell preceding the one from which the search started.

Insert:  $A_5, A_2, A_3$

0	
1	
2	$A_2$
3	$A_3$
4	
5	$A_5$
6	
7	
8	
9	

(a)

$B_5, A_9, B_2$

0	
1	
2	$A_2$
3	$A_3$
4	$B_2$
5	$A_5$
6	$B_5$
7	
8	
9	$A_9$

(b)

$B_9, C_2$

0	$B_9$
1	
2	$A_2$
3	$A_3$
4	$B_2$
5	$A_5$
6	$B_5$
7	$C_2$
8	
9	$A_9$

(c)

# LINEAR PROBING | CONSTRAINTS

- Linear probing has the tendency to create Clusters. (These Clusters are known as **Primary Clusters**)
- The empty cells following clusters have a much greater chance to be filled than other positions.
  - This probability is equal to  $(\text{sizeof}(\text{cluster}) + 1) / \text{TSize}$ .
  - Other empty cells have only  $1 / \text{TSize}$  chance of being filled.
- If a cluster is created, it has a tendency to grow, and the larger a cluster becomes, the larger the likelihood that it will become even larger.

# QUADRATIC PROBING

- The cluster created by Linear Probing undermines the performance of hash table for storing and retrieving data.
  - The idea is to avoid Clusters.
- To achieve this goal probing function needs to be chosen carefully. Quadratic Probing is one such choice.

$$p(i) = h(K) + (-1)^{i-1}((i+1)/2)^2 \text{ for } i = 1, 2, \dots, TSize - 1$$



# QUADRATIC PROBING

- This probing formula can also be represented as

$$h(K) + i^2, h(K) - i^2 \text{ for } i = 1, 2, \dots, (TSize - 1)/2$$

- If a collision occurs at  $h(K)$  then following sequence will be tried, the result will be divided by the modulo  $TSize$ .

$$h(K), h(K) + 1, h(K) - 1, h(K) + 4, h(K) - 4, \dots, h(K) + (TSize - 1)^2/4,$$

$$h(K) - (TSize - 1)^2/4$$

# QUADRATIC PROBING

- This probing formula can also be represented as

$$h(K) + i^2, h(K) - i^2 \text{ for } i = 1, 2, \dots, (TSize - 1)/2$$

- If a collision occurs at  $h(K)$  then following sequence will be tried, the result will be divided by the modulo  $TSize$ .

$$h(K), h(K) + 1, h(K) - 1, h(K) + 4, h(K) - 4, \dots, h(K) + (TSize - 1)^2/4,$$

$$h(K) - (TSize - 1)^2/4$$

# QUADRATIC PROBING

- The size of the table should not be even number,
  - Then only even positions or odd positions will be tried.
- Ideally, the table size should be a prime  $4j + 3$  for an integer  $j$ 
  - Consider  $j = 4$ , then,  $TSize = 19$ , and assuming  $h(K) = 9$   
9, 10, 8, 13, 5, 18, 0, 6, 12, 15, 3, 7, 11, 1, 17, 16, 2, 14, 4

Insert:  $A_5, A_2, A_3$

0	
1	
2	$A_2$
3	$A_3$
4	
5	$A_5$
6	
7	
8	
9	

(a)

$B_5, A_9, B_2$

0	
1	$B_2$
2	$A_2$
3	$A_3$
4	
5	$A_5$
6	$B_5$
7	
8	
9	$A_9$

(b)

$B_9, C_2$

0	$B_9$
1	$B_2$
2	$A_2$
3	$A_3$
4	
5	$A_5$
6	$B_5$
7	
8	$C_2$
9	$A_9$

(c)

# QUADRATIC PROBING

- Quadratic probe gives much better results than linear probe.
  - the problem of cluster buildup is not avoided altogether.
  - Because for keys hashed to the same location, the same probe sequence is used.
    - Such clusters are called secondary clusters
    - These are less harmful than primary clusters.
- Another option is to have  $p$  be a random number generator.

