Data Structures

Fall 2023

---

21. Hashing

Introduction

- The search operation on a sorted array using the binary search method takes O(logn)

- We can improve the search time by using an approach called Hashing

- Usually implemented on Dictionaries
  - Only support INSERT, SEARCH, and DELETE operations
    - No traversals etc.
  - These are sometimes called dictionary operations

- Hashing can make this happen in O(1) and is quite fast in practice

Dictionary

- A dictionary is a collection of elements where every element is a *(key, value)* pair
- Every key is usually distinct
- Typical dictionary operations are:
  - Insert a pair into the dictionary
  - Search the pair with a specified key
  - Delete the pair with a specified key

- Example: Collection of student records in a class
  - (key, value) = (student-number, a list of assignment and exam marks)
  - All keys are distinct

Dictionary as an Ordered Linear List

- We may implement a dictionary using array or chain (linked list) representation
  - L = (e1, e2, e3, …, en)
  - Each e**i** is a pair (key, value)

- But it leads to poor computational complexity
  - unsorted array: O(n) search time
  - sorted array: O(log$n$) search time
  - unsorted chain: O(n) search time
  - sorted chain: O(n) search time

- We need a better way to implement dictionary

Hash Table

- Hash table is a data structure that stores elements and allows insertions, lookups, and deletions to be performed in O(1) time
- A hash table is an alternative method for representing a dictionary
- In a hash table, a hash function is used to map keys into positions in a table. This act is called hashing
- Hash Table Operations
  - Search: compute h(k) and see if a pair exists
  - Insert: compute h(k) and place it in that position
  - Delete: compute h(k) and delete the pair in that position
- In ideal situation, hash table search, insert or delete takes O(1)

The properties of a good hash function

- Rule1: The hash value is fully determined by the data being hashed
- Rule2: The hash function uses all the input data
- Rule3: The hash function uniformly distributes the data across the entire set of possible hash values
- Rule4: The hash function generates very different hash values for similar strings

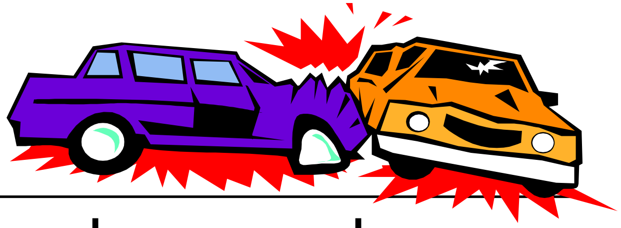
Designing a Good Hash Function

- A good hash function minimize collisions
- Collision: The condition resulting when two or more keys produce the same hash location
  - One Solution
    - Use a data structure that has more space for keys
  - Another Solution
    - Design hash function to minimize the collisions
    - Produce unique keys as much as possible
- To avoid collision causing worst case need to know statistical distribution of keys.

Issues in Hashing

- **Collision**
  - Collison occurs when two or more keys have save hash value
- **Clustering**
- **Overflow**
  - There is no space in the bucket for the new pair.
- **Solution**
  - Solution of overflow and collision is Rehashing

Collision



- The condition resulting when two or more keys produce the same hash location
- A good hash function minimizes collisions by spreading the elements uniformly throughout the array.
- Collision handling techniques
  - Linear Probing
  - Quadratic Probing
  - Random Probing
  - Double Hashing
  - Buckets
  - Chaining

Some Terminologies

- Open Addressing
  - All items are stored in the hash table itself
  - In addition to the cell data (if any), each cell keeps one of the three states: EMPTY, OCCUPIED, DELETED.
  - While inserting, if a collision occurs, alternative cells are tried until an empty cell is found.
- Probe sequence
  - A probe sequence is the sequence of array indexes that is followed in searching for an empty cell during an insertion, or in searching for a key during find or delete operations.
- Overflow
  - When hash table has no space to accommodate more keys hash table overflow occurs.
- Synonyms
  - When some entries are candidate for same slot or location in hash table

Collision Resolution Techniques

There are two broad ways of collision resolution:

1. Open Addressing: Array-based implementation.

   I. Linear probing (linear search)
   II. Quadratic probing (nonlinear search)
   III. Random Probing
   IV. Double hashing (uses two hash functions)

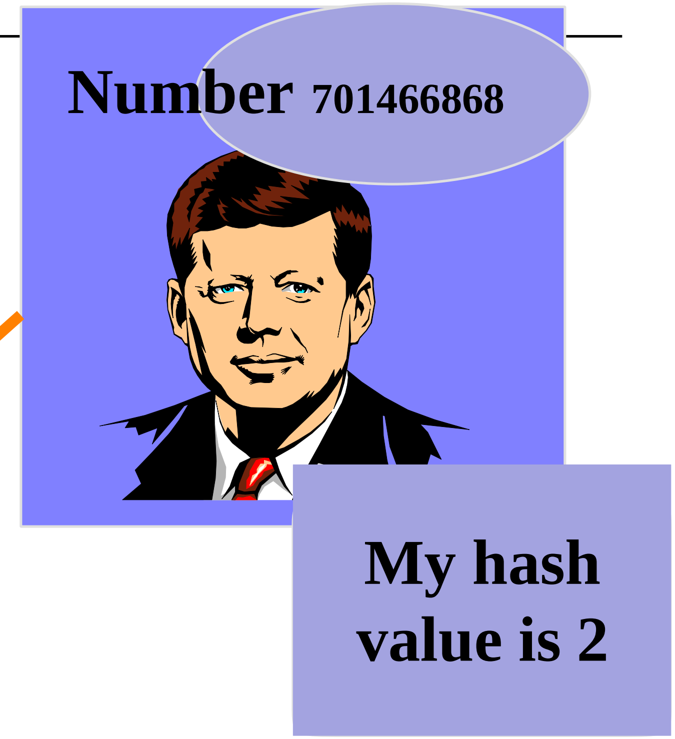
2. Separate Chaining: An array of linked list implementation

