BACS2063 Data Structures and Algorithms

# HASHING

Chapter 10

# Learning Outcomes

At the end of this chapter, you should be able to

- Write application programs that uses the ADT Dictionary.
- Override the method **`hashCode`** for a class whose instances are to be used as search keys.
- Implement hash functions and collision-resolution schemes.
- Discuss the advantages and disadvantages of various hashing and collision resolution schemes.
- Describe clustering and the problem it causes.


# *Computing the Frequency of Words*

## Points to Consider #1: What ADT to use?

- *List* or *Ordered list*?
- *Array* or *linked implementation*?
- If ordered list, use *array*, *linked implementation* or *binary search tree*?
- *Dictionary*?

# Hash Codes

- The **`Object`** class has a method called **`hashCode`** which returns an integer hash code.
- If a class's instances are to be search keys, you should override **`hashCode`** to produce suitable hash codes.
- A hash code should depend on the entire search key.

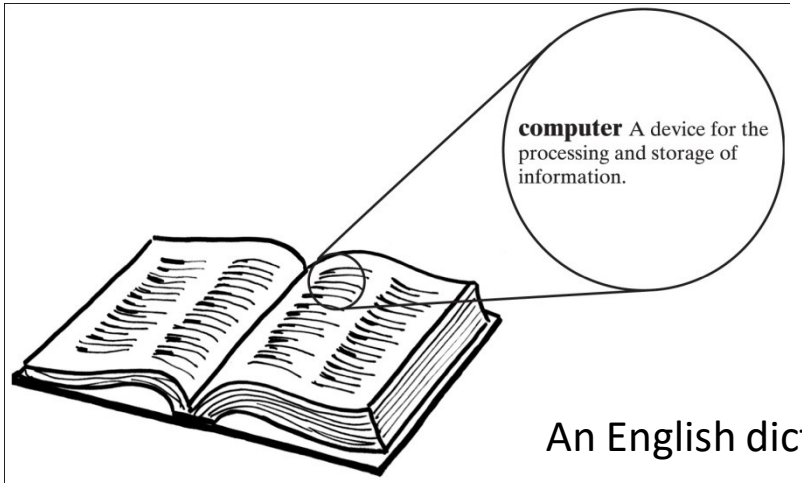

# *Computing the Frequency of Words*

**Points to Consider #2: What hash code to use?**

- We want to store words and update their count (frequency) based on the word itself (i.e. a string).
- Method: sum up each character's integer value. Options for a character's value:
  a) Assign the integers 1 through 26 to the letters 'A' through 'Z' and the integers 27 through 52 to the letters 'a' through 'z'?
  b) Use the character's Unicode integer?

# ADT Dictionary

- a.k.a. *map*, *table* or *associative array*
- Has entries that comprises 2 parts:
  - A search key, and
  - A value associated with the key




An English dictionary

- Organizes and identifies entries by their search keys

  ➔can retrieve or remove an entry from a dictionary given only the entry's search key

# Dictionary Applications

- A Directory of Telephone Numbers
- The Frequency of Words
- A Concordance of Words

# java.util.Map

- **java.util.Map<K, V>** is Java class library's interface for the ADT dictionary.
- This interface is implemented by various classes, e.g.:
  - **java.util.HashMap<K, V>**
  - **java.util.Hashtable<K, V>**
  - **java.util.TreeMap<K, V>**

# Sample code

In the **`Chapter10\map`** folder:

- **`Word.java`**
  - Consists of 2 fields:
    - **`word`** – also used as the search key.
    - **`count`** – keeps track of the frequency of this word.
    - The **`hashCode`** method has been overridden to simply return the sum of the Unicode values of all the characters in the word.
- **`FrequencyOfWords.java`**
  - Uses Java API's **`Map`** and **`HashMap`**

# Specifications for the ADT Dictionary

- Data
  - Pairs of objects (key, value)
  - Number of pairs in the collection
- Operations
  - add
  - remove
  - retrieve
  - contains
  - isFull
  - get number of entries
  - remove all entries

# Using the ADT Dictionary

- A directory of telephone numbers

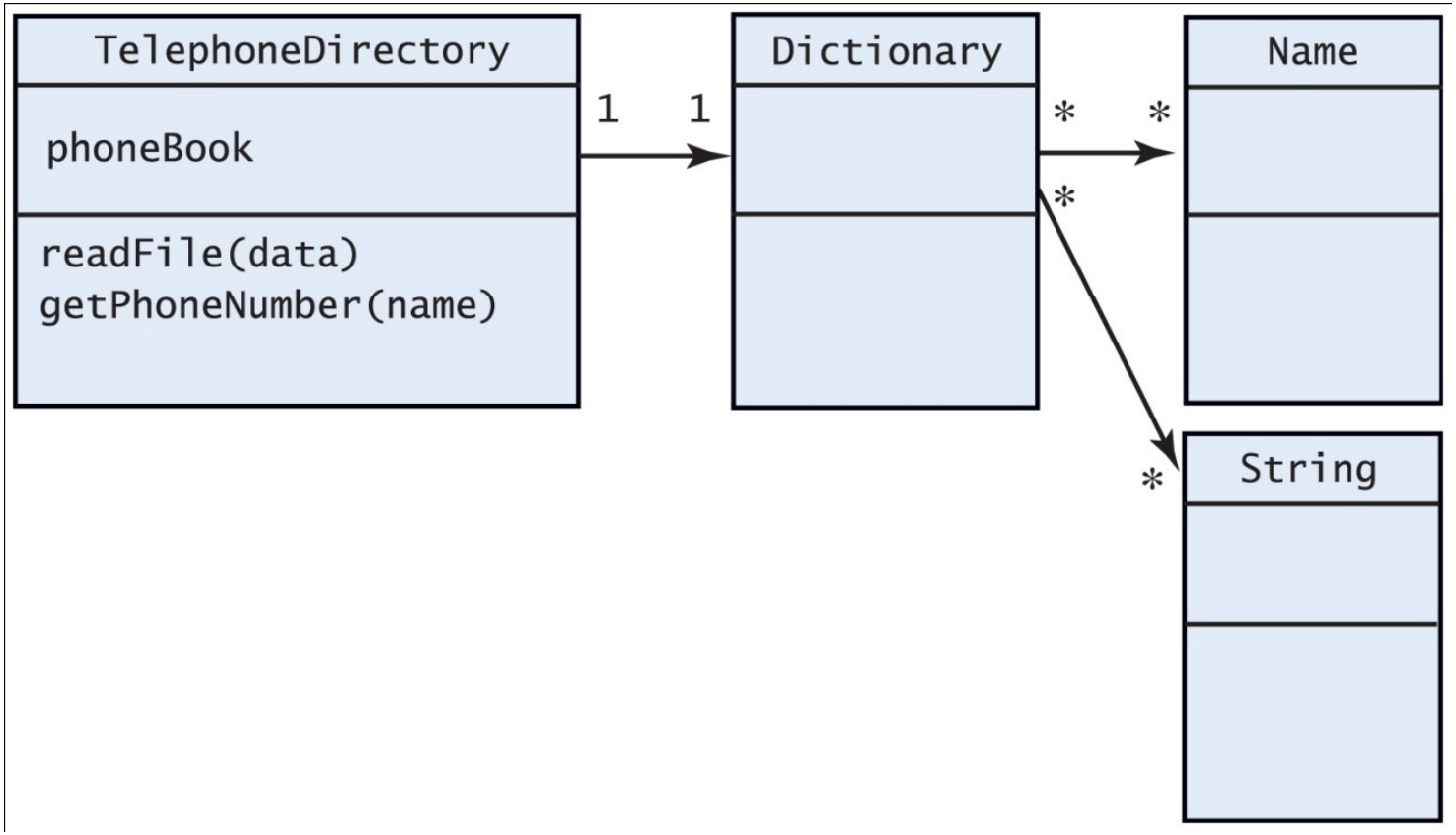



Fig 17-4 A class diagram for a telephone directory.

# ADT **Dictionary**

- Because searching databases is so widely used in computer applications, the dictionary is an important ADT
- Even more important is the efficiency of the searching process to locate an entry in a dictionary
- This chapter introduces a technique called *hashing* that ideally can result in O(1) search times.

# Hashing: Overview

- *Hashing* is a technique that determines an index or location for storage of an item in a data structure *without searching*
- A *hash function* maps the search key of an entry into a location that will contain the item
  - Returns the *hash index* of an entry in the hash table
  - An *hash table* is an array that contains the entries, as assigned by a hash function
  - The index / location computed by the hash function is known as the hash index
- A perfect hash function maps each search key into a different integer suitable as an index to the hash table.

