HASHING

REVIEW OF SEARCHING TECHNIQUES

Recall the efficiency of searching techniques covered earlier.

- The sequential search algorithm takes time proportional to the data size, i.e, O(n).
- Binary search improves on liner search reducing the search time to O(log n).
- With a BST, an O(log n) search efficiency can be obtained; but the worst-case complexity is O(n).
- To guarantee the O(log n) search time, BST height balancing is required.

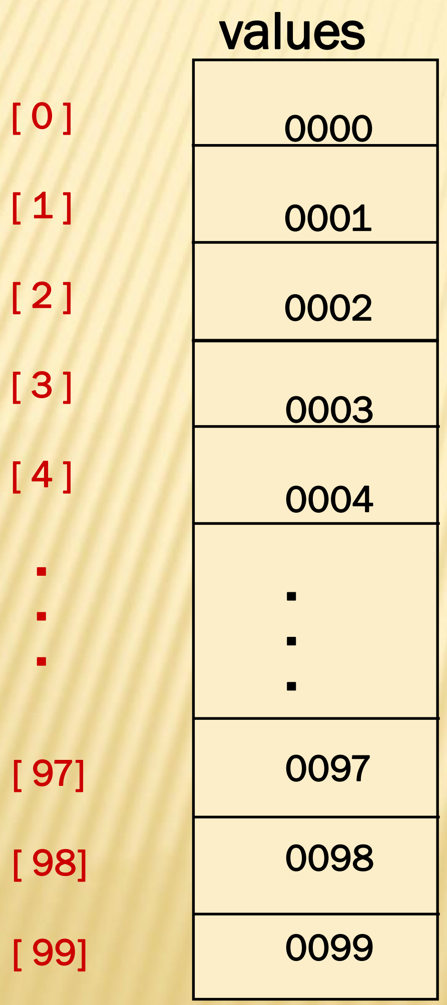
REVIEW OF SEARCHING TECHNIQUES

- Can we do better than that? Is it possible to design a search of O(1) – that is, one that has a constant search time, no matter where the element is in the list

HASHING

- is used to order and access elements in a list quickly -- the goal is O(1) time -- by using a function of the key value to identify its location in the list.
- The function of the key value is called a hash function.

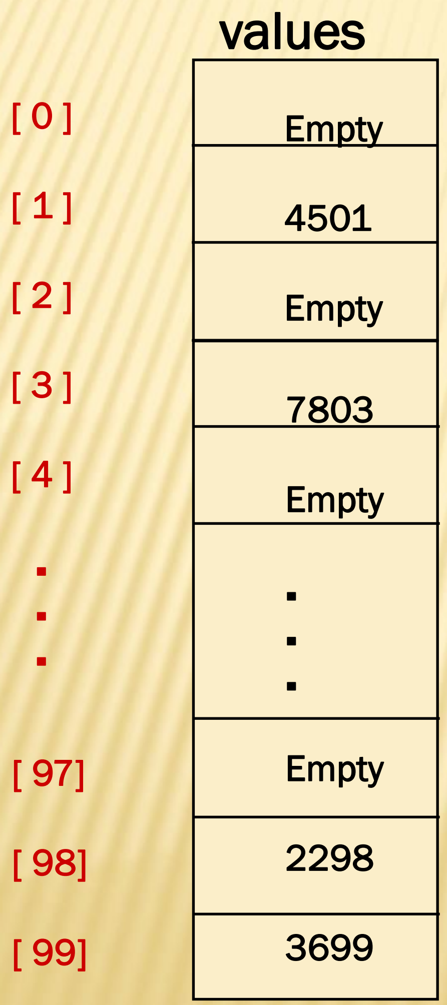
USING A HASH FUNCTION



HandyParts company makes no more than 100 different parts. But the parts all have four digit numbers which ranges from 0000 to 0100.

We can directly access any part record through the array index. i.e. there is one-to-one correspondence between Part number & index

USING A HASH FUNCTION



Now if another company makes no more than 100 different parts. But the parts all have four digit numbers with no restriction on range .

What to do?

This hash function can be used to store and retrieve parts in an array.

Hash(key) = partNum % 100

PLACING ELEMENTS IN THE ARRAY

	values
[0]	5500 Empty
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Use the hash function

Hash(key) = partNum % 100

to place the element with

part number 5500 in the

array.