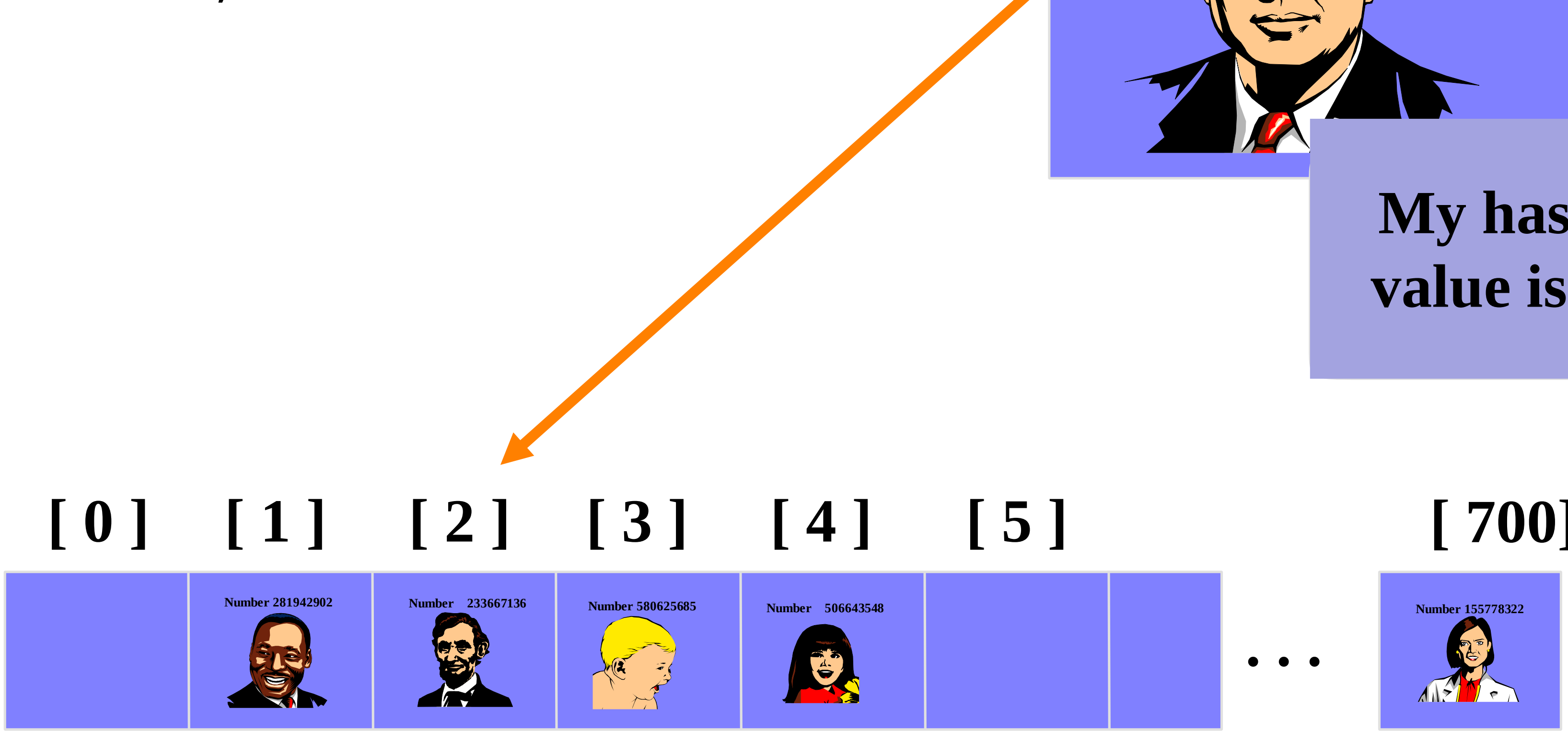
Linear Probing (Linear Search)

- Resolving a hash collision by sequentially searching a hash table beginning at the location return by the hash function
  - easy to compute
  - minimal number of collisions

Collisions

- Here is another new record to insert, with a hash value of 2.

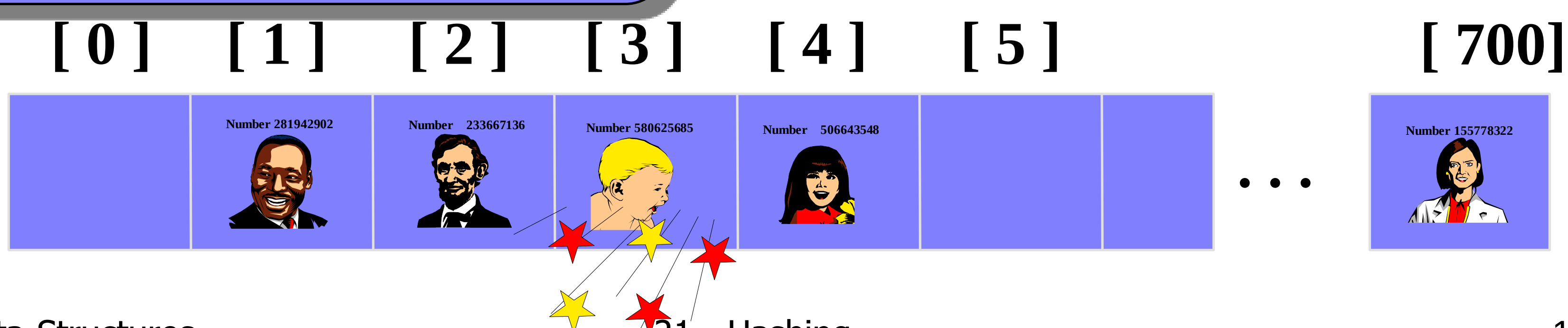




Collisions

- This is called a **collision**,
  because there is already another valid record at [2].

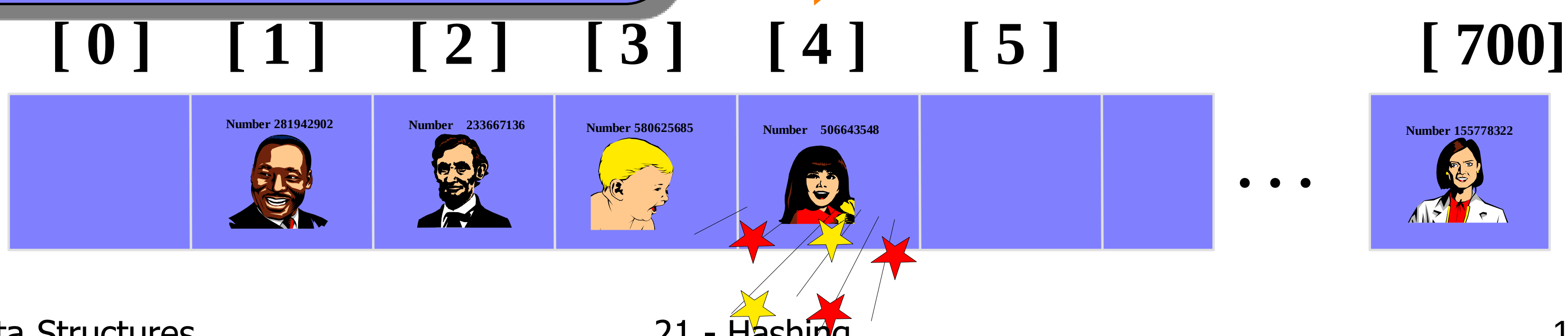
**When a collision occurs,
move forward until you find an empty spot.**



Collisions

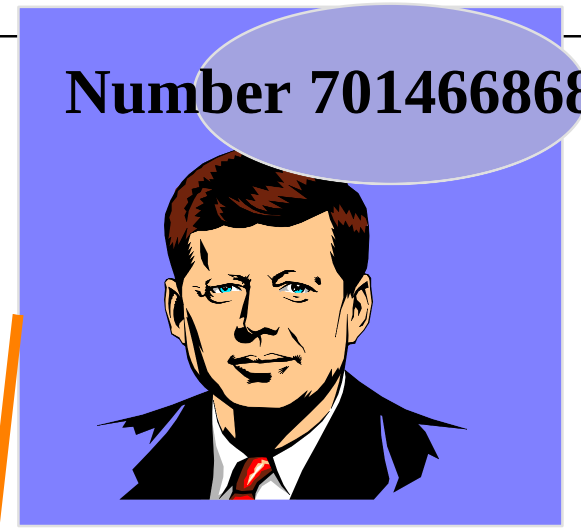
- This is called a **collision**, because there is already another valid record at [2].

When a collision occurs, move forward until you find an empty spot.

