04-630
Data Structures and Algorithms for Engineers

Lecture 16: Hashing

Agenda

Hashing

- Dictionaries
- Hashing
- Hash functions
- Collision resolution
- Complexity
- Applications

Recall: Containers vs. Dictionaries

- Containers are data structures that permit storage and retrieval of data items independent of content.
  - *e.g., stacks, queues.*

- Dictionaries are data structures that retrieve data based on key values (i.e., content).
  - e.g., hash tables.

Recall: Containers vs. Dictionaries

- Containers are distinguished by:
  - the particular retrieval order they support.

- In the case of stacks and queues, the retrieval order depends on the insertion order, i.e.,
  - Last-in, first-out (LIFO) and first-in, first-out (FIFO), respectively.

Recall: Containers vs. Dictionaries

Stack: supports retrieval by last-in, first-out (LIFO) order

– Push(x, S)	Insert item x at the **top** of a stack S
– Pop (S)	Return (and remove) the **top** item of a stack S

Recall: Containers vs. Dictionaries

Queue: support retrieval by first-in, first-out (FIFO) order

– Enqueue(x, Q)	Insert item x at the **back** of a queue Q
– Dequeue (Q)	Return (and remove) the the **front** item from a queue Q

Recall: Containers vs. Dictionaries

Dictionaries permits access to data items by content.

- You put an item into a dictionary
  so that you can find it when you need it

Containers vs. Dictionaries

Main dictionary operations are

- Search(D, k) Given a search key k, return a pointer to the element in dictionary D whose key value is k, if one exists
- Insert(D, x) Given a data item x, add it to the dictionary D
- Delete(D, x) Given a pointer to a given data item x in the dictionary D, remove it from D

Containers vs. Dictionaries

Some dictionary data structures also efficiently support other useful operations

- Max(D) Retrieve the item with the largest key from D
- Min(D) Retrieve the item with the smallest key from D

These operations allow the dictionary to serve as a priority queue!

Containers vs. Dictionaries

Some dictionary data structures also efficiently support other useful operations

– Predecessor(D, x)	Retrieve the item from D whose key is immediately before x in sorted order
– Successor(D, x)	Retrieve the item from D whose key is immediately after x in sorted order

These operations enable us to iterate through the elements of the data structure!

Recall: Containers and Dictionaries

- We have defined these container and dictionary operations in an **abstract** manner,

  without reference to their implementation or the implementation of the structure itself

- There are many implementation options

  - Unsorted arrays
  - Sorted arrays
  - Singly-linked lists
  - Doubly-linked lists
  - Binary search trees
  - Balanced binary search trees
  - Heaps
  - **Hash tables**
  - …

Applications requiring dictionaries

- Applications that require insert, search, and delete operations.
  - e.g., maintaining a symbol table of variable identifiers for a compiler.

