HASHING TECHNIQUES

CSC 212

Hashing Techniques

- Several ADTs for storing and retrieving data were discussed – Linear Lists, Binary Trees, BSTs, AVL Trees.

- An important operation Findkey() has a time complexity:
  - O(n) in Lists,
  - O(n) in Binary Trees,
  - O(log n) in BSTs,
  - O(log n) in AVL trees.
- Can Findkey() be implemented with a time complexity better than O(log n)?
- With Hash Tables it is possible to implement Findkey() with O(1) time complexity.

Hashing Techniques

- A hash table for a given key type consists of :
  - Hash function
  - Array (called table) of size N
- Hash function **maps** a key to a location in the table.
- Hash functions can be of two types:
  - Perfect hash functions
  - Imperfect hash functions

Hashing Techniques

- Perfect hash functions:
  - map a key to the “exact address‟ in the hash table.
  - The key can be found at the address without additional search.
  - Perfect hash functions are hard to determine and compute.

exact address F(key)

- Imperfect hash functions
  - map a key to a “home address‟ in the hash table.
  - The key may not be at the address and finding it may require additional search.
  - Imperfect hash functions are easy to determine and compute.

home address H(key)

Hashing Techniques

- In the hash table the data elements are scattered randomly throughout the hash table
  - there is no first, root or last element.
- Hash tables are suitable for implementing sets but not linear or hierarchical structures.

Example 1: Hash Table

- Table Size, N = 7.
  Hash Function: H(key) = key mod 7.
- Initially Hash Table is empty

0	empty
1	empty
2	empty
3	empty
4	empty
5	empty
6	empty

Example 1: Hash Table

- Insert:
- keys 374, 1091, 911are inserted.
- H(374) = 374 mod 7 = 3
- H(1091)=1091 mod 7= 6
- H(911)= 911 mod 7=1

0	empty
1	911
2	empty
3	374
4	empty
5	empty
6	1091

Example 1: Hash Table

- Retrieve:
- keys 374 and 740 are retrieved

H(374) = 374 mod 7 = 3 Table address 3 contains the key.

H(740) = 740 mod 7 = 5 Table address 5 is empty.

0	empty
1	911
2	empty
3	374
4	empty
5	empty
6	1091

Example 1: Hash Table

- key 227 is to be inserted.
- H(227)=227mod7=3
- Hash functions tells us store it in home address 3 but there is already a key stored in home address 3. This called a **collision**.
- **collision:** two distinct keys, *k*1 and *k*2, but *h*(*k*1) = *h*(*k*2).
- Since table is not empty the collided key 227 must be stored somewhere in the table.
- **Rehashing or Collision Resolution Strategies**

0	empty
1	911
2	empty
3	374
4	empty
5	empty
6	1091

HASH FUNCTION

Hash Function

- Performance of a hash table depends on its hash function.
- For a data set with a certain type of keys hash function formulated should minimize the number of collisions.
- The following slides deal with techniques of formulating various types of hash functions.

1. Digit Selection

- Keys may be student id. numbers (e.g. 427102345) or social security numbers (e.g. 981-101-0002)

- Hash function based on digit selection selects a subset of digits from the key e.g. last 3 digits may be selected.

345 ← H(427102345)

1. Digit Selection

- Which digits to select?
- The ones that are random in the data.
  - First 3 digits from the left, in student ids are the same for many students... not a good choice... last 3 digits are random.
- How many digits to select?
- Depends on the table size we want.
  - Select 3 digits – table size 1000, 4 digits – table size 10000.

- How to select the digits?
  - Last 3 digits from the left can be selected using mod 1000.
  - The middle 3 digits can be selected through:
  - (key / 1000) mod 1000

2. Division

- Hash function based on division is of the following form: H(key) = key mod m
- 0 <= H(key) < m.
- These functions lead to a number of collisions for certain values of m.
- E.g. if m = 25 all keys divisible by 5 map to positions 0, 5, 10, 15 and 20.

2. Division

- Ideally **m** must have no common factors with the keys
  - an easy way to ensure this is to chose m a prime number
  - prime number (or a prime) is a natural number greater than 1 that has no positive divisors other than 1 and itself-.
- Example:
- 2,3,5,7,11,13,17,19,23,31,37, ……

3. Multiplication

- A hash function based on multiplication
  - first squares the key
  - then chooses a certain subset of the digits of the product.
- Example:

  Suppose key = 54321. Hash function finds the square: (54321)2 = 2950771041 and then middle 3 digits are chosen i.e. 077.