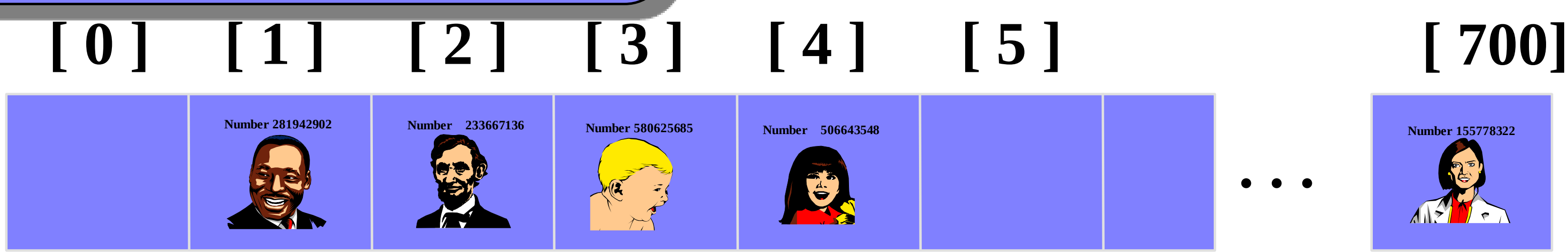


Collisions

- This is called a **collision**, because there is already another valid record at [2].



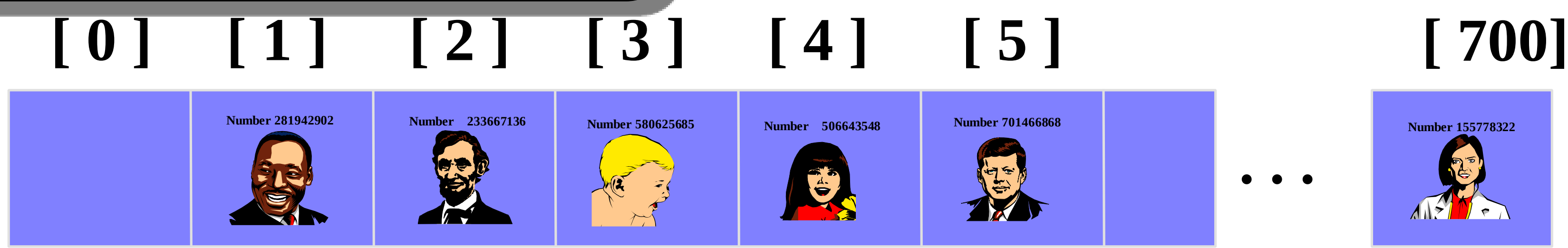
When a collision occurs, move forward until you find an empty spot.



Collisions

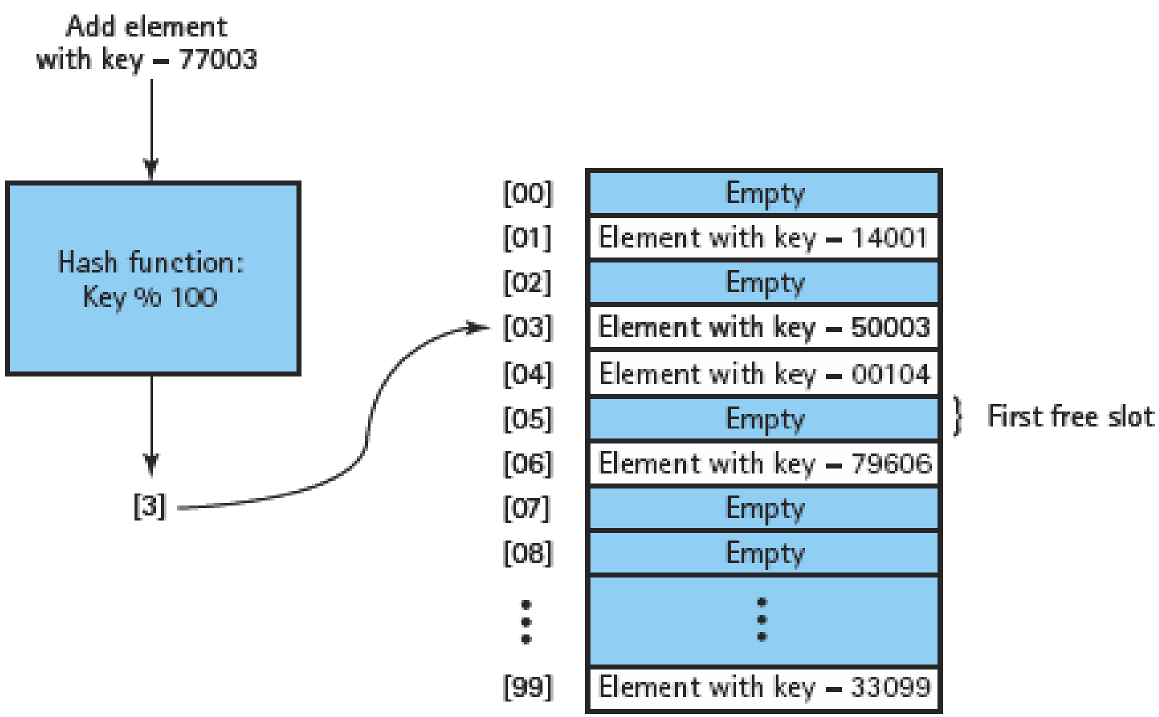
- This is called a **collision**, because there is already another valid record at [2].

The new record goes in the empty spot.



Linear Probing

- Resolving a hash collision by sequentially searching a hash table beginning at the location return by the hash function.



Linear Probing - Searching

To search for an element using Linear probing

- Perform the hash function on the key
- Compare the desired key to the actual key in the element at the designated location
- If the keys do not match use linear probing beginning at the next slot in array
- If key is found return true
- If not found return false

Linear Probing – Get And Insert

- number of buckets = 17
- H(key) = key % 17

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	0	45				6	23	7			28	12	29	11	30	33

- Insert pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

Linear Probing Task

- h(k) = *k* mod 13
- Insert keys:
- 18 41 22 44 59 32 31 73

0	1	2	3	4	5	6	7	8	9	10	11	12

Linear Probing Example

- h(k) = *k* mod 13
- Insert keys:
- 18 41 22 44 59 32 31 73

0	1	2	3	4	5	6	7	8	9	10	11	12

		41			18	44	59	32	22	31	72	
0	1	2	3	4	5	6	7	8	9	10	11	12

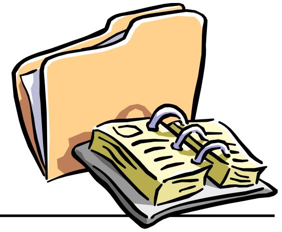
Insertion: Linear probing

```
template<class ItemType>
void ListType<ItemType>::InsertItem(ItemType item)
// Post: item is stored in the array at position item.Hash()
//       or the next free spot.
{
  int location;

  location = item.Hash();
  while (info[location] != emptyItem)
    location = (location + 1) % MAX_ITEMS;
  info[location] = item;
  length++;
}
```

Search: Linear Probing

```
template<class ItemType>
void ListType<ItemType>::RetrieveItem(ItemType& item, bool& found)
{
  int location;
  int startLoc;
  bool moreToSearch = true;

  startLoc = item.Hash();
  location = startLoc;
  do
  {
    if (info[location] == item || info[location] == emptyItem)
      moreToSearch = false;
    else
      location = (location + 1) % MAX_ITEMS;
  } while (location != startLoc && moreToSearch);
  found = (info[location] == item);
  if (found)
    item = info[location];
}
```