Hash(key) = 5500 % 100 =0

PLACING ELEMENTS IN THE ARRAY

	values
[0]	5500
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Use the hash function

Hash(key) = partNum % 100

to place the element with

part number 5502 in the

array.

PLACING ELEMENTS IN THE ARRAY

	values
[0]	5500
[1]	4501
[2]	Empty
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Use the hash function

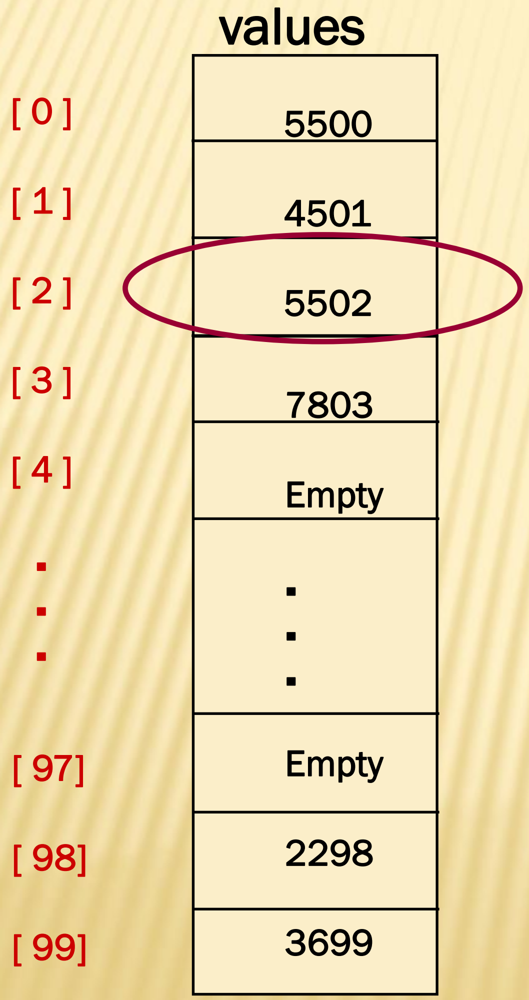
Hash(key) = partNum % 100

to place the element with

part number 5502 in the

array.

PLACING ELEMENTS IN THE ARRAY



Next place part number 6702 in the array.

Hash(key) = partNum % 100

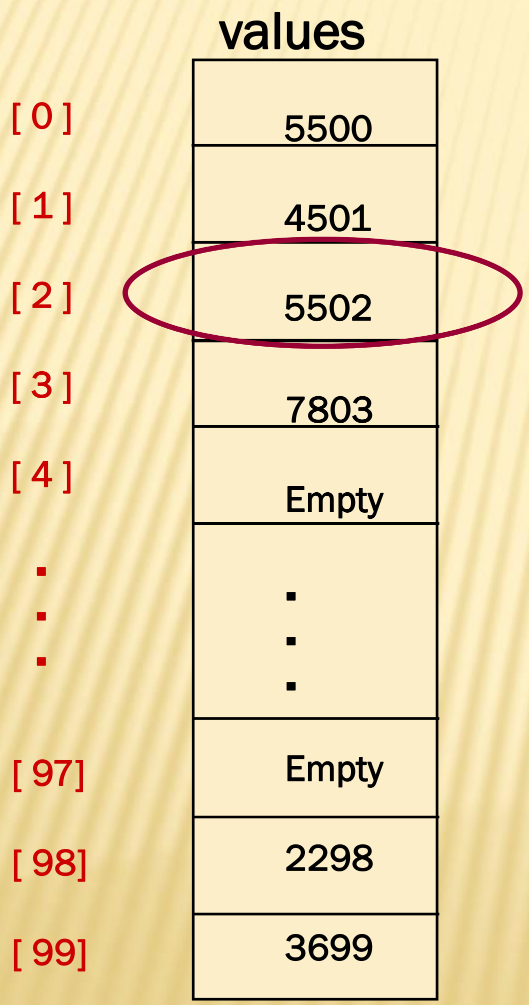
6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS

The condition resulting when two or more keys produce the same hash location

HOW TO RESOLVE THE COLLISION?



One way is by linear probing.
This uses the rehash function

(HashValue + 1) % 100

repeatedly until an empty location
is found for part number 6702.