# Use of Hash Functions

- In dictionary operations that add or retrieve entries.
- Example of algorithms for dictionary operations:

```
Algorithm add(key, value)
  index = h(key)
  hashTable[index] = value


Algorithm getValue(key)
  index = h(key)
  return hashTable[index]
```

# Two Steps of a Hash Function

1. Convert the search key into an integer called the hash code
2. Compress the hash code into the range of indices for the hash table

*E.g.*: Hash function algorithm

```
Algorithm getHashIndex(phoneNumber)
  i = last 4 digits of phoneNumber // Step 1
  return i % tableSize   // Step 2
```

# Hash Functions

- Characteristics of a good hash function
  - Minimize collisions
  - Distribute entries uniformly throughout the hash table
  - Fast to compute
- The search key used for computing the hash code can either be
  - A primitive type, or
  - An instance of a class

# Computing Hash Codes

- Recall: Java's base class **`Object`** has a method **`hashCode`** that returns an integer hash code
- The default implementation returns an **`int`** value based on the invoking object's memory address

  ➔ Not appropriate for hashing because equal but distinct objects will have different hash codes

- Thus, we have to override the **`hashCode`** method to map equal objects into the same location in a hash table for use in a dictionary implementation

# Guidelines for Defining **`hashCode`**

- If a class overrides the method **`equals`**, it should override **`hashCode`**
- If the method **`equals`** considers two objects equal, **`hashCode`** must return the same value for both objects
- If an object invokes **`hashCode`** more than once during execution of program on the same data, it must return the same hash code
- An object's hash code during one execution of a program can differ from its hash code during another execution of the same program

# Hash Codes for Strings

- Make use of each character's Unicode in the computation
- Methods
  1. Add up each character's Unicode value
  2. Multiply each character's Unicode value with the character's position within the string and then sum the products *(better*  *)*
  3. Horner's method (*even better*  ):

```
int hash = 0;
int n = s.length();
for (int i = 0; i < n; i++)
   hash = g * hash + s.charAt(i);
   // g is a positive constant
```


# Java's 8 primitive types

- `byte`
- `short`
- `int`
- `long`
- `float`
- `double`
- `char`
- `boolean`

# Hash Codes for Primitive Types

- Use the primitive-typed key itself
- Methods
  1. Manipulate internal binary representations
  2. Use folding
     - Break the key into groups of digits and then combine the groups using either addition or a bitwise operator such as **exclusive or**
     - The number of digits in a group should correspond to the size of the array

# Compressing a Hash Code

- Must compress the hash code so it fits into the index range
- Method for a code *c* is to compute *c* % *n*
  - *n* is a prime number (the size of the table)
  - If **`hashIndex`** is negative, add the size of the table
  - Index will then be between 0 and *n* – *1*

```
private int getHashIndex (K key)
{
    int hashIndex = key.hashCode () % hashTable.length;
    if (hashIndex < 0)
        hashIndex = hashIndex + hashTable.length;
    return hashIndex;
} // end getHashIndex
```

# Implementing Dictionary using Hashing

Sample code in **`Chapter10\version1`** folder:

- **`DictionaryInterface.java`**
  - Methods **`add`**, **`remove`**, **`getValue`**, **`contains`**, **`isEmpty`**, **`isFull`**, **`getSize`**, **`clear`**
- **`HashedDictionary.java`**
  - Implements interface **`DictionaryInterface`**
  - Note inner class **`TableEntry<S, T>`** for representing a table entry
- **`HashedDictionaryDriver.java`**
- **`Student.java`**
  - Returns the student id as the hash code value


# Emergency IS

- In HigglyTown, everyone's telephone number begins with 555, *e.g.* 555-1234.
- They require an emergency information system which enables them to retrieve the street address associated with each telephone number.

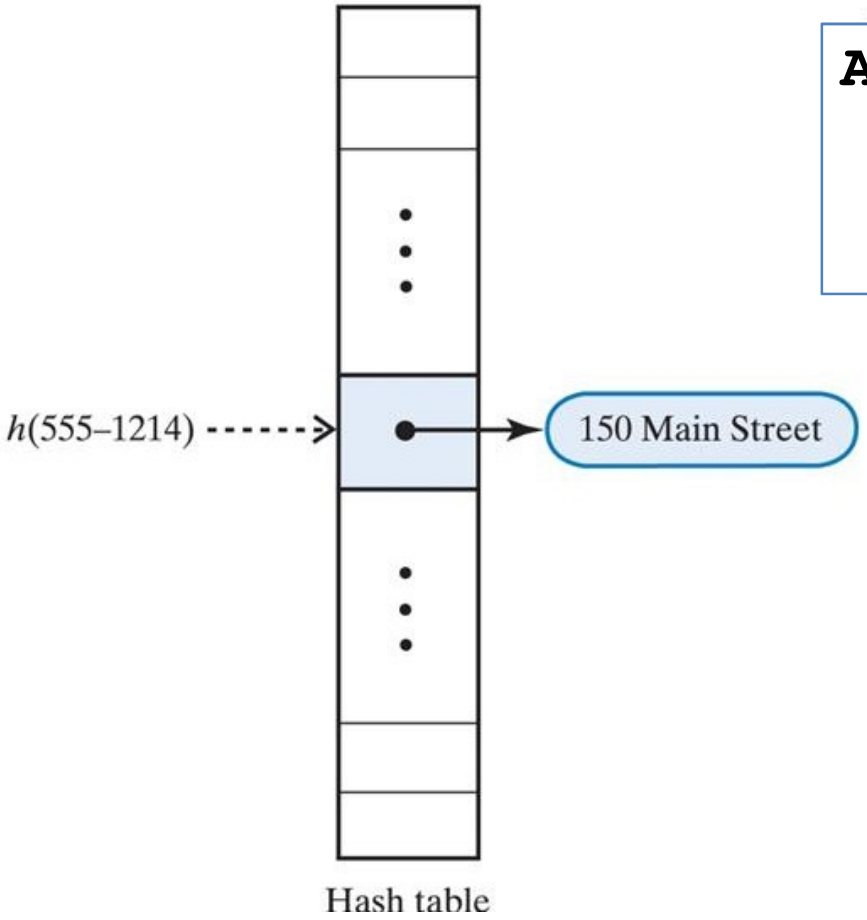

# Emergency IS – Hash Function

- We will use the hash function *h* which converts a telephone number to its last four digits. *E.g.*,

  $h(555\text{-}1214) = 1214$

- In the hash table `hashTable`, we would place a reference to the street address associated with this telephone number in `hashTable[1214]`.
- If the cost of evaluating the hash function is low, adding an entry to the array hashTable is an O(1) operation.

*h*(555-1214) = 1214, the street address for this phone number is added to the hash table




```
Algorithm add(key, value)
  index = h(key)
  hashTable[index] = value
```

Fig. 21-1 A hash function indexes its hash table

# Retrieving from the hash table

- To retrieve the street address associated with the number 555-1214, we once again compute *h*(555-1214) and use the result to index `hashTable`. Thus, from `hashTable[1214]`, we get the desired street address.

```
Algorithm getValue(key)
  index = h(key)
  return hashTable[index]
```

# Will these algorithms always work?

- We can make them work if we know all the possible search keys.
- In this case study, the search keys range from

  555-0000 to 555-9999, so the hash function will produce indices from 0 to 9999.
- If the array hashTable has 10,000 elements, each telephone number will correspond to one unique element in hashTable. That element references the appropriate street address.
- This scenario describes the ideal case for hashing, and the hash function here is a **perfect hash function.**


# Hash Table Size

- If HigglyTown required only 700 telephone numbers, most of the 10,000-location hash table would be unused. We would waste most of the space allocated to the hash table.
- If the 700 numbers were not sequential, we would need a different hash function if we wanted to use a smaller hash table.


# Two Steps of a Hash Function

1. Convert the search key into an integer called the hash code
2. Compress the hash code into the range of indices for the hash table


# Convert the search key into an integer called the hash code

- The hash function *h* converts a telephone number to its last four digits. *E.g.*,

*hashCode* = *h*(555-1264) = 1264


# Compress the hash code into the range of indices for the hash table

- *E.g.*, if the size of the hash table was 101,

$$index = hashCode \% 101$$
$$= 1264 \% 101$$
$$= 52$$


# Algorithm for Hash Function

```
Algorithm getHashIndex(phoneNumber)
  i = last 4 digits of phoneNumber  // Step 1
  return i % tableSize              // Step 2
```

# Collision


# The hash function gives two different entries the same hash index?


# Consider:

- `getHashIndex("555-1264")` and `getHashIndex("555-8132")` each map into 52.
- If we have already stored the street address for 555-1214 in `hashTable[52]`, what will we do with the address for 555-8132?
- Handling such collisions is called collision resolution.