Linear Probing



To delete an element using Linear probing

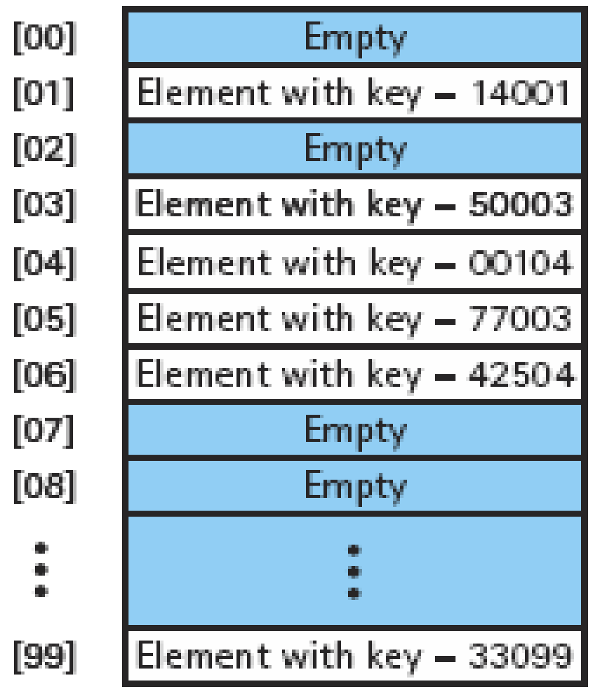
- Find the element with same search approach
- Replace the element with a constant to identify the place was previously occupied
- It will help pre-mature termination of the loop for searching.

Linear Probing – Clustering

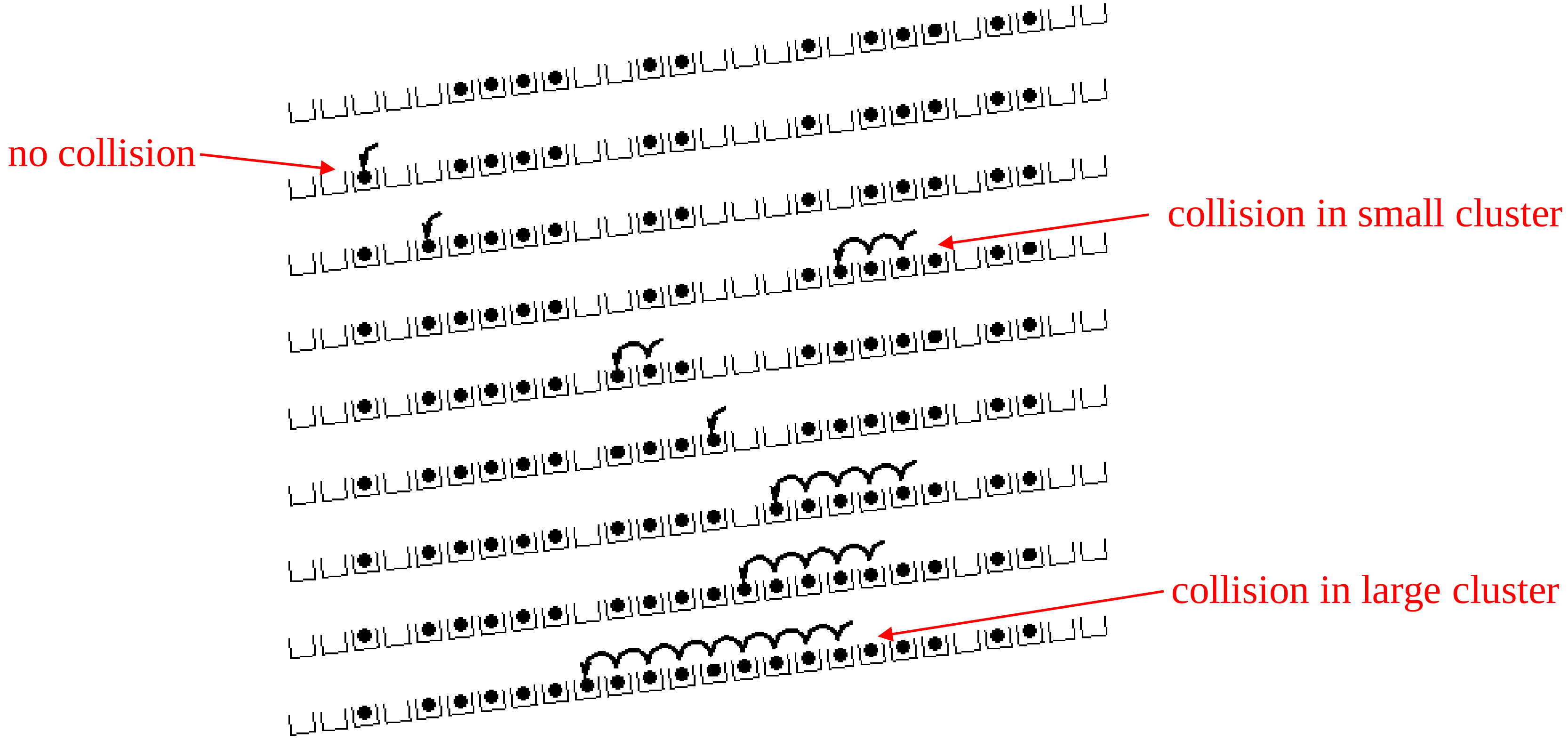
- If a hashing function groups key values together, this is called clustering of the keys.
- A good hashing function distributes the key values uniformly throughout the range.

Linear Probing – Clustering

- Clustering is the tendency of elements to become unevenly distributed in the hash table, with many elements clustering around a single hash location.



Linear Probing – Clustering



Linear Probing – Clustering

- A possible solution for reducing clustering:
  - Standard linear probing:

    (hash + 1) % arraySize
  - Improved linear probing:

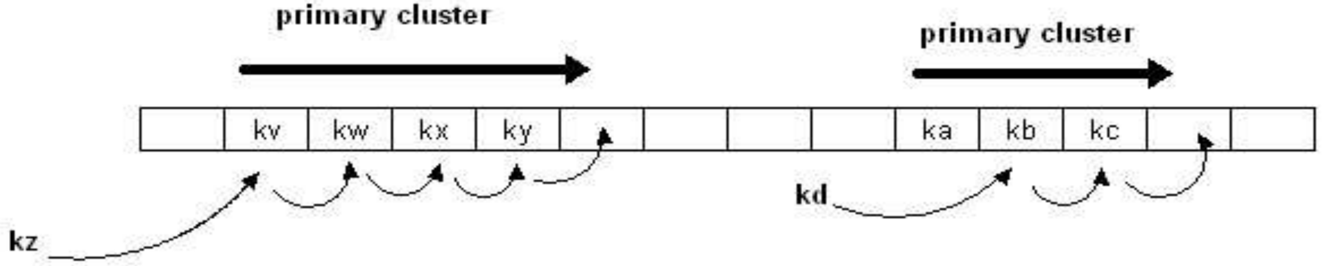
    (hash + stepSize) % arraySize

- Step size and array size should be co-prime

- (Example on board)
  - arraySize = 8, stepSize = 2
    - Test collision at 2
    - collision at 3
  - arraySize = 8, stepSize = 3