Linear Probing: Resolving a hash collision by sequentially searching a hash table beginning at the location returned by the has function.

RESOLVING THE COLLISION

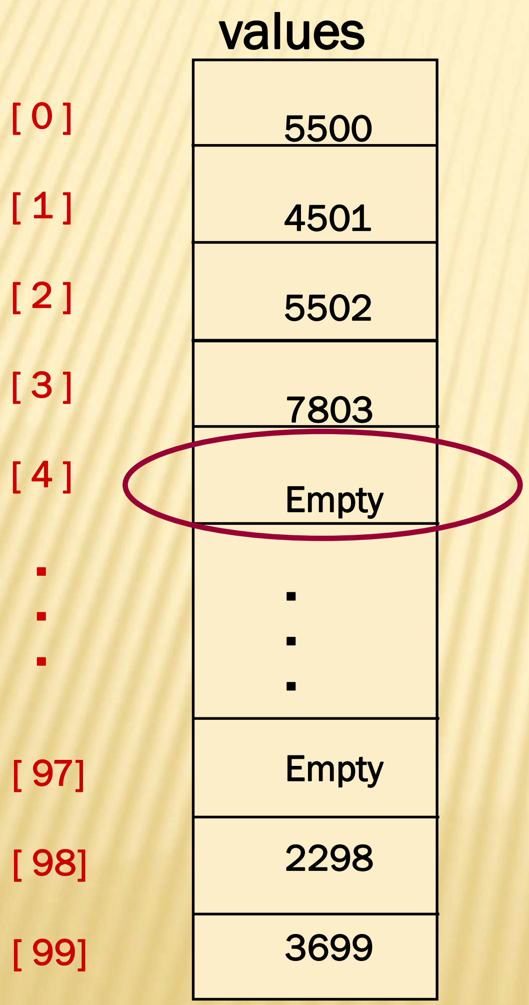
	values
[0]	5500
[1]	4501
[2]	5502
[3]	7803
[4]	Empty
. . .	. . .
[97]	Empty
[98]	2298
[99]	3699

Still looking for a place for 6702 using the function

(HashValue + 1) % 100

(6702 + 1) % 100 = 3

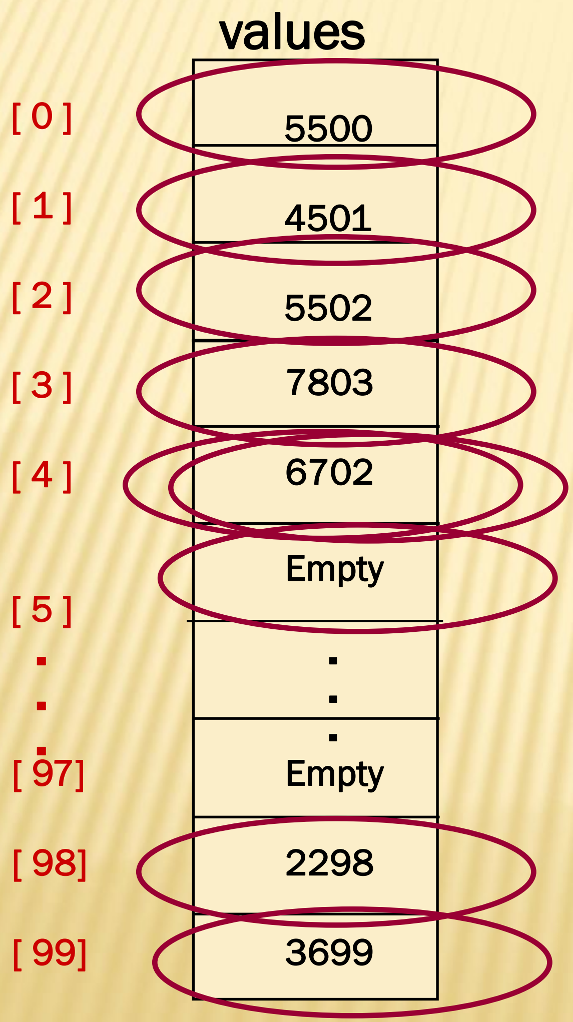
COLLISION RESOLVED



Part 6702 can be placed at the location with index 4.

(6702 + 2) % 100 = 4

COLLISION RESOLVED



Part 6702 is placed at the location with index 4.

Where would the part with number 4598 be placed using linear probing?