Hashing

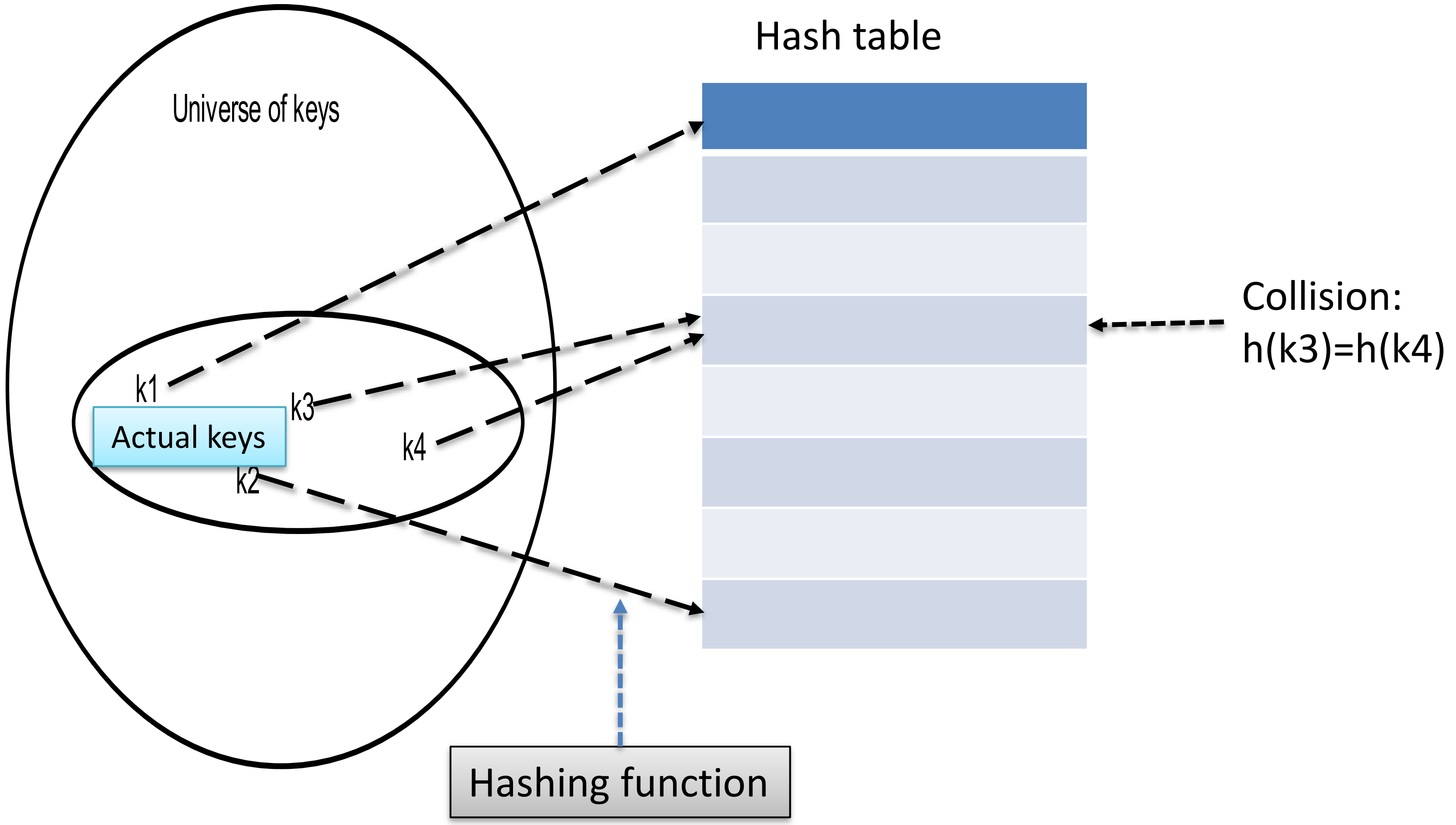
- Hash tables are a **very** practical way to maintain a dictionary.

- It takes a constant amount of time to look up an item in an array, **if** you have its index … O(1)