4. Folding

- A hash function based on folding adds the digits of the key.
- Suppose key = $d_1d_2d_3d_4d_5$.
- The key can be folded at single digits, as follows:
- H(key) = $d_1+d_2+d_3+d_4+d_5$.
- Table size will be 46 since 0 <= H(key) <= 45.

4. Folding

- What if a larger table size is needed?
- the key can be folded at 2 digits, as follows:

$H(key) = 0d_1 + d_2d_3 + d_4d_5$"

Table size will be 208 since 0 <= H(key) <= 207.

- Folding can be used with other operations e.g.

H(key) = fold (key) mod m

5. Character Valued Keys

- Keys can be characters or strings e.g. key = xyz.
- In such keys the characters are replaced by their binary ASCII codes and the binary number is converted to decimal integer and treated as any other integer key.

COLLISION RESOLUTION STRATEGIES

Collision Resolution Strategies

- Also known as rehashing techniques.
- Strategies determine where to store a collided key in the event of a collision.
- Strategies can be grouped into the following three categories:
  - Open address methods
  - External or separate chaining
  - Coalesced chaining

Open Address Method

- The strategy finds an empty position in the table after the collision and stores the collided key at the empty position.

- Linear rehashing, quadratic rehashing, random rehashing and double rehashing fall in this category.

Linear Rehashing

- Also called linear probing.
- Each table location inspected is referred to as a **probe**
- Linear rehashing starts a (circular) sequential search through the table until an empty location is found or all the table has been examined.
- Rehash address (i.e. address of the next location to be inspected) is computed by

  rehash address = (p + c) mod TableSize
- p = previous collision address; c = a constant (we take c = 1)
- If an empty location is found at the rehash address the collided key is stored at the location.

Linear Rehashing

- Consider the hash table in the state shown.
- Attempt to insert 227 leads to a collision with 374 ... H(227) = 227 mod 7 = 3.
- According to linear rehashing
- the collided key 227 will be stored in the circularly next empty location, which is at table address 4. (See the next slide)

0	empty
1	911
2	empty
3	374
4	empty
5	empty
6	1091

Linear Rehashing

After Insert 227

		probe
0	empty	0
1	911	1
2	empty	0
3	374	1
4	227	2
5	empty	0
6	1091	0

Insert 421

H(421) = 421 mod 7 = 1

		probe
0	empty	0
1	911	1
2	421	2
3	374	1
4	227	2
5	empty	0
6	1091	0

Insert 624

H(624) = 624mod 7 = 1

		probe
0	empty	0
1	911	1
2	421	2
3	374	1
4	227	2
5	624	5
6	1091	0

- One probe is an access into the hash-table. Number of probes required to insert a key indicates the cost of inserting the key.

Linear Rehashing

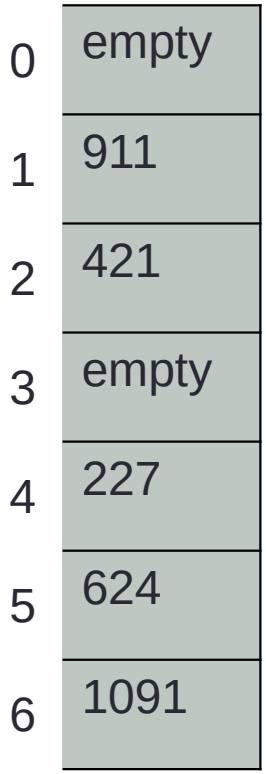
- Suppose key 624 is to be retrieved.
- H(624) = 1.
- After rehashing and 5 probes the key is 2 found at address 5.

- Suppose key 631 is to be retrieved. H(631) = 1. After rehashing and 7 probes the „empty" location 0 is found – if the key was present it would have been at 0. Therefore, retrieval algorithm returns „not found

0	empty	0
1	911	1
2	421	2
3	374	1
4	227	2
5	624	5
6	1091	0

Linear Rehashing

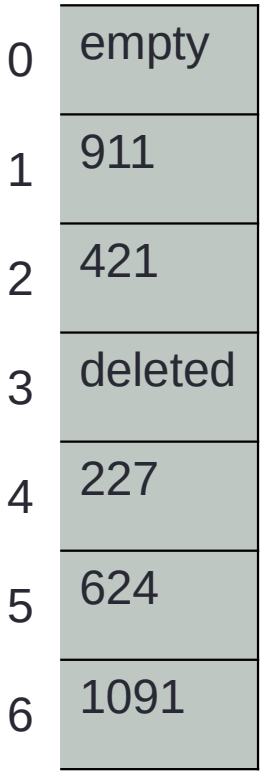
- Suppose one of the keys 421, 374 or 227 are removed
- e.g. 374 is removed, address 3 is marked empty
- Now attempt to find key 624 will incorrectly result in a failure – search will stop at address 3.
- This problem can be solved by having three different status for a table location.