4598 will be stored at index 5
/*treating list as circular*/

BUCKETS & CHAINING

- Another alternative for handling collisions is to allow multiple element keys to hash to the same location.
- Bucket
  - A collection of elements associated with a particular hash location

BUCKETS & CHAINING

- Suppose we have a bucket of size 3. so 3 elements can share the location.

[00]	5460 Empty	Empty	Empty
[01]	14001	72101	Empty
[02]	9872	5462 Empty	9462 Empty
. .	. .	. .	. .
.	.	.	.
[99]	19899	2399	199

Insert 5462

5462%100 = 2

Insert 5460

5460%100 = 0

Insert 9462

9462%100 = 2

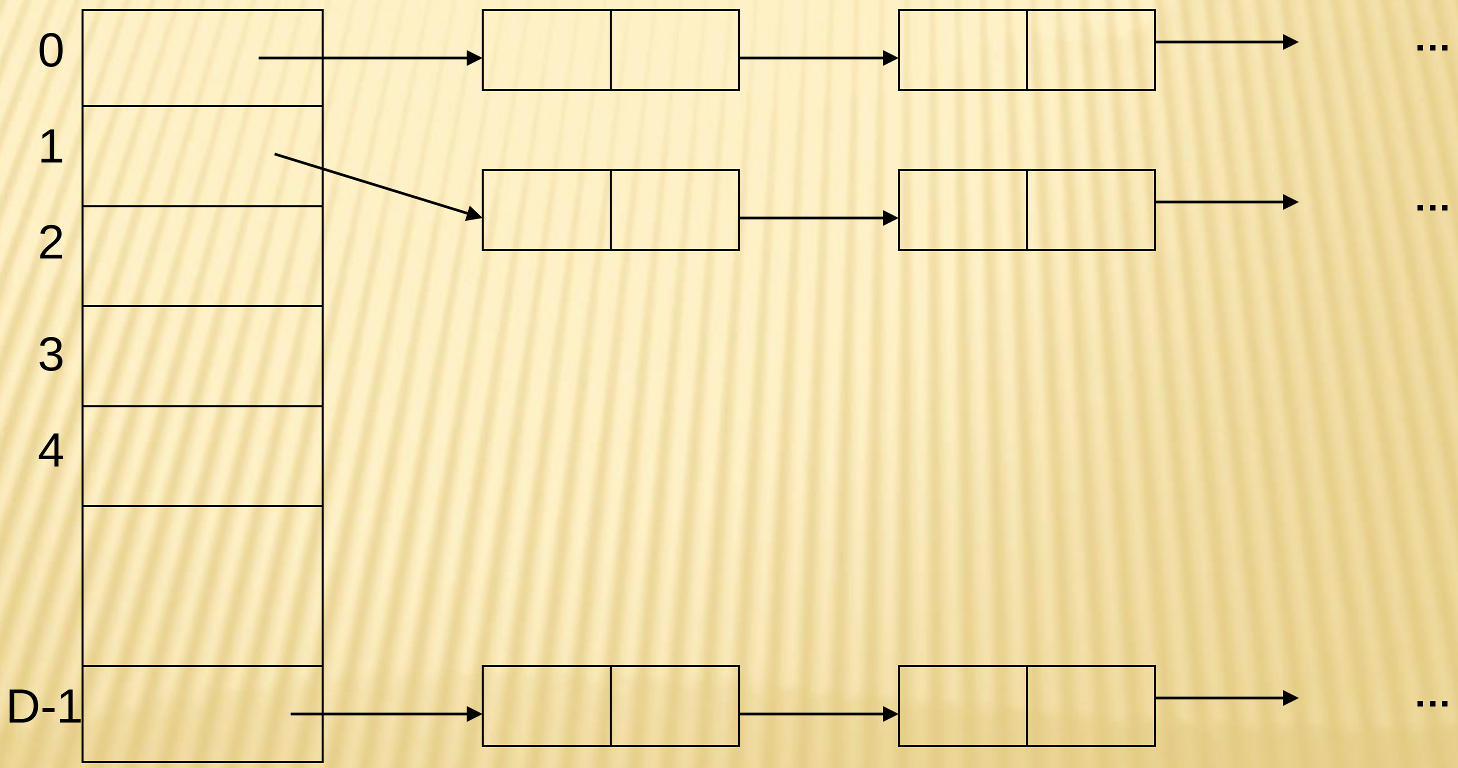
Insert 71462

71462%100 = 2

Where to insert?