# Collision Caused by Hash Function

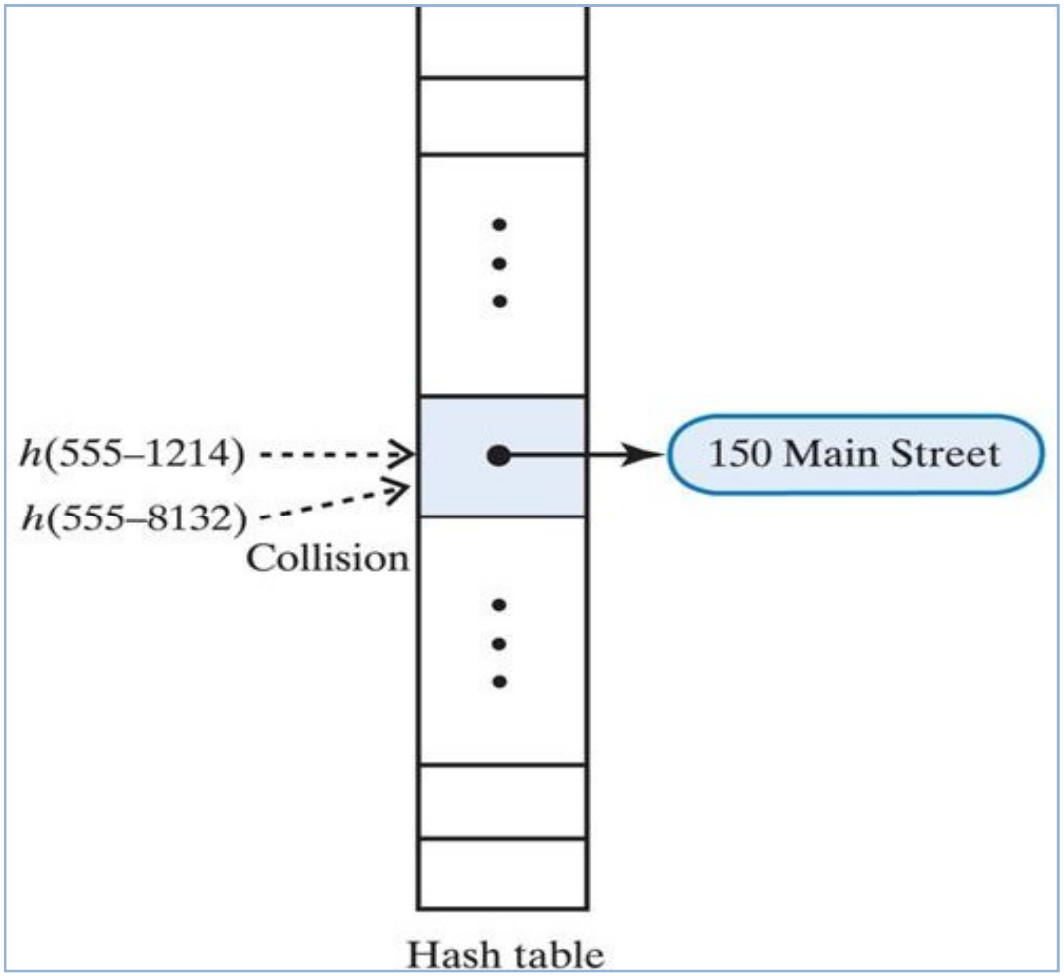



Fig. 18-2 A collision caused by the hash function *h*

# Collisions

- Typical hash functions are not perfect
  - They can allow more than one search key to map into a single index

    ➔ This is known as a *collision*
- A collision occurs when the hash function maps more than one item into the same array location
  - Refer to **`HashedDictionaryDriver.java`** when the last 4 entries are added


# the hash function returns location already used in the table

1. Use another location in the table
2. Change the structure of the hash table so that each array location can represent multiple values.

# Resolving Collisions

Two general approaches:

1. Open addressing
   a) Linear probing
   b) Quadratic probing
   c) Double hashing
2. Separate Chaining

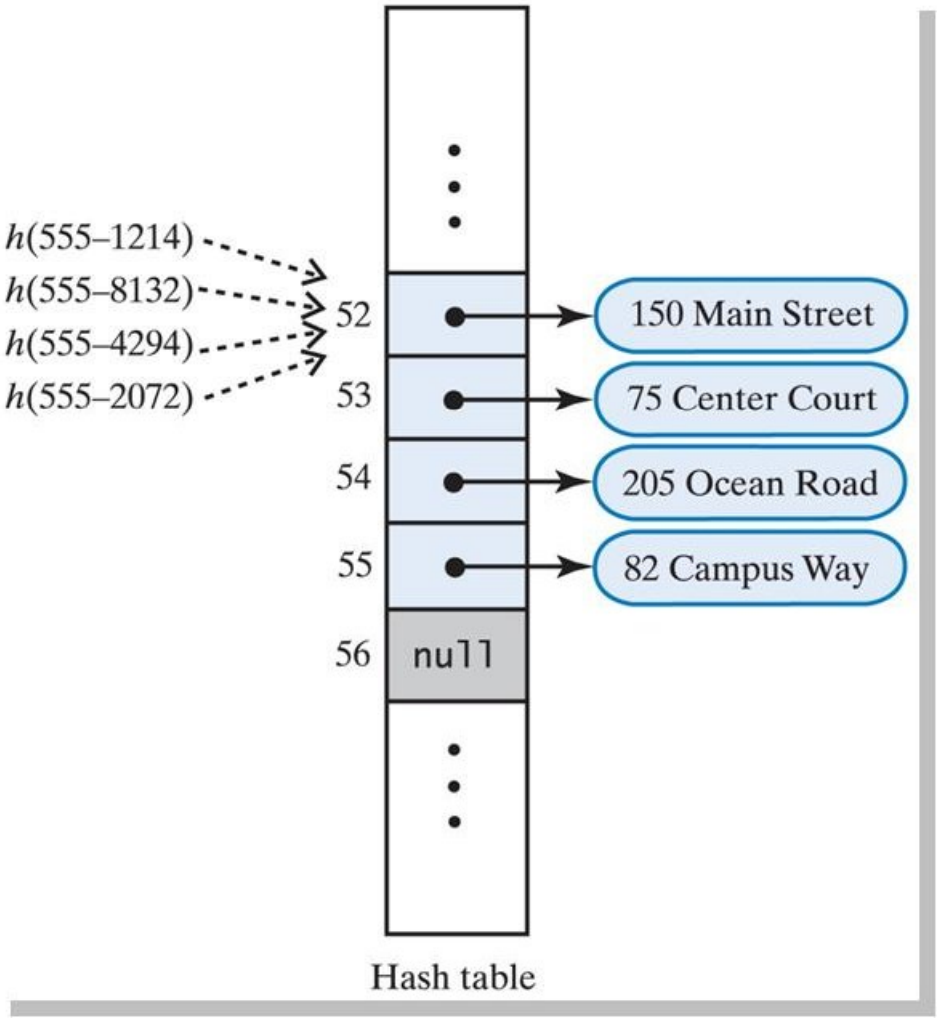
# Open Addressing with Linear Probing

- *Open addressing* scheme locates alternate location *within the hash table*
  - New location must be open, available
- Linear Probing
  - If collision occurs at hashTable[k], look successively at location k + 1, k + 2, …
  - Searches the hash table sequentially for vacant cells, incrementing the index until an empty cell is found
  - When probing reaches the end of the table, it continues at the table's beginning

# Linear Probing Example 1

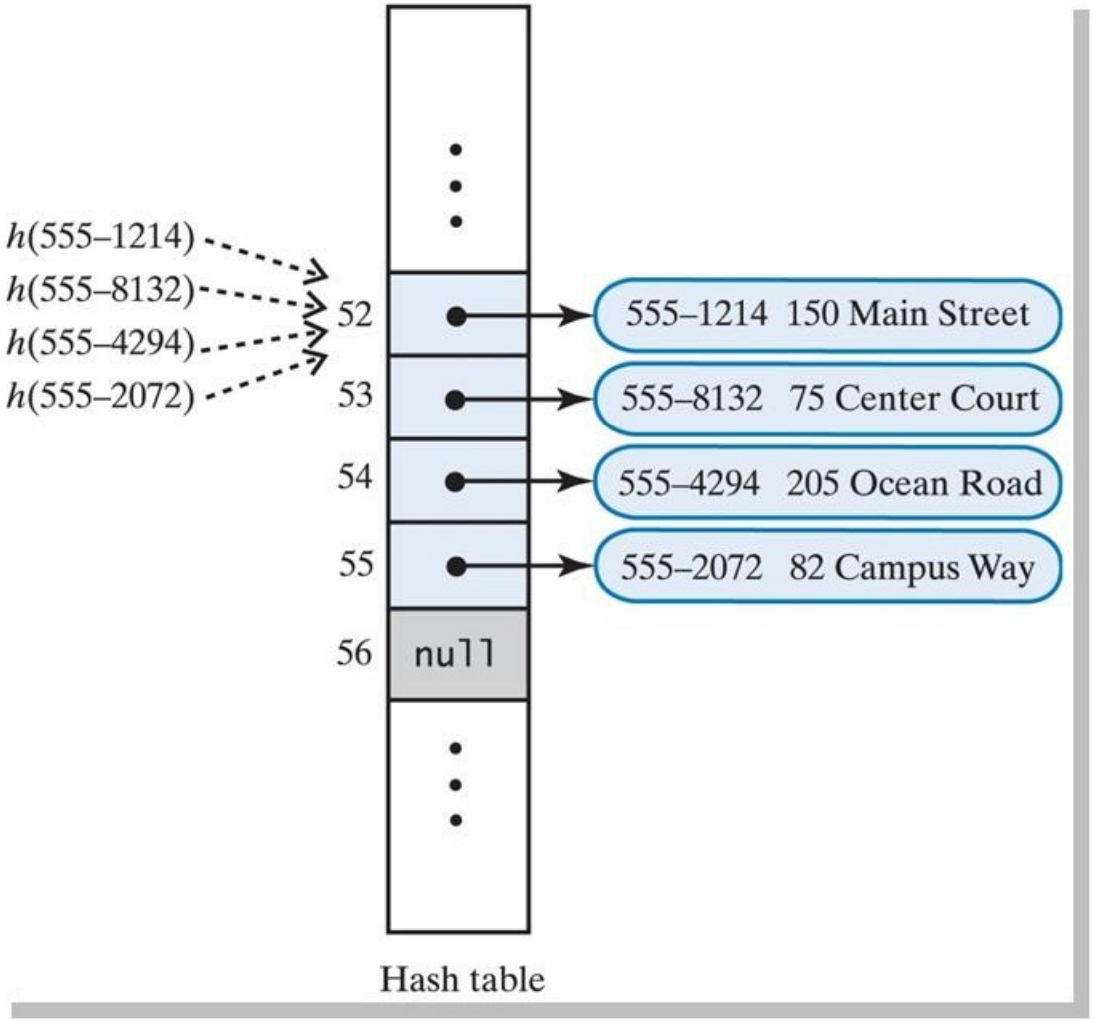
Fig. 21-3 After adding 4 entries whose search keys hash to the same index.