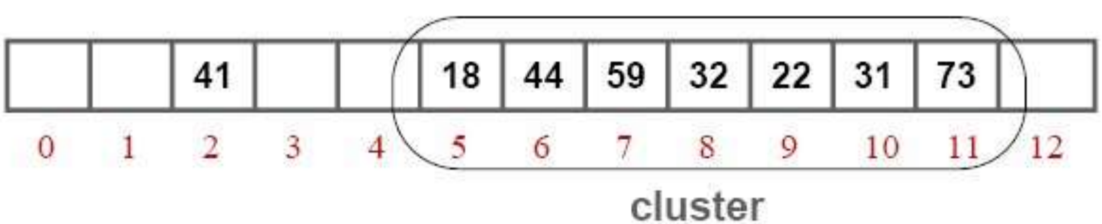
Disadvantage of Linear Probing: Primary Clustering

- Linear probing is subject to a primary clustering phenomenon
- Elements tend to cluster around table locations that they originally hash to
- Primary clusters can combine to form larger clusters. This leads to long probe sequences and hence deterioration in hash table efficiency



Disadvantage of Linear Probing: Primary Clustering

- **Example of a primary cluster:** Insert keys: *18, 41, 22, 44, 59, 32, 31, 73,* in this order, in an originally empty hash table of size 13, using the hash function h(key) = key % 13 and c(i) = i:
  - h(18) = 5
  - h(41) = 2
  - h(22) = 9
  - h(44) = 5+1
  - h(59) = 7
  - h(32) = 6+1+1
  - h(31) = 5+1+1+1+1+1
  - h(73) = 8+1+1+1



Task

- Insert the following numbers in a Hash Table

22, 122, 55, 174, 66, 555, 99, 11, 155

- Array size = 11
- Hash = Key % array size
- ReHash = (Hash + **3**) % array size

Quadratic Probing

- Resolving a hash collision by using rehashing formula

$$(hash \pm i^2) \% array\_size$$

- where i is the number of times that the rehash function has been applied

- It distributes the key on wide range over the hash table
- Quadratic probing reduces clustering

Quadratic Probing

$$f(i) = i^2$$

Less likely to encounter Primary Clustering

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + 1) mod TableSize

  2^{th} probe = (h(k) + 4) mod TableSize

  3^{th} probe = (h(k) + 9) mod TableSize

  . . .

  i^{th} probe = $(h(k) + i^2)$ mod TableSize

Quadratic Probing

- Example: Insert the keys **23, 13, 21, 14, 7, 8, and 15**, in this order, in a hash table of size **7** using quadratic probing with $\mathbf{c(i) = \pm i^2}$ and the hash function: **h(key) = key % 7**

- The required probe sequences are given by:

$$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$$

Quadratic Probing

$h_i(key) = (h(key) \pm i^2) \% 7 \quad i = 0, 1, 2, 3$

$h_0(23) = (23 \% 7) \% 7 = 2$

$h_0(13) = (13 \% 7) \% 7 = 6$

$h_0(21) = (21 \% 7) \% 7 = 0$

$h_0(14) = (14 \% 7) \% 7 = 0$ collision

$h_1(14) = (0 + 1^2) \% 7 = 1$

$h_0(7) = (7 \% 7) \% 7 = 0$ collision

$h_1(7) = (0 + 1^2) \% 7 = 1$ collision

$h_{-1}(7) = (0 - 1^2) \% 7 = -1$

NORMALIZE: $(-1 + 7) \% 7 = 6$ collision

$h_2(7) = (0 + 2^2) \% 7 = 4$

$h_0(8) = (8 \% 7) \% 7 = 1$ collision

$h_1(8) = (1 + 1^2) \% 7 = 2$ collision

$h_{-1}(8) = (1 - 1^2) \% 7 = 0$ collision

$h_2(8) = (1 + 2^2) \% 7 = 5$

$h_0(15) = (15 \% 7) \% 7 = 1$ collision

$h_1(15) = (1 + 1^2) \% 7 = 2$ collision

$h_{-1}(15) = (1 - 1^2) \% 7 = 0$ collision

$h_2(15) = (1 + 2^2) \% 7 = 5$ collision

$h_{-2}(15) = (1 - 2^2) \% 7 = -3$

NORMALIZE: $(-3 + 7) \% 7 = 4$ collision

$h_3(15) = (1 + 3^2) \% 7 = 3$

0	O	21
1	O	14
2	O	23
3	O	15
4	O	7
5	O	8
6	O	13

Task

- Insert the following numbers

  2, 12, 14, 18, 20, 24, 32, 22, 144, 55, 66, 45, 49

- Array size = 13
- Hash = Key % array size
- ReHash = (Hash $\pm$ i^2) % array size

Overflow

- Hash Table may get full
  - No more insertions possible
- Hash table may get *almost* full
  - Insertions, deletions, search take longer time
- Solution: Rehash
  - Build another table that is twice as big and has a new hash function
  - Move all elements from smaller table to bigger table
- Cost of Rehashing = $O(N)$
  - But happens only when table is close to full
  - Close to full = table is X percent full, where X is a tunable parameter

Random Probing

- Resolving a hash collision by generating pseudo-random hash values in successive applications of the rehash function

- Random probing eliminate primary clustering but produce secondary clustering and is a slower technique

Random Probing

$$h_0(X) = key \ \% \ TableSize$$

$$h_1(X) = (h_0(X) + r_i) \ \% \ TableSize$$

where r_i is the ith value in a random permutation of the numbers from 1 to TableSize −1.

Example

Size of array = 10

Insert 157, 273, 17, 913, 110 using Random probing

1st step : Define a random permutation of values 0 to size-1

0	3	7	6	1	4	9	2	5	8
0	1	2	3	4	5	6	7	8	9

Insert 157

Insert a record with key value 157.

0	1	2	3	4	5	6	7	8	9
							157		

Permutation:

0	1	2	3	4	5	6	7	8	9
0	3	7	6	1	4	9	2	5	8

Insert 273

Insert a record with key value 273.

0	1	2	3	4	5	6	7	8	9
			273				157		

Permutation:

0	1	2	3	4	5	6	7	8	9
0	3	7	6	1	4	9	2	5	8

Insert 17

Insert a record with key value 17. Unfortunately there is already a value in slot 7.

			273				157		
0	1	2	3	4	5	6	7	8	9

Permutation:

0	3	7	6	1	4	9	2	5	8
0	1	2	3	4	5	6	7	8	9

Insert 17

So now we look in the permuation array for the value at position perm[1], and add that value to the home slot index (which is 7), to get a value of 10 % 10, which is slot 0.

17			273				157		
0	1	2	3	4	5	6	7	8	9

Permutation:

0	3	7	6	1	4	9	2	5	8
0	1	2	3	4	5	6	7	8	9

Insert 913

Insert a record with key value 913. Unfortunately there is already a value in slot 3.

0	1	2	3	4	5	6	7	8	9
17			273				157		

Permutation:

0	1	2	3	4	5	6	7	8	9
0	3	7	6	1	4	9	2	5	8

Insert 913

So now we look in the permuation array for the value at position perm[1], and add that value to the home slot index (which is 3), to get a value of 6.

17			273			913	157		
0	1	2	3	4	5	6	7	8	9

Permutation:

0	3	7	6	1	4	9	2	5	8
0	1	2	3	4	5	6	7	8	9

Insert 110

Insert a record with key value 110. Unfortunately there is already a value in slot 0.