CHAINING

- A linked list of elements that share the same hash location



HASH TABLES

- There are two types of Hash Tables: **Open-addressed Hash Tables** and **Separate-Chained Hash Tables**.

- **An Open-addressed *Hash Table*** is a one-dimensional array indexed by

  integer values that are computed by an index function called a ***hash function***.

- **A Separate-Chained *Hash Table*** is a one-dimensional array of linked lists indexed by integer values that are computed by an index function called a ***hash function***.

- Hash tables are sometimes referred to as ***scatter tables***..\

- Typical hash table operations are:

  - ***Insertion.***
  - ***Searching***
  - ***Deletion.***

TYPES OF HASHING

- There are two types of hashing :

  1. ***Static hashing***: In static hashing, the hash function maps search-key values to a fixed set of *locations*.

  2. ***Dynamic hashing***: In dynamic hashing a hash table can grow to handle more items. The associated hash function must change as the table grows.

- The ***load factor*** of a hash table is the ratio of the number of keys in the table to the size of the hash table.

- Note: The higher the load factor, the slower the retrieval.

- With open addressing, the load factor cannot exceed 1. With chaining, the load factor often exceeds 1.

HASH FUNCTIONS (CONT'D)

- A good hash function should:

  · *Minimize* collisions.

  · Be *easy* and *quick* to compute.

  · Distribute key values *evenly* in the hash table.

  · Use *all the information* provided in the key.

COMMON HASHING FUNCTIONS

1. **Division Remainder (using the table size as the divisor)**

- Computes hash value from key using the % operator.
- Prime numbers are better table size values.

COMMON HASHING FUNCTIONS (CONT'D)

2. **Truncation or Digit/Character Extraction**

- Works based on the distribution of digits or characters in the key.
- More evenly distributed digit positions are extracted and used for hashing purposes.
- For instance, students IDs or ISBN codes may contain common subsequences which may increase the likelihood of collision.
- Very fast but digits/characters distribution in keys may not be very even.

COMMON HASHING FUNCTIONS (CONT'D)

3. Folding

- It involves splitting keys into two or more parts and then combining the parts to form the hash addresses.
- To map the key 25936715 to a range between 0 and 9999, we can:
  - · split the number into two as 2593 and 6715 and
  - · add these two to obtain 9308 as the hash value.
- Very useful if we have keys that are very large.
- Fast and simple especially with bit patterns.
- A great advantage is ability to transform non-integer keys into integer values.

COMMON HASHING FUNCTIONS (CONT'D)

4. **Radix Conversion**

- Transforms a key into another number base to obtain the hash value.
- Typically use number base other than base 10 and base 2 to calculate the hash addresses.
- To map the key 55354 in the range 0 to 9999 using base 11 we have:

$$55354_{10} = 38652_{11}$$

- We may truncate the high-order 3 to yield 8652 as our hash address within 0 to 9999.

COMMON HASHING FUNCTIONS (CONT'D)

5. **Mid-Square**

- The key is squared and the middle part of the result taken as the hash value.
- To map the key 3121 into a hash table of size 1000, we square it $3121^2 = 9740641$ and extract 406 as the hash value.
- Works well if the keys do not contain a lot of leading or trailing zeros.
- Non-integer keys have to be preprocessed to obtain corresponding integer values.

SOME APPLICATIONS OF HASH TABLES

- **Database systems**: Specifically, those that require efficient random access. Generally, database systems try to optimize between two types of access methods: sequential and random. Hash tables are an important part of efficient random access because they provide a way to locate data in a constant amount of time.

- **Symbol tables**: The tables used by compilers to maintain information about symbols from a program. Compilers access information about symbols frequently. Therefore, it is important that symbol tables be implemented very efficiently.

- **Data dictionaries**: Data structures that support adding, deleting, and searching for data. Although the operations of a hash table and a data dictionary are similar, other data structures may be used to implement data dictionaries. Using a hash table is particularly efficient.

- **Network processing algorithms**: Hash tables are fundamental components of several network processing algorithms and applications, including route lookup, packet classification, and network monitoring.

- **Browser Cashes**: Hash tables are used to implement browser cashes.