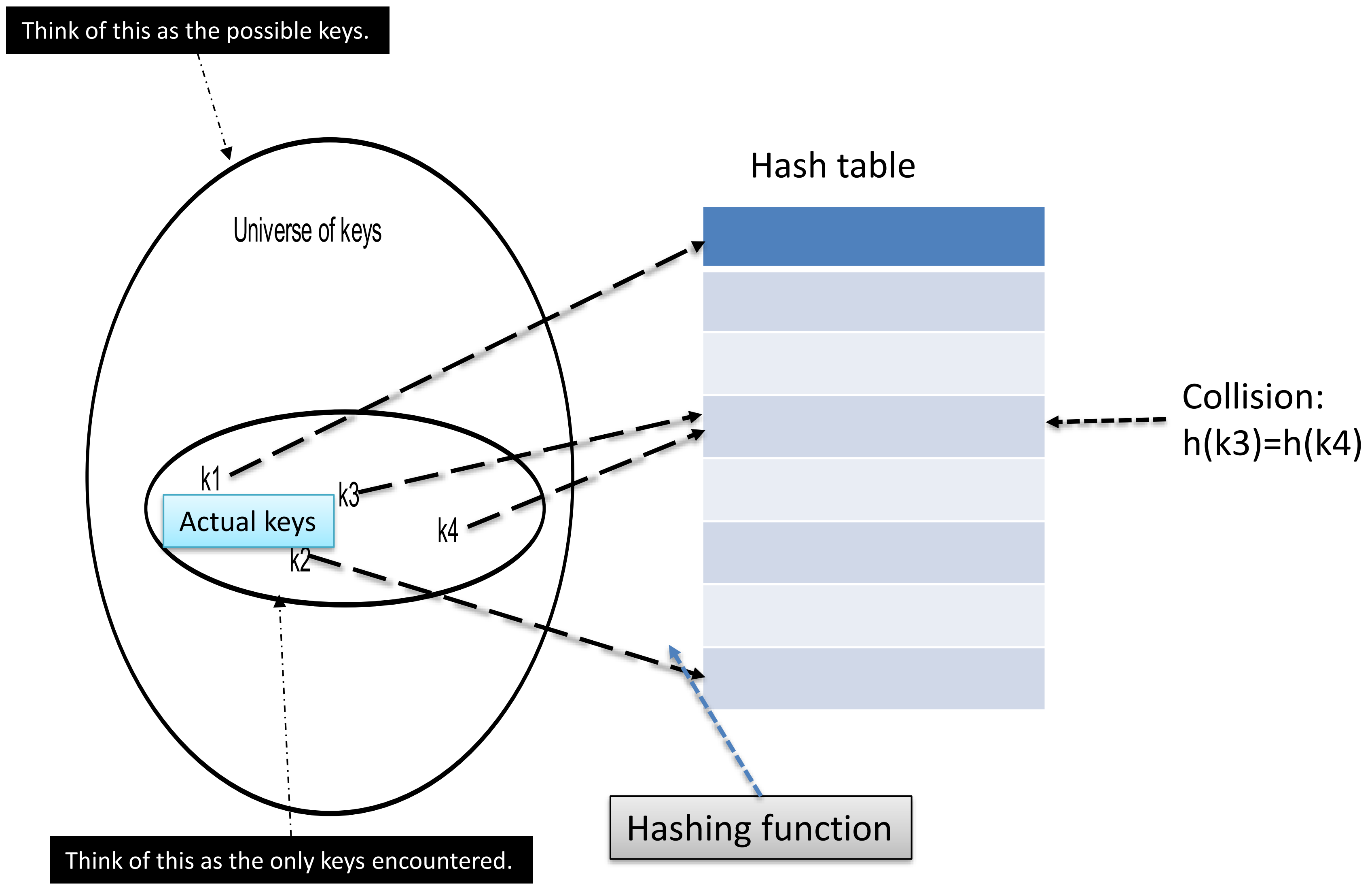
  - This is the ***property*** that ***hash tables*** exploit.
  - Think of **associative arrays** with the key used to compute the array index.

- Hash function: A hash function is a mathematical function to map keys to integers.
  - E.g. for hash function h, h: key->integer.
  - This integer is used as an index into an array.
  - We store our item at that position, at **slot h(key),** where h is the **hashing function.**

Hashing


Hash table
Universe of keys
k1
k3
Actual keys
k4
k2
Collision:
h(k3)=h(k4)
Hashing function

Hashing


Think of this as the possible keys.
Universe of keys
Hash table
k1
k3
Actual keys
k4
k2
Collision:
h(k3)=h(k4)
Hashing function
Think of this as the only keys encountered.

Hashing: Motivation

HandyParts company makes no more than 100 different parts.

But the parts all have four-digit numbers.

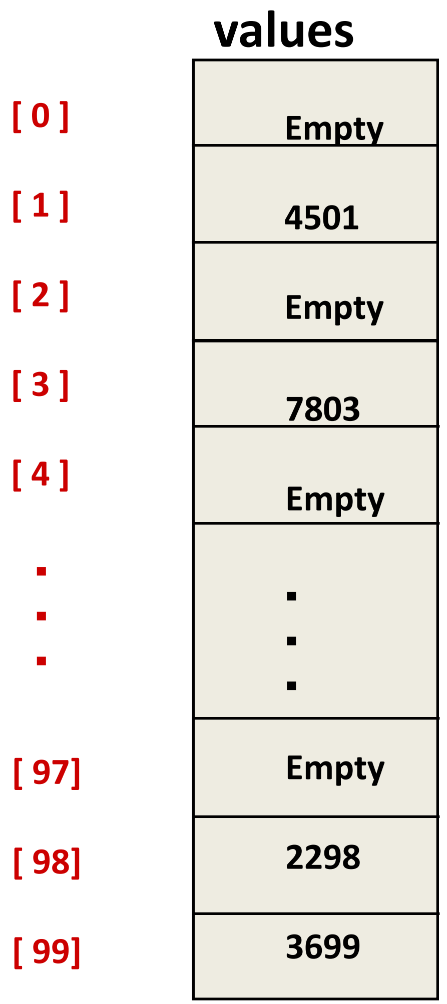
How do we store information about each part?