0	1	2	3	4	5	6	7	8	9
17			273			913	157		

Permutation:

0	1	2	3	4	5	6	7	8	9
0	3	7	6	1	4	9	2	5	8

So now we look in the permuation array for the value at position perm[1], and add that value to the home slot index (which is 0), to get a value of 3. Unfortunately, slot 3 is full as well!

0	1	2	3	4	5	6	7	8	9
17			273			913	157		

Permutation:

0	1	2	3	4	5	6	7	8	9
0	3	7	6	1	4	9	2	5	8

So now we look in the permuation array for the value at position perm[2], and add that value to the home slot index (which is 0), to get a value of 7. Unfortunately, slot 7 is full as well!

0	1	2	3	4	5	6	7	8	9
17			273			913	157		

Permutation:

0	1	2	3	4	5	6	7	8	9
0	3	7	6	1	4	9	2	5	8

So now we look in the permuation array for the value at position perm[3], and add that value to the home slot index (which is 0), to get a value of 6. Unfortunately, slot 6 is full as well!

17			273			913	157		
0	1	2	3	4	5	6	7	8	9

Permutation:

0	3	7	6	1	4	9	2	5	8
0	1	2	3	4	5	6	7	8	9

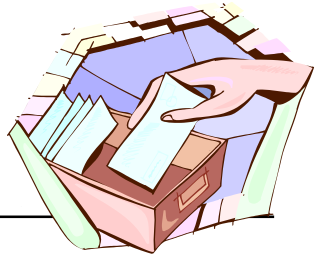
So now we look in the permuation array for the value at position perm[4], and add that value to the home slot index (which is 0), to get a value of 1. Finally!

17	110		273			913	157		
0	1	2	3	4	5	6	7	8	9

Permutation:

0	3	7	6	1	4	9	2	5	8
0	1	2	3	4	5	6	7	8	9

Double Hashing



(1) Use one hash function to determine the first slot

(2) Use a second hash function to determine the increment for the probe sequence

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m, \quad i=0,1,\ldots$$

- Initial probe: $h_1(k)$
- Make the offset to the next position probed depend on the key value, so it can be different for different keys
- Avoids clustering

Double Hashing: Example

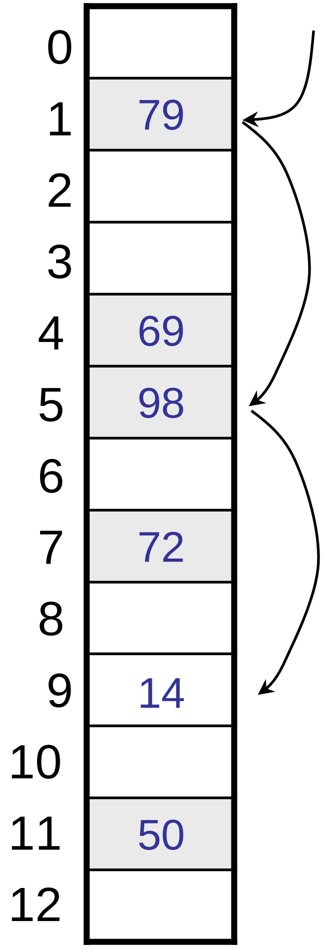
$$h_1(k) = k \bmod 13$$
$$h_2(k) = 1 + (k \bmod 11)$$
$$h(k, i) = (h_1(k) + i\, h_2(k)) \bmod 13$$

- Insert key 14:

$$h_1(14, 0) = 14 \bmod 13 = 1$$

$$h(14, 1) = (h_1(14) + h_2(14)) \bmod 13$$
$$= (1 + 4) \bmod 13 = 5$$

$$h(14, 2) = (h_1(14) + 2\, h_2(14)) \bmod 13$$
$$= (1 + 8) \bmod 13 = 9$$



Double Hashing

$$f(i) = i * g(k)$$

where g is a second hash function

- A good choice for g is to choose a prime R < TableSize and let $g(k) = R - (k \bmod R)$.
- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + g(k)) mod TableSize

  2^{th} probe = (h(k) + 2*g(k)) mod TableSize

  3^{th} probe = (h(k) + 3*g(k)) mod TableSize

  . . .

  i^{th} probe = (h(k) + i*g(k)) mod TableSize

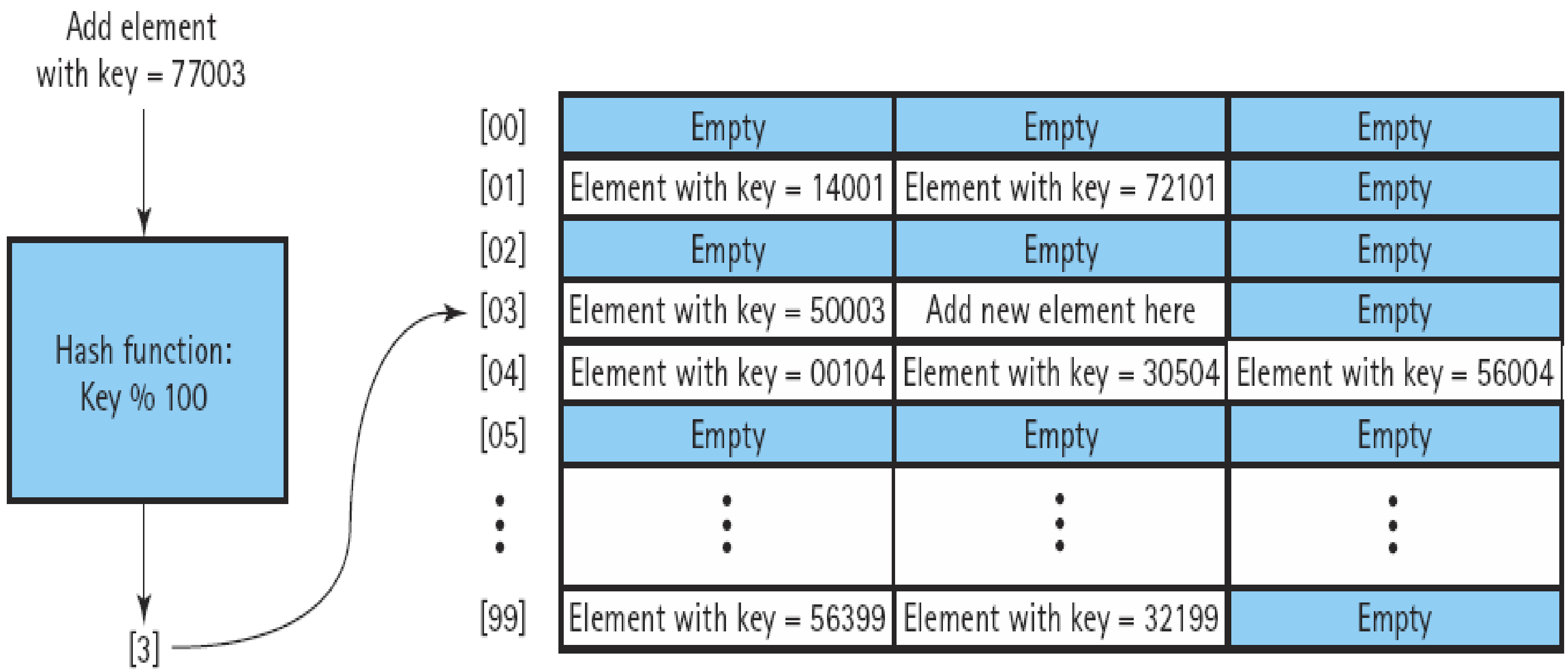
Bucket

- A collection of elements associated with a particular hash location
- Handle collision by allowing multiple element keys to hash to the same location
- A solution is to let each computed hash location contain slots for multiple elements
- Each of these multi-element location is called a bucket