Linear Rehashing

- status = {empty, occupied, deleted}

- During search for a key the search does not stop at locations with status deleted or occupied. So the key 624 is found at address 5.



Linear Rehashing

- The performance of a hash table employing linear rehashing as collision resolution strategy is measured in terms of the average number of probes required to insert a set of keys.

External Chaining

- Also called separate chaining
- According to this strategy the collided keys are stored in a list associated with the home address.
- Hash table is an array of lists.
- Insertions and deletions are easy to implement.

External Chaining

Insertions

Key=374

- 374mod7=3

Key = 1091

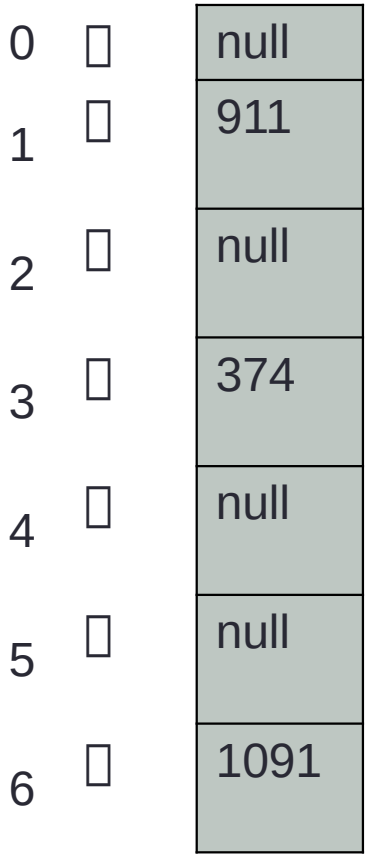
- 1091 mod 7 = 6

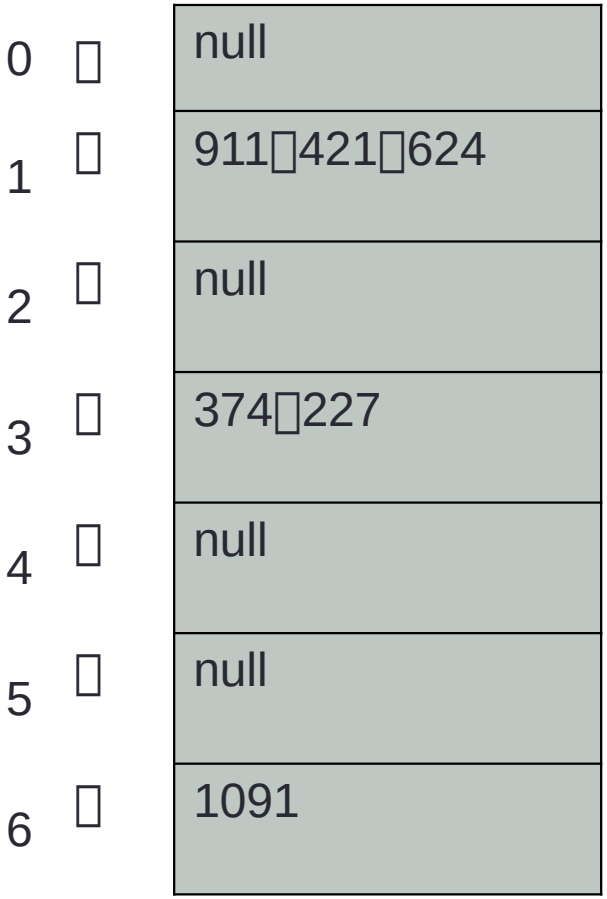
Key = 911

- 911 mod 7 = 1

Collisions

- Key=227
  - 227mod7=3
- Key = 421
  - 421mod7=1
- Key = 624
  - 624mod7=1





External Chaining

- **Advantages** (compared to Linear Rehashing)
- Deletions are easily possible.
- Number of elements can be greater than the table size.
- Retrieval operations are efficient since hash function is computed only once during retrieval.