- Problem: During retrieval, cannot tell which address is the right one for the given phone number.
- Solution: Package a search key with its value.



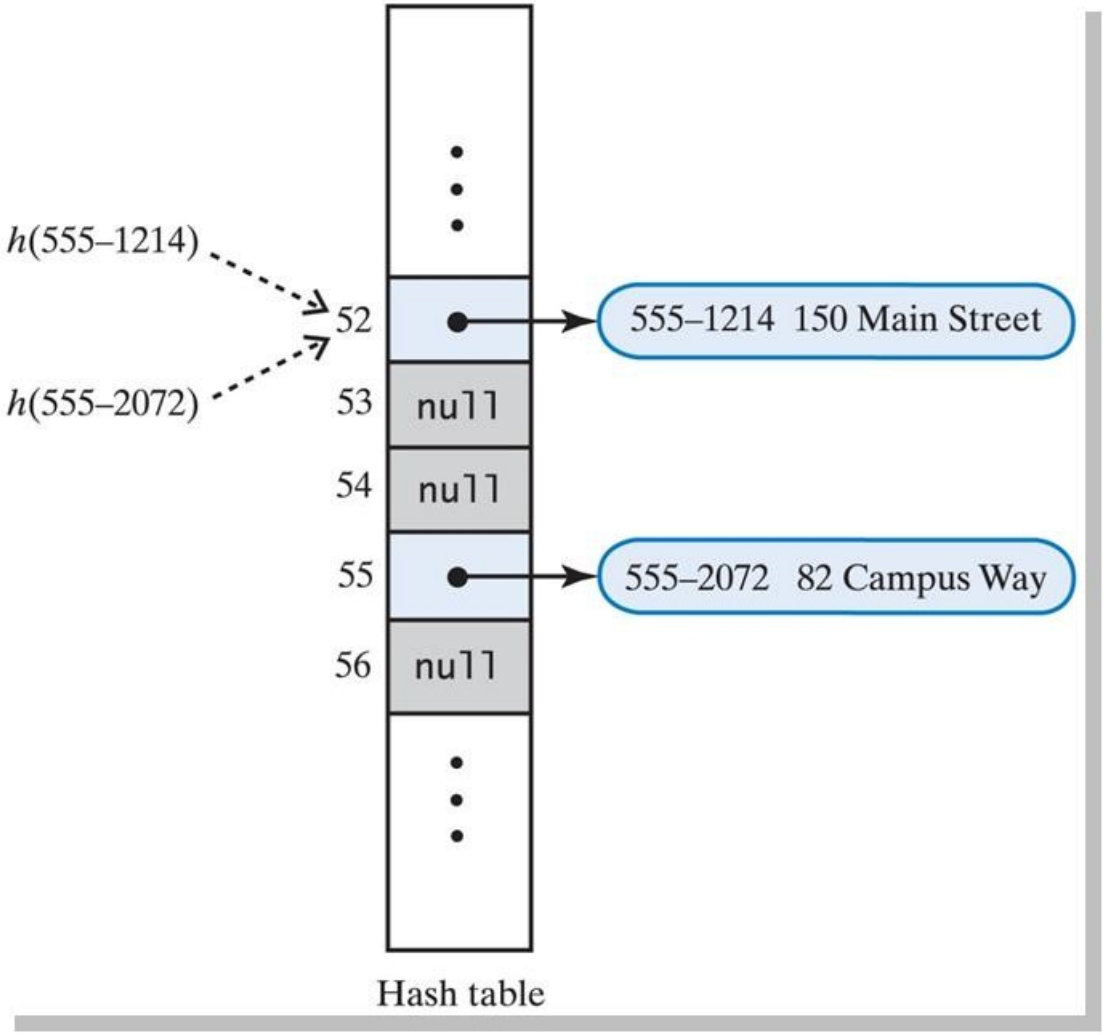
# Linear Probing Example 2

Fig. 18-4 A revision of the hash table shown in 18-3 when linear probing resolves collisions; each entry contains a search key and its associated value



# Linear Probing Removals

Fig. 18-5 A hash table if **`remove`** used **`null`** to remove entries.



# Need to distinguish 3 kinds of locations

- *Occupied*
  - The location references an entry in the dictionary
- *Empty*
  - The location contains **`null`**
  - Indicates a location that has never been used
- *Available*
  - The location's entry was removed from the dictionary

# For open addressing,


do we remove an entry by placing it in an "*Available*" state instead of setting its table location to **`null`**?


To avoid premature termination of the probe during retrieval or future removals.

# Dictionary Operations that Require Searching

- To retrieve an entry
  - Search the probe sequence for the key
  - Examine entries that are present, ignore locations in *available* state
  - Stop search when key is found or **`null`** reached
- To remove an entry
  - Search the probe sequence same as for retrieval
  - If key is found, mark location as available
- To add an entry
  - Search probe sequence same as for retrieval
  - *Note first available slot*
  - Use available slot if the key is not found

# Linear Probe Sequence Example 1

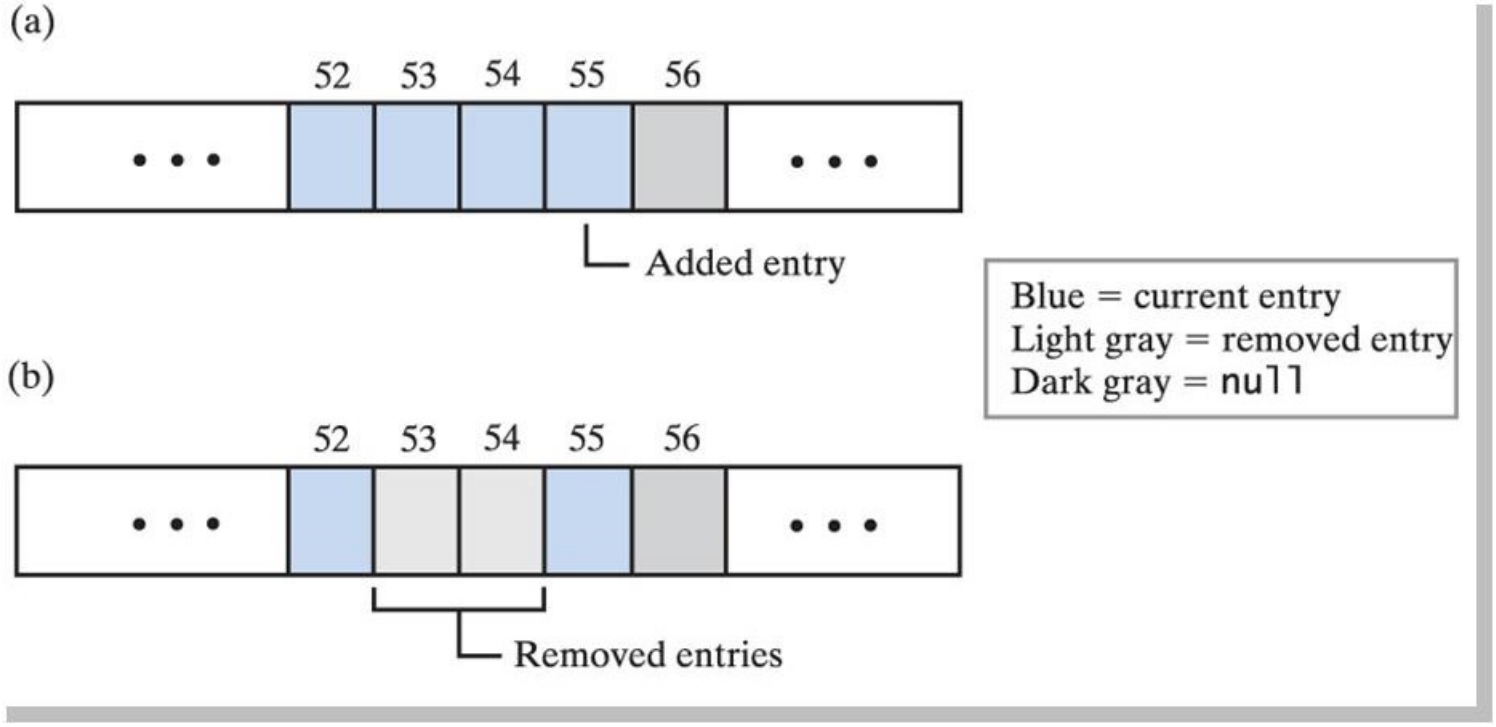



Fig. 21-6 A linear probe sequence (a) after adding an entry; (b) after removing two entries;