Bucket

- Slots are grouped into buckets
- The hash function transforms the key into a bucket number
- Each bucket contains B slots and no collision occurs until the bucket is full.
- At that point you need to apply a collision processing strategy to find another bucket

Bucket

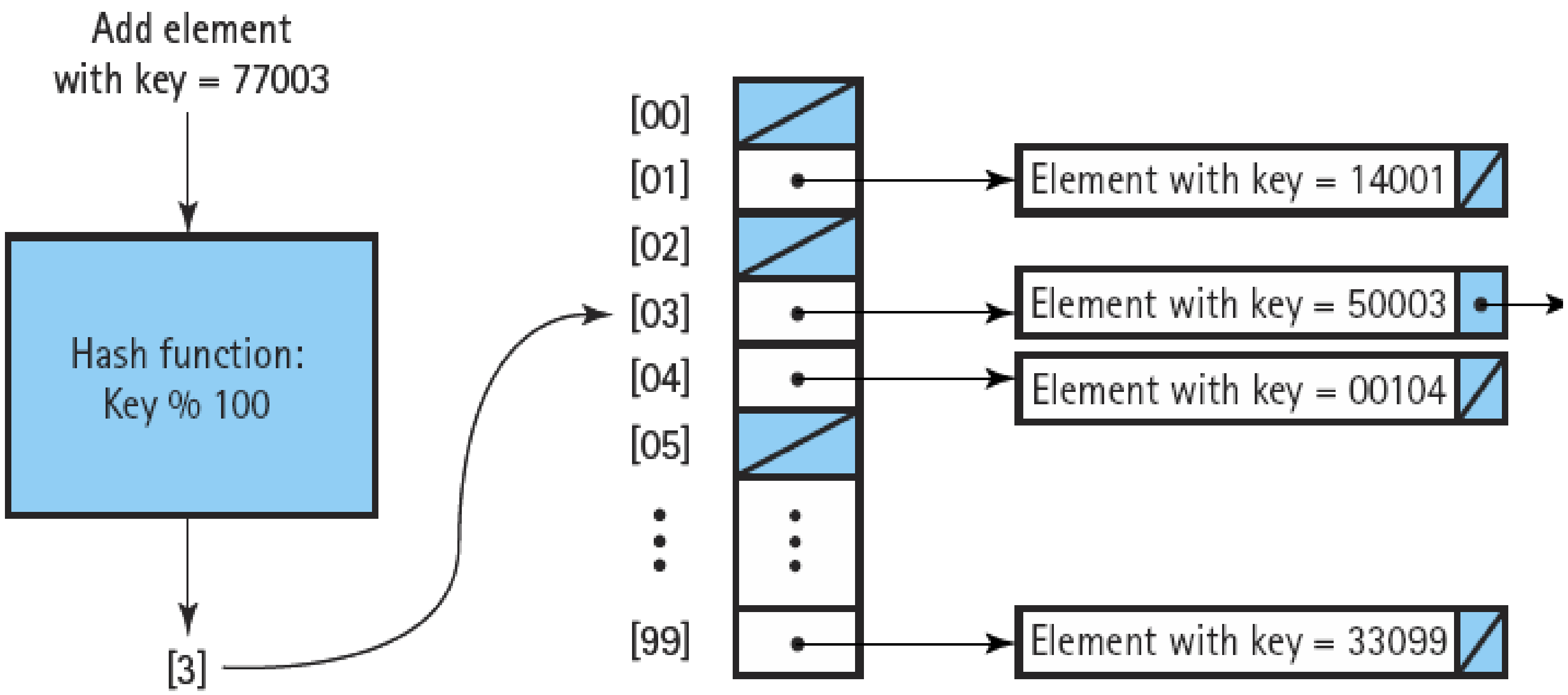


When the bucket becomes full, we must again deal with the problem of handling collision

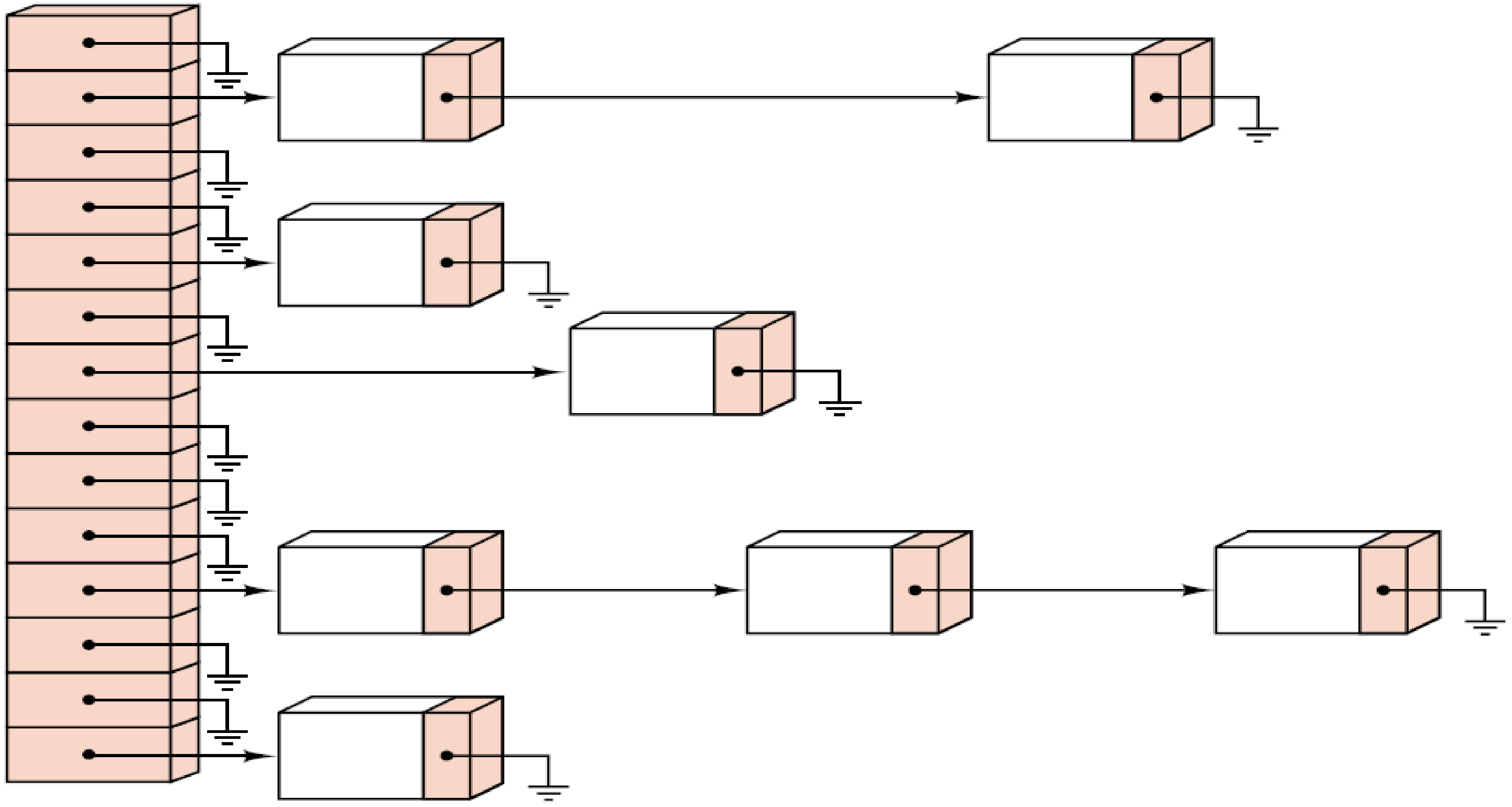
Chaining

- A linked list of elements that share the same hash location
- Use the hash value not as the actual location of the element, but rather as the index into an array of pointers
- Each pointer accesses a chain of elements that share the same hash location
- Easy to delete an element from the table