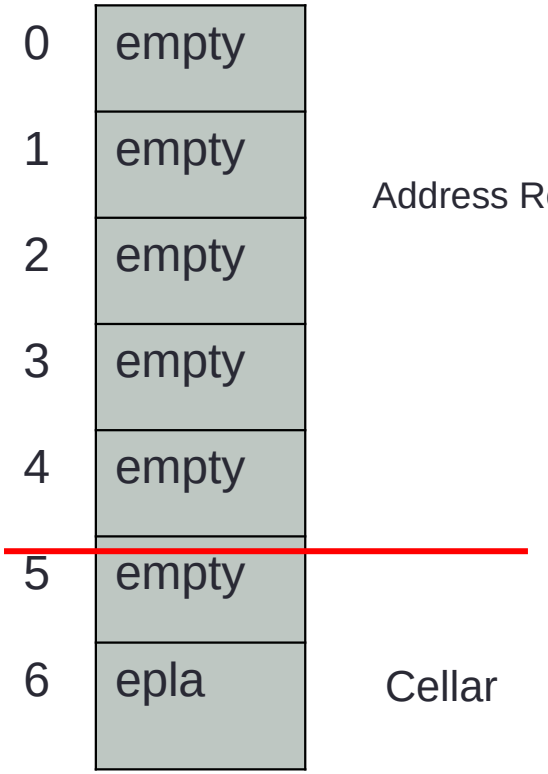
Coalesced chaining

- Similar to external chaining – collided elements are stored in lists.
- Similar to linear rehashing – lists are stored within the hash table.

- Hash table is divided into two regions:
  - an address region – stores normal keys
  - a cellar – stores collided keys.
- epla – empty position with largest address.