# DOUBLE HASHING

- The problem of Secondary clustering is best addressed with Double Hashing.
- Two hash functions are used,
  - One for accessing the primary position of a key,  $h$
  - Second function,  $h_p$ , for resolving conflicts.

$$h(K), h(K) + h_p(K), \dots, h(K) + i \cdot h_p(K), \dots$$

# DOUBLE HASHING

- The table size should be a prime number so that each position in the table can be included in the sequence.
- Experiments indicate that secondary clustering is generally eliminated because the sequence depends on the values of  $h_p$ , which, in turn, depend on the key
- Therefore, if the key  $K_1$  is hashed to the position  $j$ , the probing sequence is

$$j, j + h_p(K_1), j + 2 \cdot h_p(K_1), \dots$$

# DOUBLE HASHING

- If another key  $K_2$  is hashed to  $j + h_p(K_1)$ ,
  - then the next position tried is  $j + h_p(K_1) + h_p(K_2)$ ,
  - NOT  $j + 2 \cdot h_p(K_1)$ , which avoids secondary clustering if  $h_p$  is carefully chosen.
- Also, even if  $K_1$  and  $K_2$  are hashed primarily to the same position  $j$ , the probing sequences can be different for each



# DOUBLE HASHING

- This also depends on the choice of  $h_p$ 
  - Therefore the second hash function can be defined in terms of the first one
    - $h_p(K) = i \cdot h(K) + 1$
    - The probing sequence for  $K_1$  is
      - $j, 2j + 1, 5j + 2$
    - Modulo TSize.

# BUCKET ADDRESSING

- Another solution to the collision problem is to store colliding elements in the same position in the table
- This can be achieved by associating a bucket with each address.
  - A bucket is a block of space large enough to store multiple items.
  - The possibility of collisions is not totally avoided.

# BUCKET ADDRESSING

- If a bucket is already full, then an item hashed to it has to be stored somewhere else.
- Using Linear Probing the colliding item can be stored in the next available bucket.
- Or in any other bucket if using Quadratic Probing.
- The colliding items can also be stored in an overflow area.
  - In this case, each bucket includes a field that indicates whether the search should be continued in this area or not.
    - It can be simply a yes/no marker.