- Have an array with indices 0 to 9999

```
<parttype> handyparts[10000];
```

- Universe of keys?
- Actual keys?
- Any problem?

Hashing



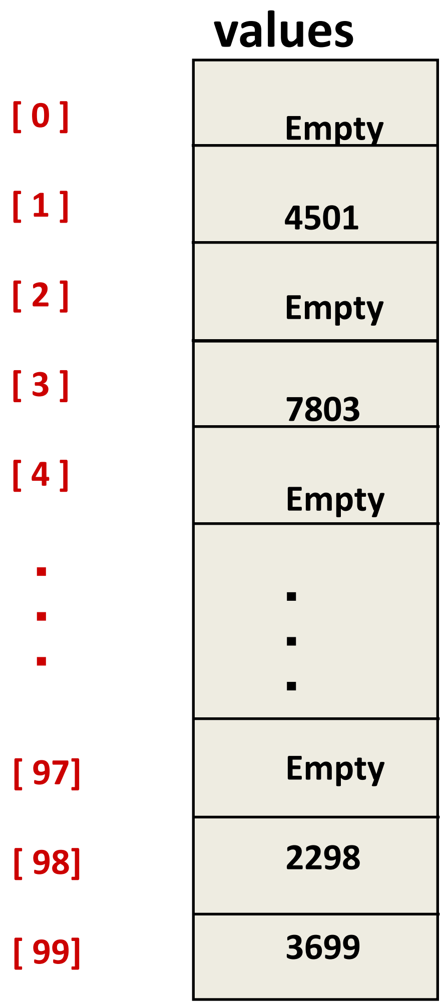
HandyParts company makes no more than 100 different parts.

But the parts all have four-digit numbers.

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

h(key) = partNum % 100

Placing Elements in the Array



Use the hash function

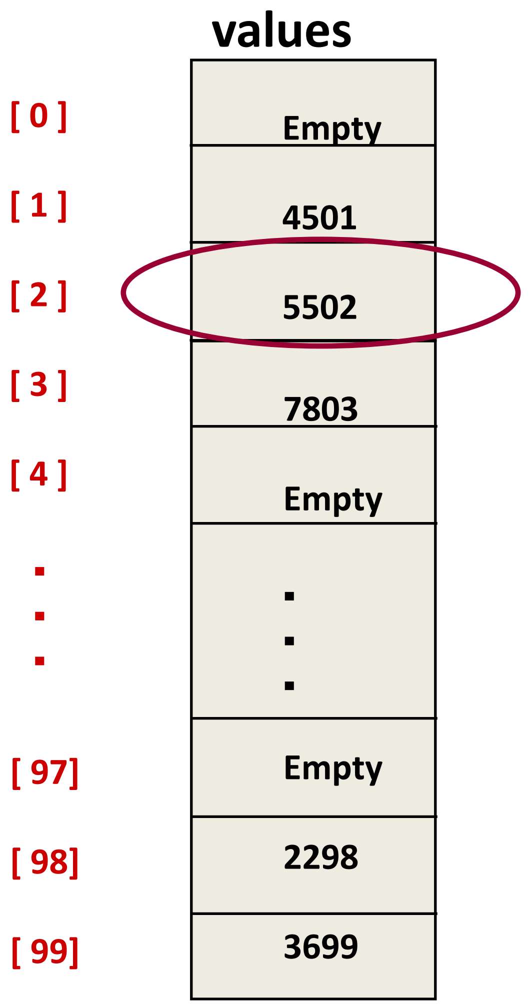
h(key) = partNum % 100

to place the element with

part number 5502 in the

array.

Placing Elements in the Array



Next place part number 6702 in the array.