Chaining


Add element
with key = 77003
Hash function:
Key % 100
[3]
[00]
[01]
[02]
[03]
[04]
[05]
[99]
Element with key = 14001
Element with key = 50003
Element with key = 00104
Element with key = 33099

Chaining

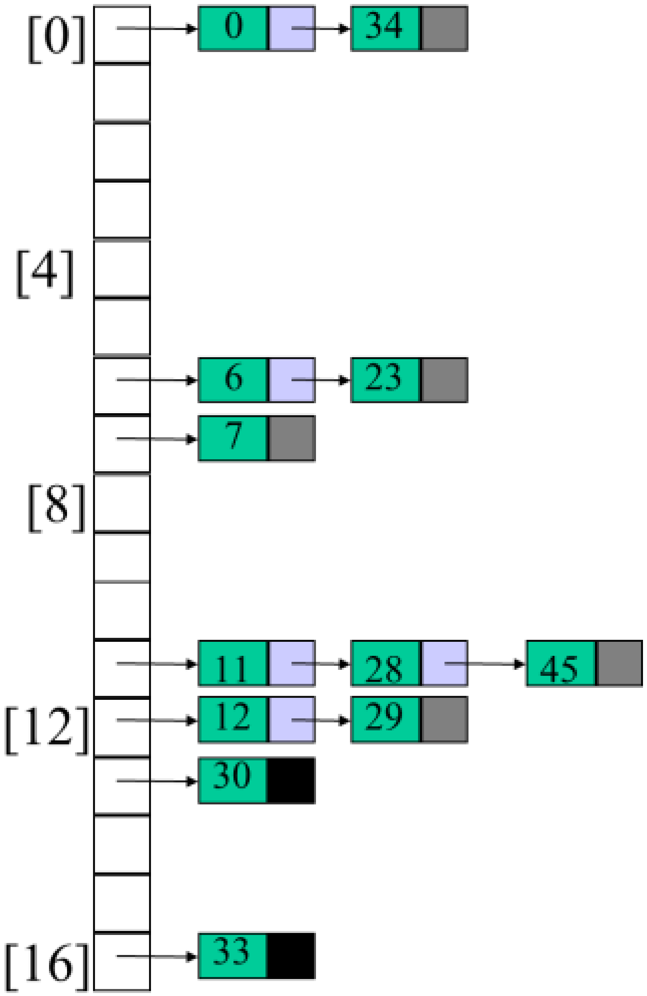


Task – Sorted Chain

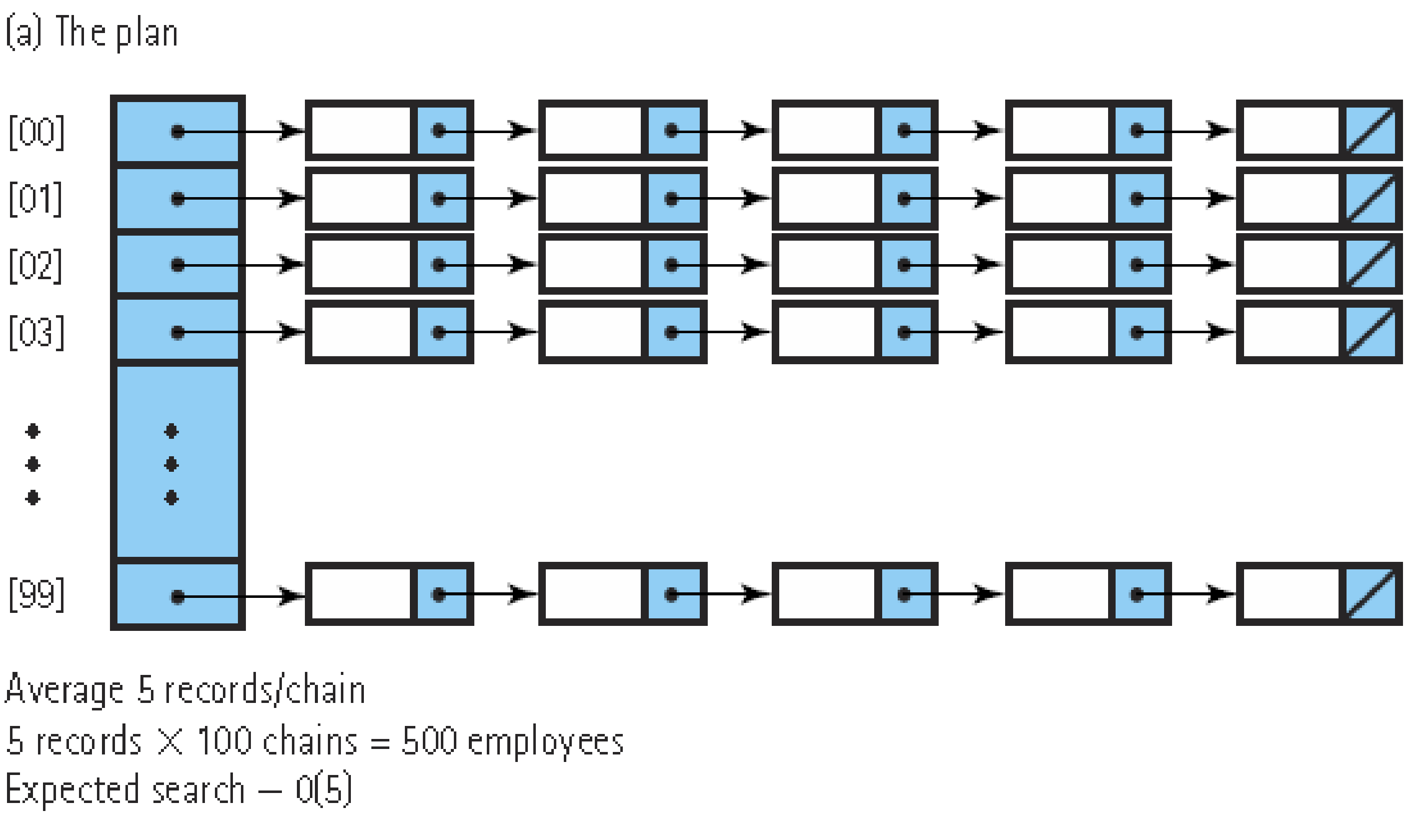
- Insert the following values in a sorted chain

  6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

- Hash = key % 17



Choosing Good Hash Function



Any Question So Far?