# Linear Probe Sequence Example 2

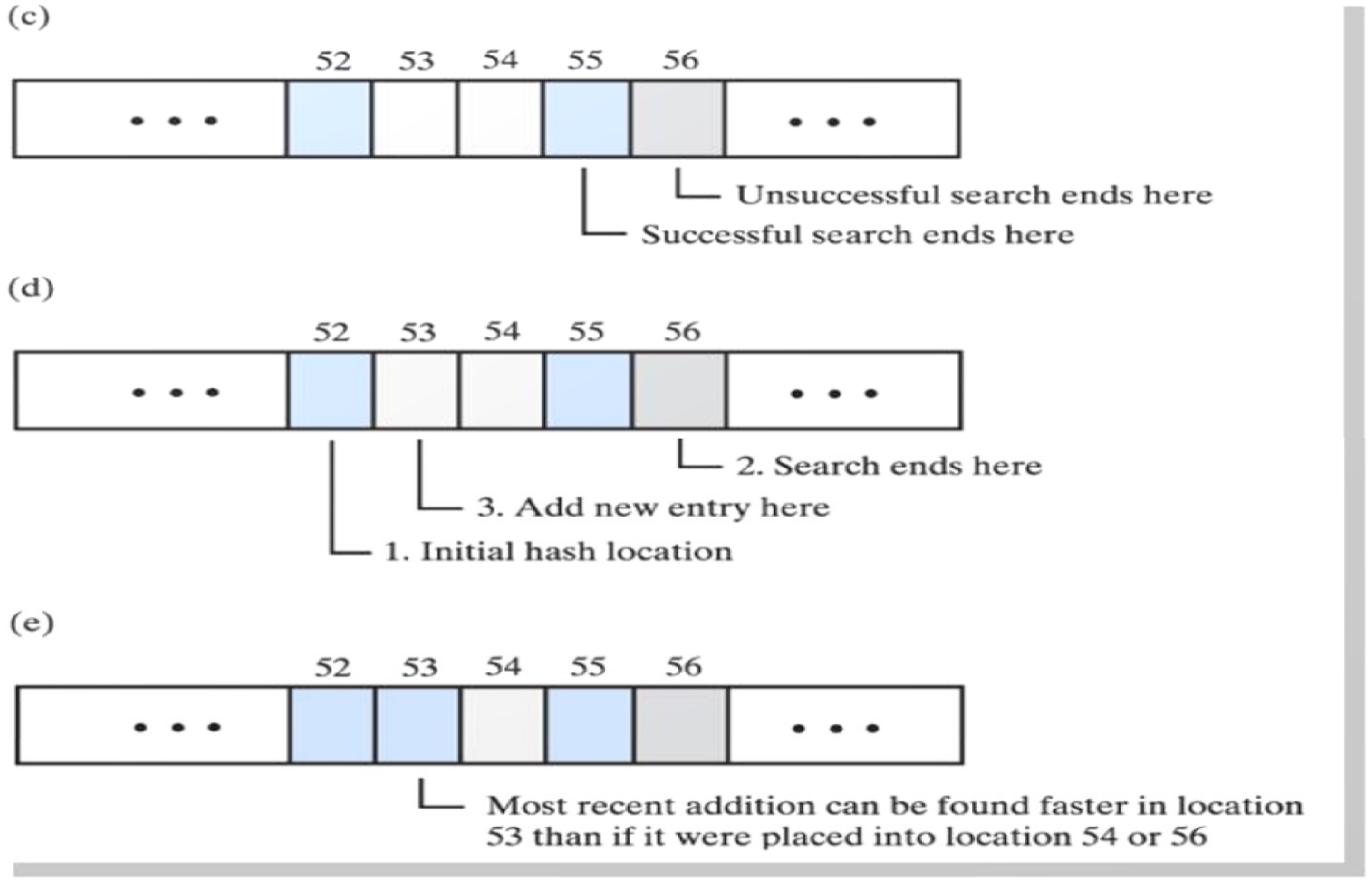



Fig. 18-6 A linear probe sequence (c) after a search; (d) during the search while adding an entry; (e) after an addition to a formerly occupied location.

# Implementing Linear Probing

Sample code in **`Chapter10\version2`** folder:

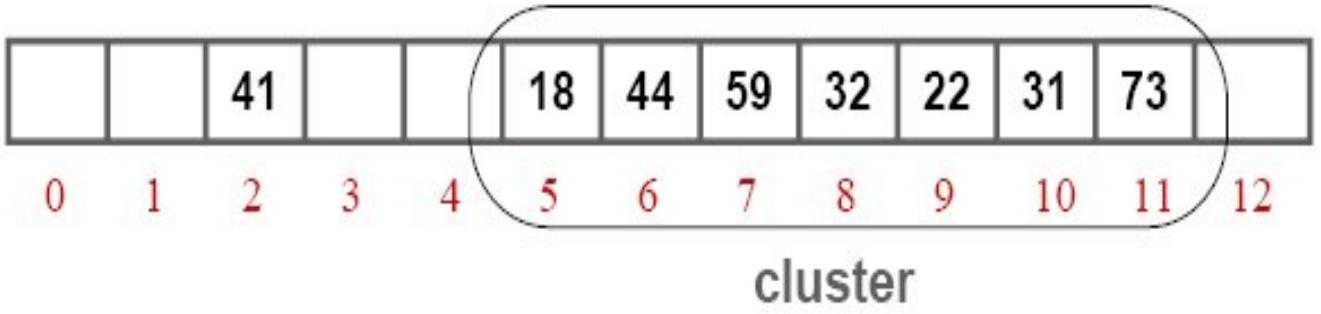
- **`HashedDictionary.java`**
  Note private methods:
  - **`probe`**, **`locate`**, **`isHashTableTooFull`**, **`getNextPrime`**, **`isPrime`**
- **`HashedDictionaryDriver.java`**
- **`Student.java`**
- **`Name.java`**
  - **`hashCode`** returns the sum of each character's Unicode value in the first name
- **`Driver.java`**

# Linear Probing Advantage

- It can reach every location in the hash table.
  - This property is important since it guarantees the success of the add operation when the hash table is not full.

# Linear Probing Disadvantages

- Linear probing is subject to primary clustering
  - A cluster is a group of consecutive table locations that are occupied.
- Primary clusters can combine to form larger clusters.
  - This leads to long probe sequences and hence deterioration in hash table efficiency.



# Quadratic Probing

- Avoids primary clustering by *changing the probe sequence*
  - Given search key *k*, the step is the square of the step number
  - Probe to $k + 1$, $k + 2^2$, $k + 3^2$, … $k + n^2$
  - More widely separated cells are probed
- Guarantees a successful add operation if
  - The hash table is at most half full and
  - The table size is a prime number

# Open Addressing, Quadratic Probing

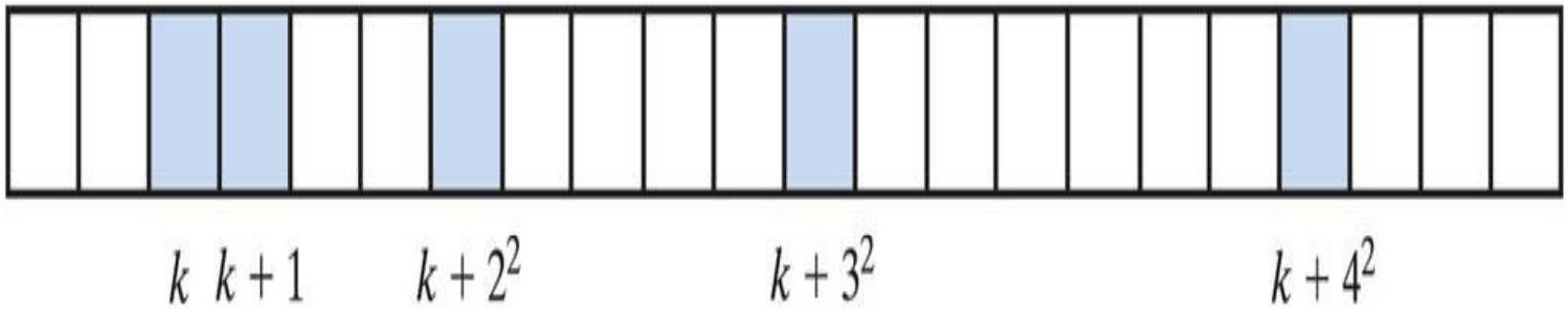



Fig. 18-7 A probe sequence of length five using quadratic probing.