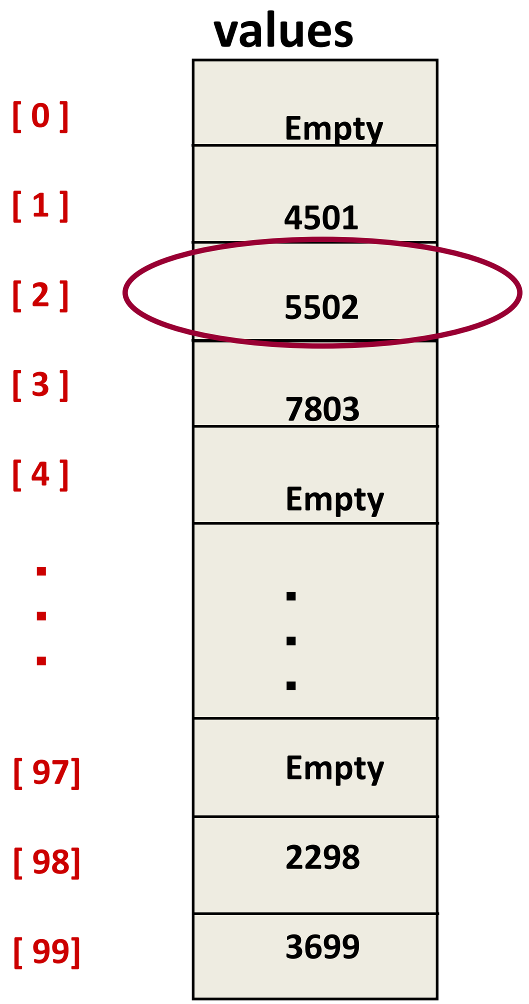
h(key) = partNum % 100

6702 % 100 = 2

But values[2] is already occupied.

COLLISION OCCURS

How to Resolve the Collision?