Coalesced chaining

H(key) = key mod 5



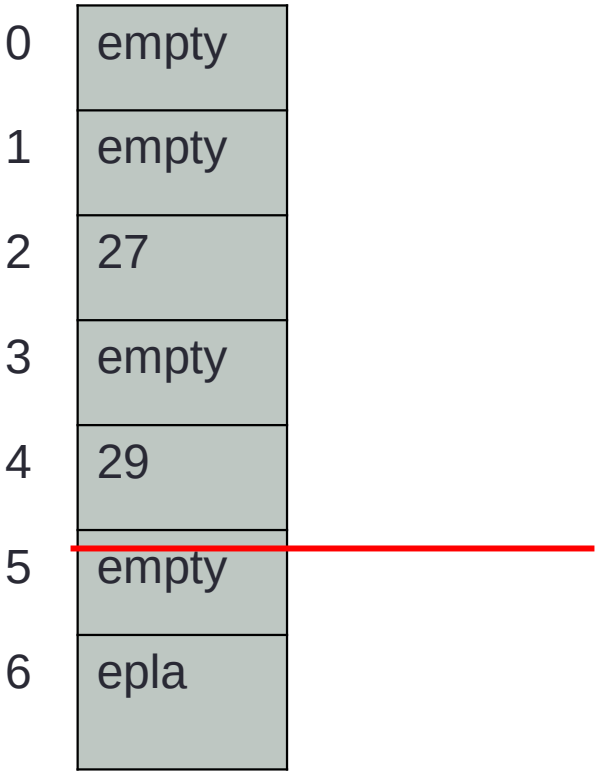
Insertions

Key = 27

27 mod 5 = 2

Key = 29

29 mod 5 = 4



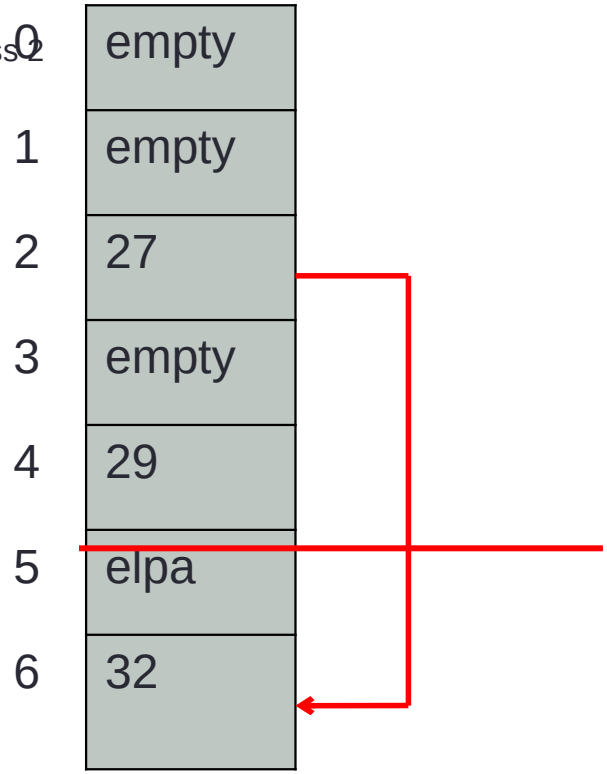
Coalesced chaining

Insertion with Collision

Key = 32

32 mod 5 = 2

Stored at epla & linked to link list at address 2

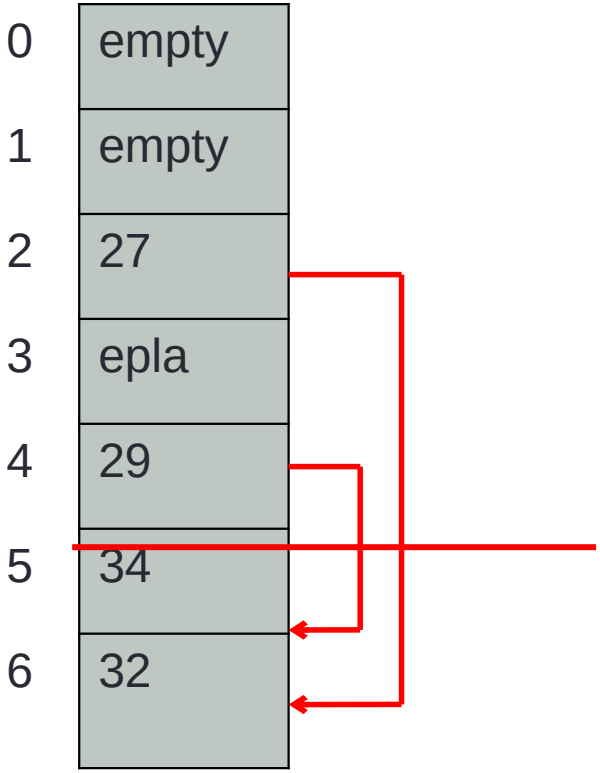


Insertion with Collision

Key = 34

34 mod 5 = 4

Stored at epla & linked to link list at address 4



Coalesced chaining

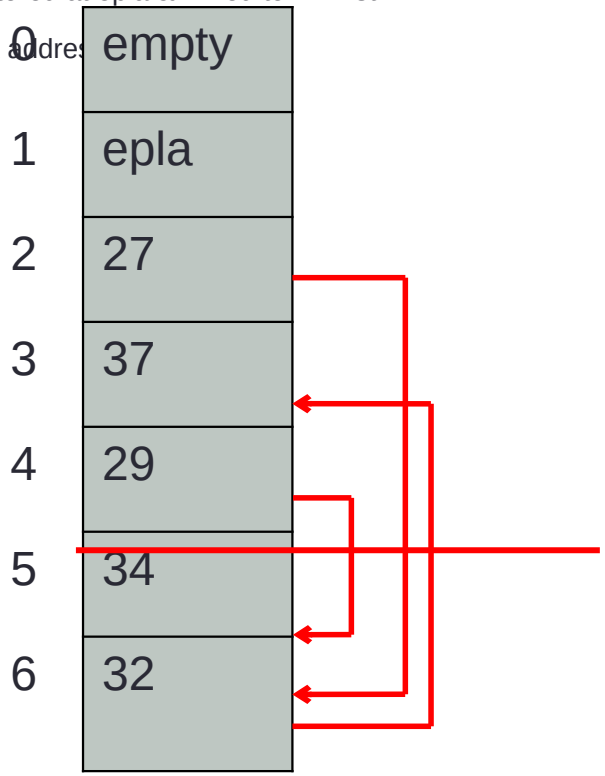
Insertions with Collision

Key = 37

37 mod 5 = 2

Stored at epla & linked to link list

at addres



insertions with Collision

Key = 47

47 mod 5 = 2

Stored at epla & linked to link list at address 2