# Quadratic Probing Advantage

- Eliminates primary clustering

# Quadratic Probing Disadvantages

- Requires more effort to compute the indices for the probe sequence than does linear probing.
- Entries that collide with an existing table entry use the same probe sequence, thereby increasing its length ➔ secondary clustering
  - Usually not a serious problem, but it increases search times

# Double Hashing: Overview

- Uses 2 hash functions
- Example:
  - $\mathbf{h_1(key) = key \% 7}$
  - $\mathbf{h_2(key) = 5 - key \% 5}$
- Method
  - Searches the hash table starting from the location that the first hash function $\mathbf{h_1(key)}$ determines
  - If a collision occurs, it considers every $n^{th}$ location from the original hash index, where n is determined from a second hash function $\mathbf{h_2(key)}$

# Open Addressing with Double Hashing

- Resolves collision by examining locations
  - At original hash index
  - Plus an increment determined by 2nd function
- Second hash function
  - Different from first
  - Depends on search key
  - Returns nonzero value

# Double Hashing Advantage

- Reaches every location in hash table if table size is prime
- Avoids both primary and secondary clustering

# Double Hashing Example

- Example: table size is 7, search key is 16:
  - $h_1$(key) = key % 7 = 16 % 7 = **2**
  - $h_2$(key) = 5 - key % 5 = 5 - (16 % 5) = 5 - 1 = **4**
- The probe sequence start at 2, probe locations at increments of 4: 2, 6, 3, 0, 4, 1, 5, 2, …

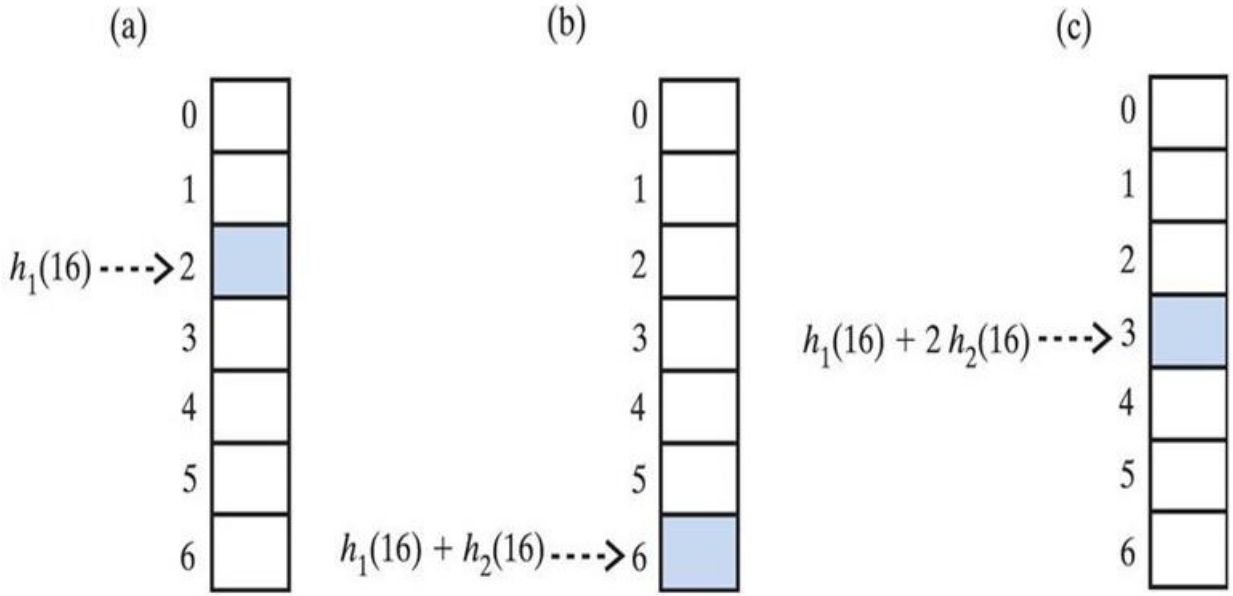



Fig. 18-8 The first three locations in a probe sequence generated by double hashing for the search key.

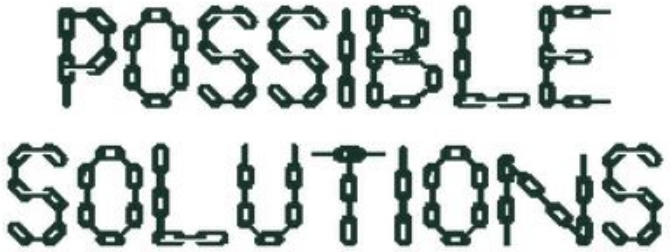

# Open Addressing

1. Linear probing
2. Quadratic probing
3. Double hashing

# Open Addressing


- Frequent additions and removals can cause every location in the hash table to reference either a current entry (*occupied* state) or a former entry (*available* state)
  - *i.e.*, a hash table might have no locations that contains `null`, regardless of the number of entries actually in the dictionary
  - If this happens, the approach to searching a probe sequence will not work ➔ every unsuccessful search can end only after considering every location in the hash table

# Open Addressing




- Rehashing
- Separate chaining

# Rehashing

- Expands the hash table to a size that is both prime and at least double its current size
- The hash function depends on the size of the table ➔ cannot copy elements from the old array and put them into the same position in the new array
  - Need to apply the revised hash function (*i.e.* with the new table size) to each entry to determine its proper position in the new table
  - However, doing so can lead to collisions. Thus, to ensure that collisions are resolved, use the method **`add`** to add the existing entries to the new, larger hash table
- Refer to method **`rehash`** in **`Chapter10\version2\HashedDictionary.java`**

# Separate Chaining

- Also known *close addressing*
- Alter the structure of the hash table so that each location can represent multiple values
  - Each location called a bucket
- Bucket can be a(n)
  - List
  - Sorted list
  - Chain of linked nodes
  - Array

# Separate Chaining Advantages

- No need to search for empty cells
- Simple and efficient
- Separate chaining may require lesser number of searches to find the match.
- Reduce the complexity in deletion
- Table size is not a prime number
- Arrays (buckets) can be used at each location in a hash table instead of a linked list

# Separate Chaining Implementation

- Each location in the hash table is a head reference to a chain of linked nodes that make up the bucket. Each node contains references to a search key, to the key's associated value and to the next node in the chain.

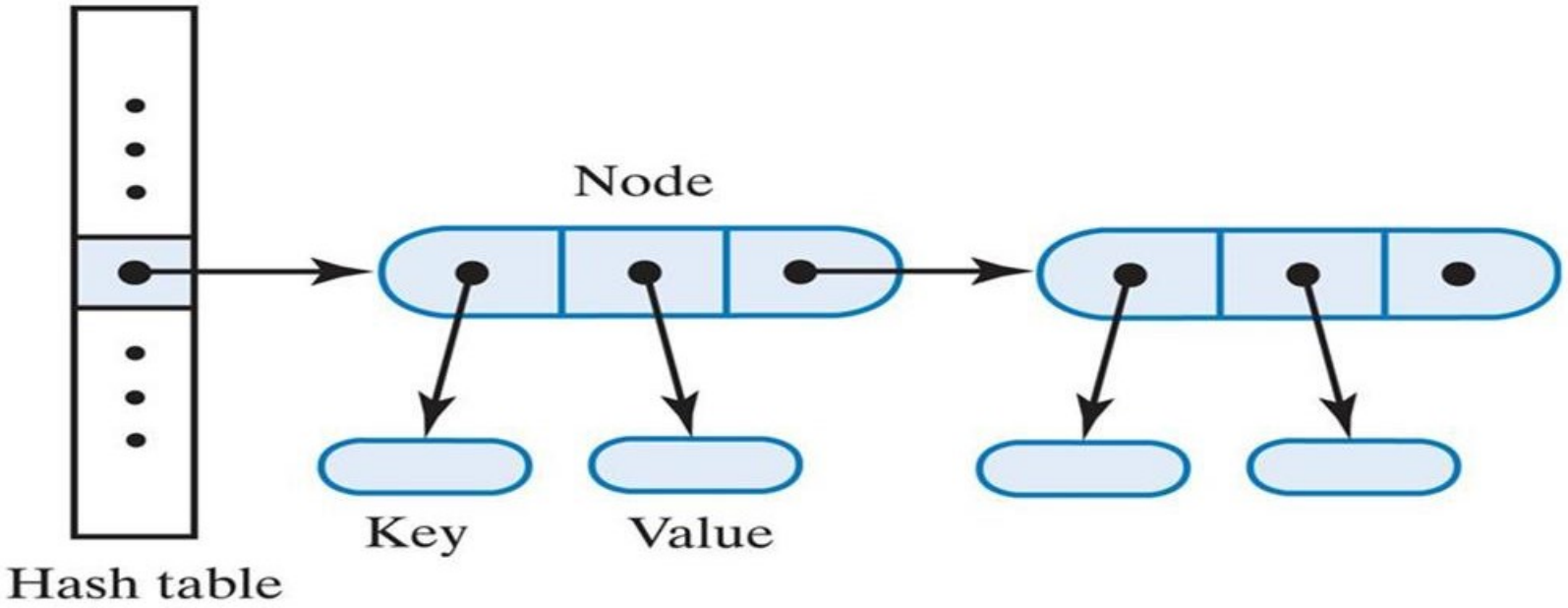



Fig. 18-9 A hash table for use with separate chaining; each bucket is a chain of linked nodes.