Insert:  $A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2$

0		
1		
2	$A_2$	$B_2$
3	$A_3$	$C_2$
4		
5	$A_5$	$B_5$
6		
7		
8		
9	$A_9$	$B_9$

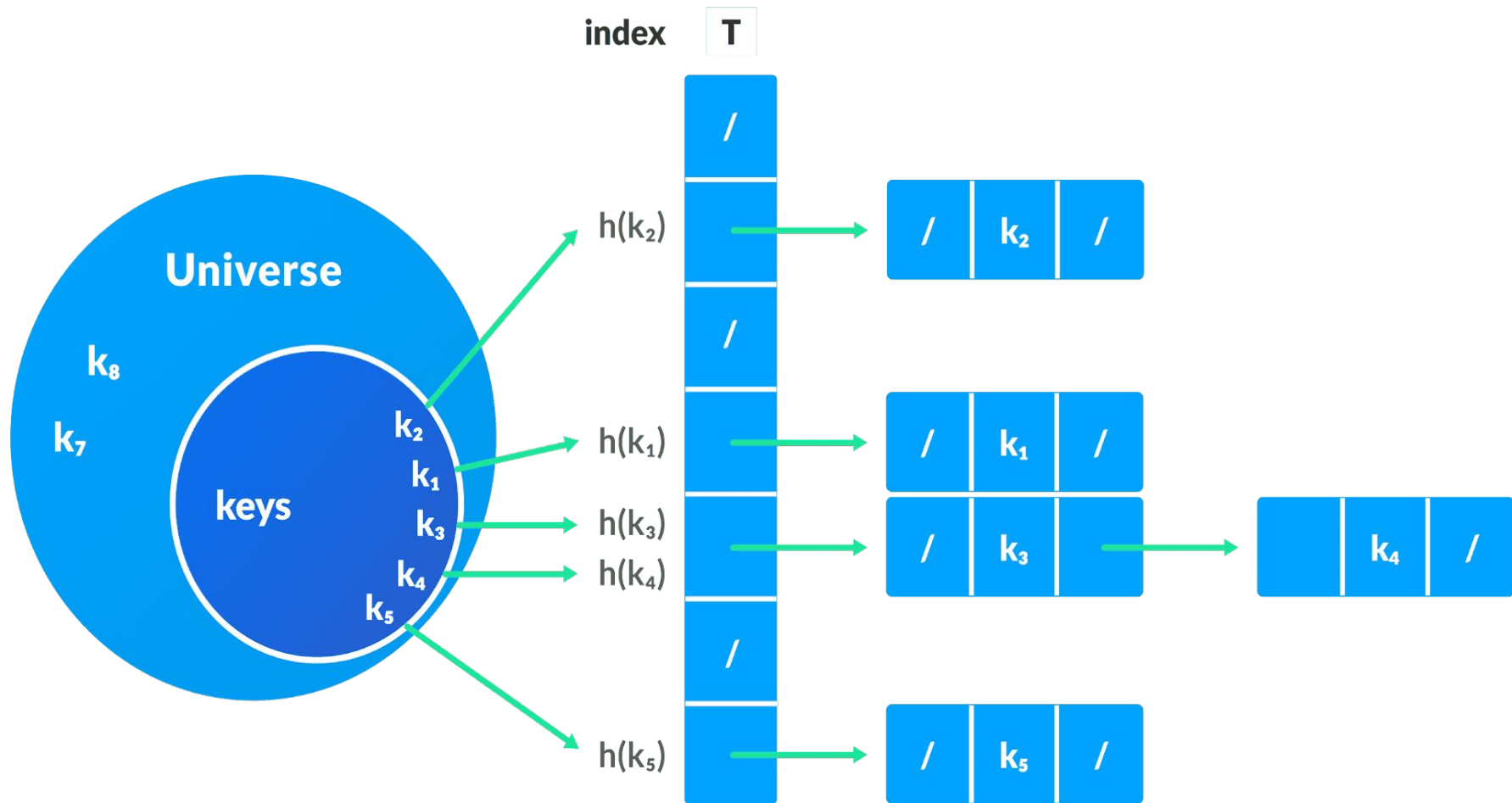
0			
1			
2	$A_2$	$B_2$	
3	$A_3$		
4			
5	$A_5$	$B_5$	
6			
7			
8			
9	$A_9$	$B_9$	