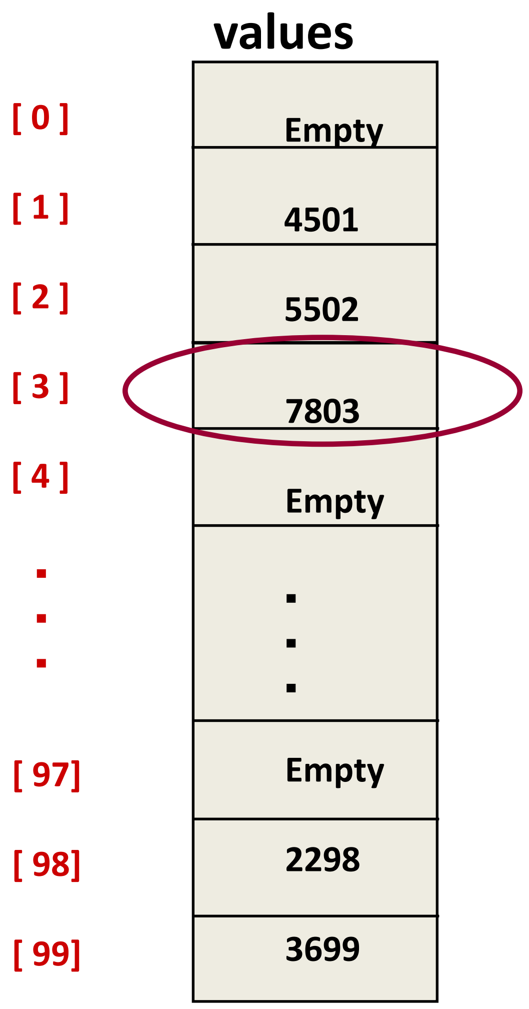


One way is by linear/sequential probing.
This uses the rehash function

(HashValue + 1) % 100

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

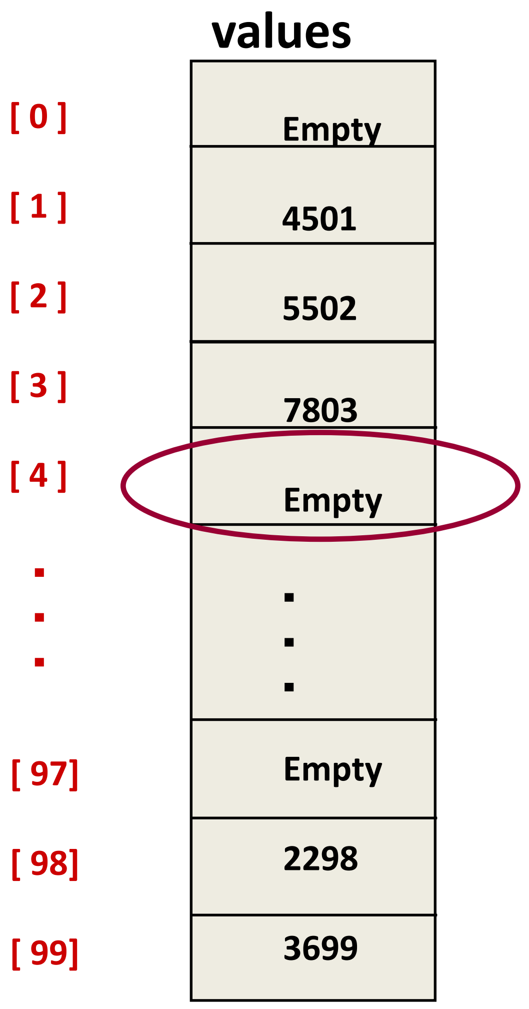
Resolving the Collision



Still looking for a place for 6702 using the function

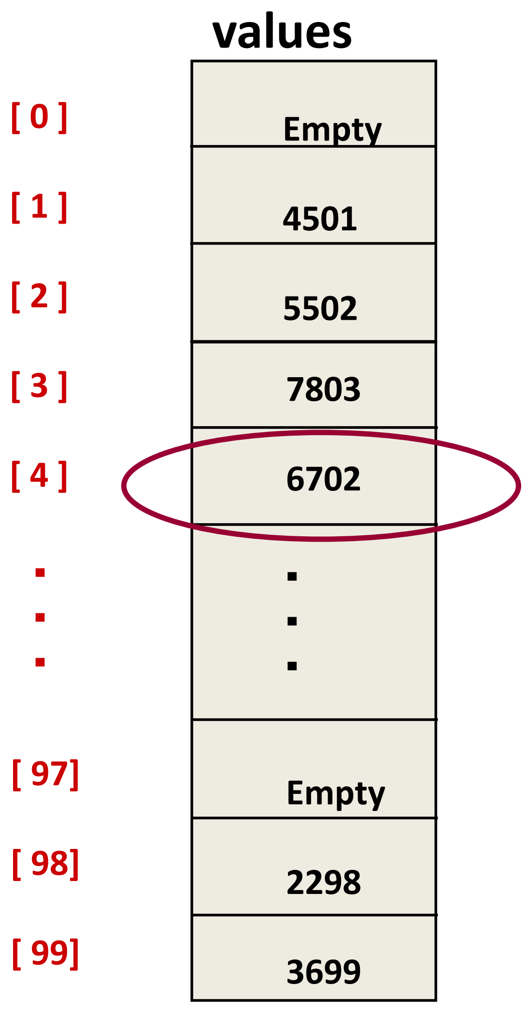
(HashValue + 1) % 100

Collision Resolved



Part 6702 can be placed at the location with index 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?

Hashing

- In general, the keys are not so conveniently defined (e.g., part numbers) and they have to be computed.
- Typically, they are some alphanumeric string S.
- The first step of a hash function is to map each key to a big integer.
- Let α be the size of the alphabet in which S is written.
- Let $char(c)$ be a function that maps each symbol of the alphabet to a unique integer from 0 to $\alpha - 1$

Hashing

- The hash function

$$H(S) = \sum_{i=0}^{|S|-1} \alpha^{|S|-(i+1)} \times char(s_i)$$

  maps each string to a unique (but large) integer by treating the characters of the string as "digits" in a base-α number system

- The result is unique identified numbers, but they are so large they will quickly exceed the number of slots m in the hash table

- We reduce this number to an integer between 0 and $m - 1$ by taking the remainder of $H(S) \bmod m$

Hashing

- If m, the size of the hash table, is selected well, the resulting hash value will be **fairly uniformly** distributed.

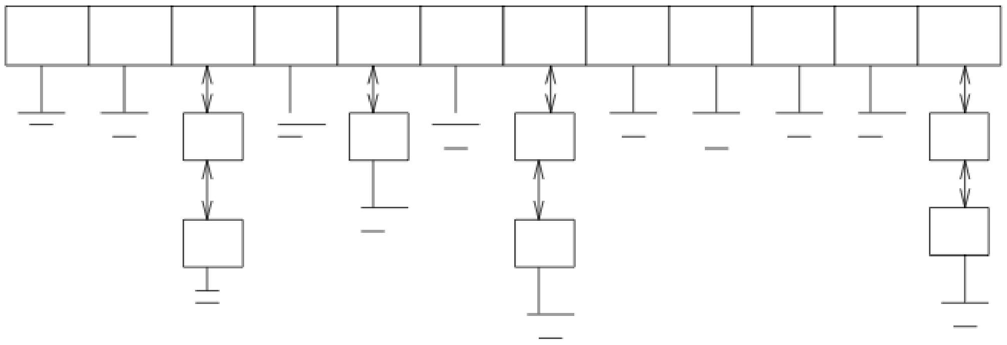
- Ideally, it is a ***large prime*** not too close to $2^i - 1$