# Separate Chaining – Duplicate Keys, Unsorted Chain

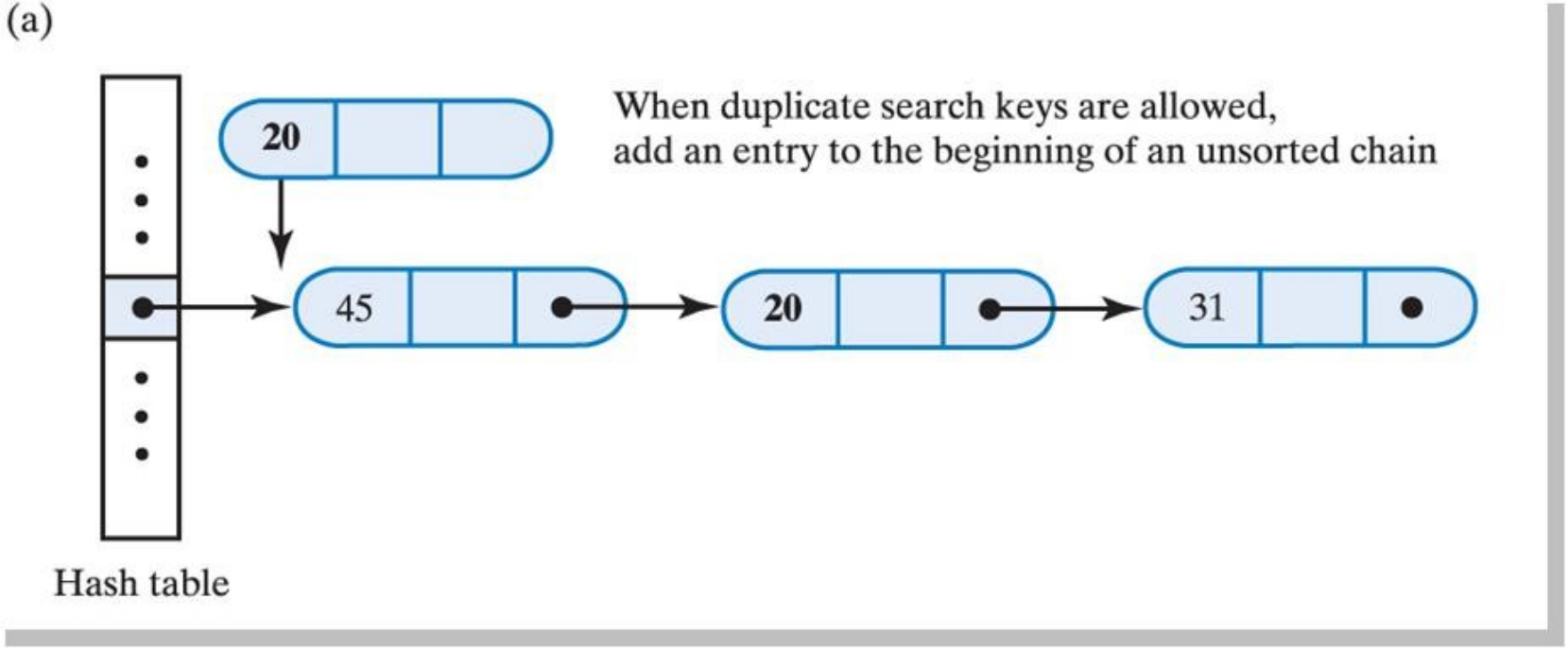



Fig. 18-10 Where new entry is inserted into linked bucket when integer search keys are (a) duplicate and unsorted;

# Separate Chaining – Distinct Keys, Unsorted Chain

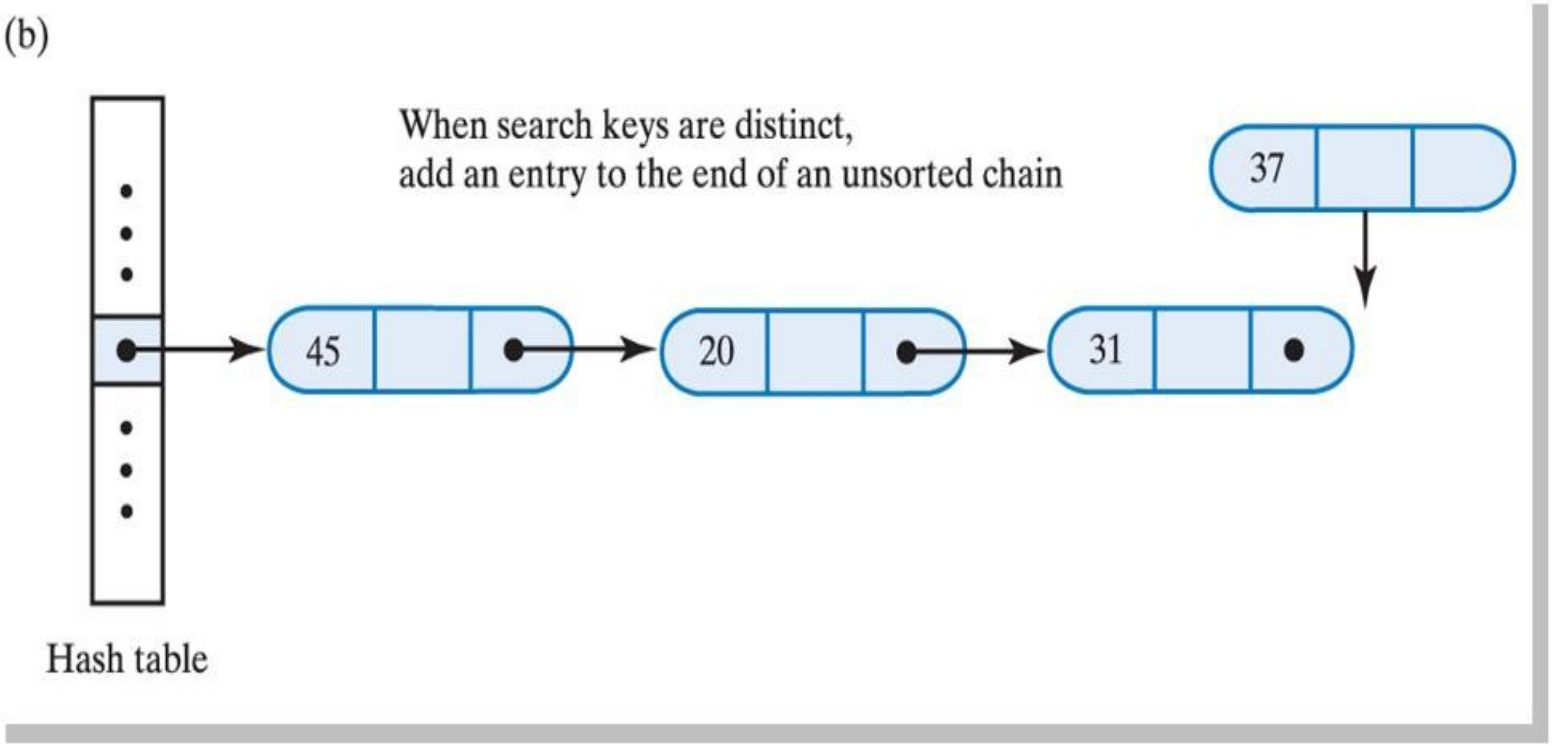



Fig. 18-10 Where new entry is inserted into linked bucket when integer search keys are (b) distinct and unsorted;