$C_2$

$\vdots$

# CHAINING

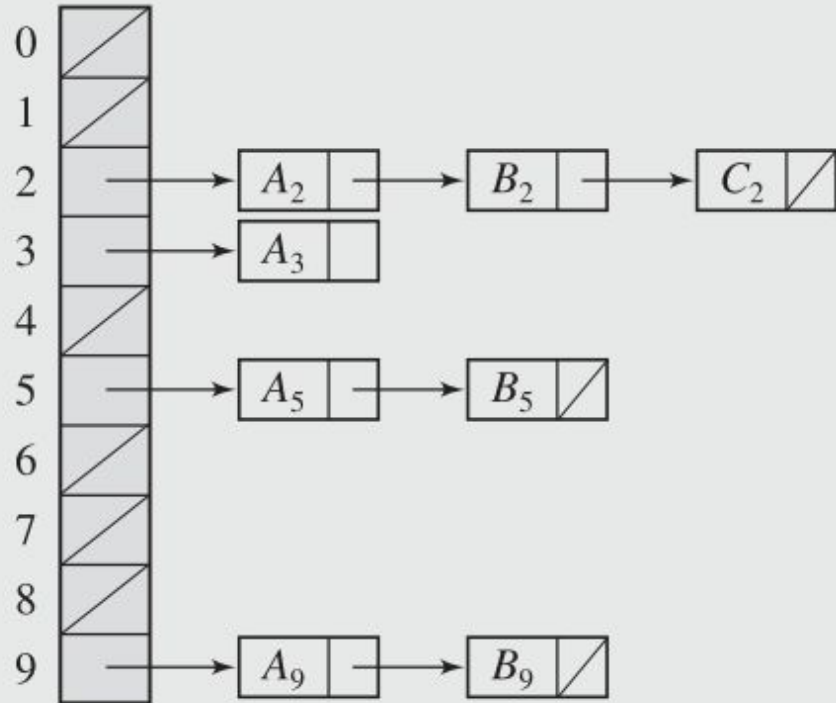
- Each position of the table is associated with a linked list ( Chain Structure).
- Each Node info field now can store keys or references to keys.
- This method is known as **Separate Chaining** and table of references (pointers )is called **Scatter Table**.
- This way is highly unlikely for Table to overflow
  - Cause linked list are are extended only upon the arrival of new keys.
- For short linked list it is very fast, but for longer may not be efficient.



# CHAINING

- Performance can be maintained on these lists by avoiding exhaustive search in case of unsuccessful search.
- Exhaustive search is usually not required in self organizing linked list.

Insert:  $A_5, A_2, A_3, B_5, A_9, B_2, B_9, C_2$



# CHAINING | CONSTRAINTS

- The table stores only pointers, and each node requires one pointer field.
- Therefore, for  $n$  keys,  $n + TSize$  pointers are needed, which for large  $n$  can be a very demanding requirement.



# COALESCED CHAINING

- Combination of Linear Probing & Chaining.
- The first available position is found for a key colliding with another key, and the index of this position is stored with the key already in the table.
- Sequential Search of linear probing is avoided by directly accessing the next element on the linked list.
- Each position pos of the table stores an object with two members: info for a key and next with the index of the next key that is hashed to pos.

# COALESCED CHAINING

- Available positions can be marked by, say,  $-2$  in next;  $-1$  can be used to indicate the end of a chain.
- Requires,  $TSize \cdot \text{sizeof}(\text{next})$  more space for the table in addition to the space required for the keys.
- This is less than Chaining, but the table size limits the number of keys that can be hashed.
- An overflow area known as a cellar can be allocated to store keys for which there is no room in the table. This area should be located dynamically if implemented as a list of arrays.


Insert:  $A_5, A_2, A_3$

0		
1		
2	$A_2$	
3	$A_3$	
4		
5	$A_5$	
6		
7		
8		
9		

(a)

$B_5, A_9, B_2$


0		
1		
2	$A_2$	
3	$A_3$	
4		
5	$A_5$	
6		
7	$B_2$	
8	$A_9$	
9	$B_5$	



(b)

$B_9, C_2$

0		
1		
2	$A_2$	
3	$A_3$	
4	$C_2$	
5	$A_5$	
6	$B_9$	
7	$B_2$	
8	$A_9$	
9	$B_5$	



(c)


Insert:  $A_5, A_2, A_3$

0		
1		
2	$A_2$	
3	$A_3$	
4		
5	$A_5$	
6		
7		
8		
9		
10		
11		
12		

(a)

$B_5, A_9, B_2$


0		
1		
2	$A_2$	
3	$A_3$	
4		
5	$A_5$	
6		
7		
8		
9	$A_9$	
10		
11	$B_2$	
12	$B_5$	



(b)

$B_9, C_2$

0		
1		
2	$A_2$	
3	$A_3$	
4		
5	$A_5$	
6		
7		
8	$C_2$	
9	$A_9$	
10	$B_9$	
11	$B_2$	
12	$B_5$	



(c)