Hashing

- Collisions
  - No matter how good the hash function is, there will sometimes be collisions: *two keys mapping to the same number/index.*
  - One approach to collision resolution:
  - a) ***open addressing***
    - Maintain hash table as an array of elements, each initialized to null.
    - On insertion, check to see if the desired position is empty
    - If so, insert it
    - If not, find some other place …
    - Simplest approach is ***sequential probing***: look for the next open spot in the table

Hashing

- Collisions
  - No matter how good the hash function is, there will sometimes be collisions: two keys mapping to the same number/index
  - Alternative approach:
  - b) ***chaining***
    - Represents the hash table as an array of m linked lists (each linked list is called a ***bucket***)
    - ith list will contain all items that hash to the value of i.
    - Search, insertion, and deletion reduce to corresponding problem in linked lists requiring access to the corresponding bucket.
    - If the n keys are ***distributed uniformly*** in the table, each list roughly contains n/m elements, making them constant size when m is approximately equal to n.



Hashing

- Complexity of operations in a hash table
  - Assuming chaining with doubly-linked lists
  - m-element hash table
  - n keys

	Hash table (expected)	Hash table (worst case)
Search(L, k)	$O(n/m)$	$O(n)$
Insert(L, x)	$O(1)$	$O(1)$
Delete(L, x)	$O(1)$	$O(1)$
Successor(L, x)	$O(n+m)$	$O(n+m)$
Predecessor(L, x)	$O(n+m)$	$O(n+m)$
Minimum(L)	$O(n+m)$	$O(n+m)$
Maximum(L)	$O(n+m)$	$O(n+m)$

Class Exercises-1

A. Understanding collision resolution:

1. Demonstrate what happens when we insert the keys 5; 28; 19; 15; 20; 33; 12; 17; 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be h(k)= k mod 9.

2. What about through open addressing using sequential probing?

Class Exercises-2

B. Optimizing performance: You hypothesize that you can obtain substantial performance gains by modifying the chaining scheme to keep each list in sorted order (assume doubly linked list for each list).

1. How does your modification affect the running time for:
   - searches (Search(L,k)),
   - insertions (Insert(L,x)), and
   - deletions (Delete(L,x))?
2. Should the hypothesis be accepted? Discuss your reasons.

Hashing: Applications

- A hash table is often the best data structure to maintain a dictionary.
- Also useful for other applications, not just dictionaries, e.g.:
  - Efficient string matching via hashing

    ***Problem* (substring pattern matching)**: given a text string *t* and a pattern string *p (**input**)*,
    does *t* contain the pattern *p* as a substring, and, if so, where?(**output**)

Hashing: application to string matching

- String matching
  - Simplest algorithm:
    - Overlay *p* in *t* at every position in the text
    - Check whether every pattern character matches the text character
    - Complexity $O(nm)$, where $n = |t|$ and $m = |p|$

Hashing: application to string matching

- String matching

  Rabin-Karp algorithm:

  - **Basic idea**: if two strings are identical, so are their hash values
  - If the two strings are different, the hash values are *almost certainly* different (may need to check, but not often)
  - With some clever computation of the hash function (to reduce complexity to constant) the algorithm will usually run in O*(n + m)*
- ***Implementation***: Left for you as an auxiliary exercise.

Hashing: other applications

- Related applications
  - *Is a given document different from all the rest in a large corpus?*
    - E.g., in web search
    - Only add a document to relevant documents if the hash value is different.
  - *Is part of this document plagiarized from a document in a large corpus?*
    - E.g., in plagiarism detection
    - Build a hashtable of all overlapping substrings of some length w (window length)
    - A match in hash codes could signify plagiarism.
  - *Can we confirm that a file has not changed?*
    - So, you submitted an assignment on 13th March but due to some problem I cannot assess it. I need you to resend it. How do I confirm that you did not change it over the last two days?

Further Reading

- *Chapter 11: Hash Tables* (Introduction to Algorithms, 3rd Edition, Thomas H. Cormen et al. (2009)
- *Section 3.7: Hashing and Strings* (The Algorithm Design Manual 2nd Edition: by Steven Skiena)

Acknowledgement

Adopted and Adapted from Material by:

David Vernon: www.vernon.eu

Augmented by material from:

The Algorithm Design Manual 2nd Edition: by Steven Skiena

Introduction to Algorithms, 3rd Edition, Thomas H. Cormen et al. (2009)