# Separate Chaining – Distinct Keys, Sorted Chain

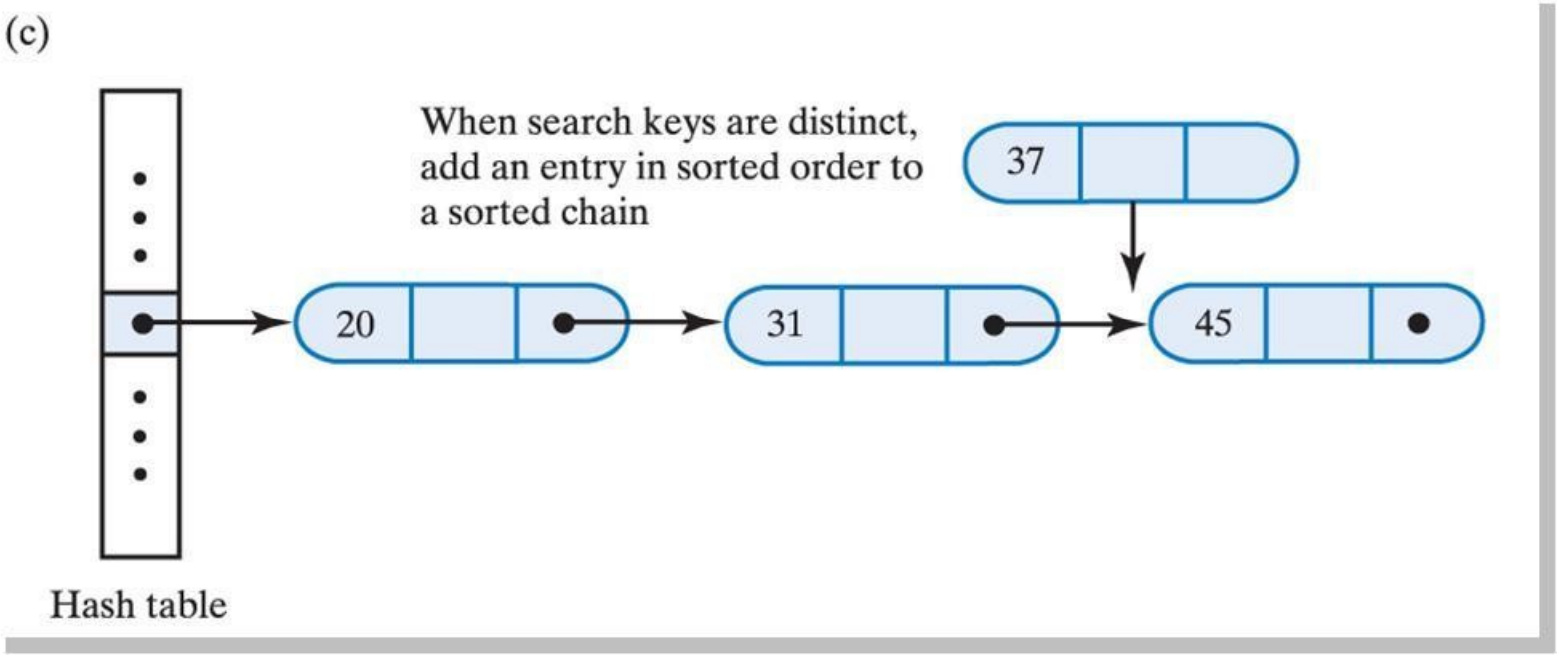
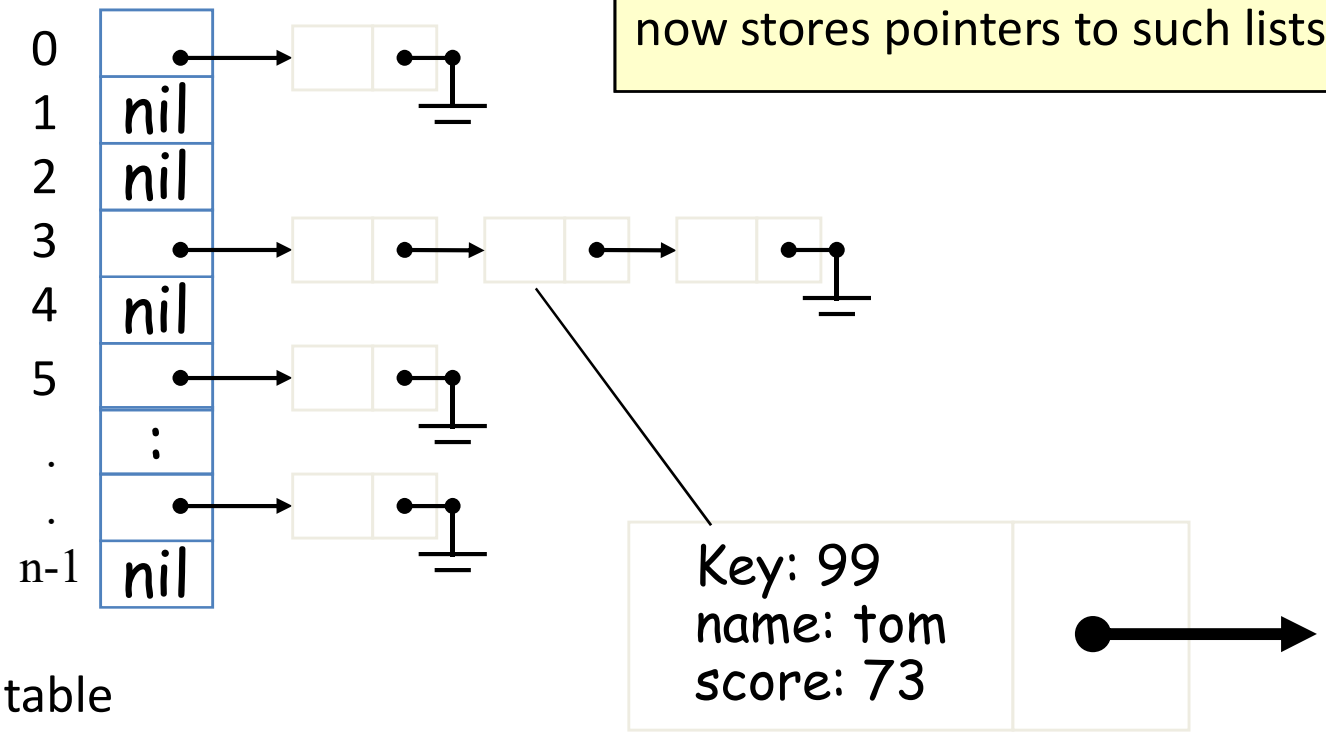



Fig. 18-10 Where new entry is inserted into linked bucket when integer search keys are (c) distinct and sorted

# Separate Chaining with Linked List

One way to handle collision is to store the collided records in a linked list. The array now stores pointers to such lists.




Hash table

# Implementing Separate Chaining

Sample code in **`Chapter10\version3`** folder:

- **`HashedDictionary.java`**

  Note

  – Inner class **`Node`** instead of **`TableEntry`**

  – Constructor:

  ```
  hashTable = new Node[primeSize];
  ```

  – Methods: **`add`** , **`remove`**, **`getValue`** , **`display`**

    - Coding for linked chain

# Efficiency of Hashing

- Resolving a collision takes time and thus causes the dictionary operations to be slower than an O(1) operation.
- As a hash table fills, collisions occur more often, decreasing performance even further .
- Since collision resolution takes considerably more time than evaluating the hash function, it is the prime contributor to the cost of hashing .

# The Load Factor

- To express the cost of hashing, we define a measure of how full a hash table is, i.e. the load factor $\lambda$.
- The load factor $\lambda$ is the *ratio of the size of the dictionary to the size of the hash table:*

$$\lambda = \frac{\text{No. of entries in the dictionary}}{\text{No. of locations in the hash table}}$$

Q: What is the load factor if the hash table is empty?

# Maximum Value of λ

- The maximum value of **λ** depends on the type of collision resolution used.
  - For open addressing schemes, λ's maximum value is 1 when the hash table is full.
  - For separate chaining, the number of entries can exceed the size of the hash table, so λ has no maximum value.
- Restricting the size of λ improves the performance of hashing.

# The Cost of Open Addressing

- All open addressing schemes use one location in the hash table per entry in the dictionary.
- The dictionary operations `getValue`, `remove`, and `add` each require a search of the probe sequence indicated by both the search key and the collision resolution scheme in effect.
- The performance of hashing degrades significantly as the load factor $\lambda$ increases.
- To maintain reasonable efficiency, the hash table should be less than half full, i.e. keep $\lambda < 0.5$.

# The Cost of Separate Chaining

- With separate chaining as the collision resolution strategy, each entry in the hash table can reference a chain of linked nodes.
- The number of such chains (including empty ones) is then the size of the hash table.
  - The load factor $\lambda$ is the number of dictionary entries divided by the number of chains, i.e. the average number of dictionary entries per chain.
- The average performance does not degrade significantly as the load factor $\lambda$ increases.
- To maintain reasonable efficiency, keep $\lambda < 1$ .

# Load Factors and Rehashing

- **λ** should be < 1 for separate chaining and < 0.5 for open addressing
- If the load factor exceeds the bounds, must rehash the table
- See method **`isHashTableTooFull`** in class **`HashedDictionary`**

# Separate Chaining vs. Open Addressing

- Advantages of separate chaining compared to open addressing:
  - Provides faster dictionary operations on average
  - Can use smaller hash table
  - Needs less frequent rehashing
- If both approaches have the same size array for a hash table, separate chaining uses more memory due to its linked chains
- Among open addressing schemes, double hashing is a good choice. It uses fewer comparisons than linear probing. Additionally, its probe sequence can reach the entire table, whereas quadratic probing cannot.

# Exercise

The following keys are to be hashed into a hash table of size 11:

11, 52, 32, 28, 85, 90, 61

Given the primary hash function : **$h_1$(key) = key modulo 11**

and that collisions are resolved by **double hashing** with the secondary hash function **$h_2$(key) = 5 - key modulo 5**

Table size of hash table is 11

# Exercise

Draw the resulting hash table when entries with the following sequence of keys are inserted to the hash table:

12, 44, 23, 94, 11, 39, 20, 3

Use the hash function: **h(i) = (2i + 5) % 11**

and **separate chaining** as the collision-resolution scheme.

# Advantages & disadvantages

| | Linear Probing | Quadratic probing | Double hashing |
|---|---|---|---|
| Advantage | Easy to implement | reduce the clustering problem | eliminates secondary clustering in most cases |
| Disadvantage | creates clusters in the table | requires more effort to compute the indices for the probe sequence than linear probing. | can be time-consuming to compute the 2 hash functions |

# Learning Outcomes

You should now be able to

- Write application programs that uses the ADT Dictionary.
- Override the method **`hashCode`** for a class whose instances are to be used as search keys.
- Implement hash functions and collision-resolution schemes.
- Discuss the advantages and disadvantages of various hashing and collision resolution schemes.
- Describe clustering and the problem it causes.

# References


- Carrano, F. M., 2019, Data Structures and Abstractions with Java, 5th edn, Pearson
- Liang, Y.D., 2018. Introduction to Java Programming and Data Structures.11th ed.United Kingdom:Pearson