CS2400 - Data Structures and Advanced Programming
Module 14: Hashing

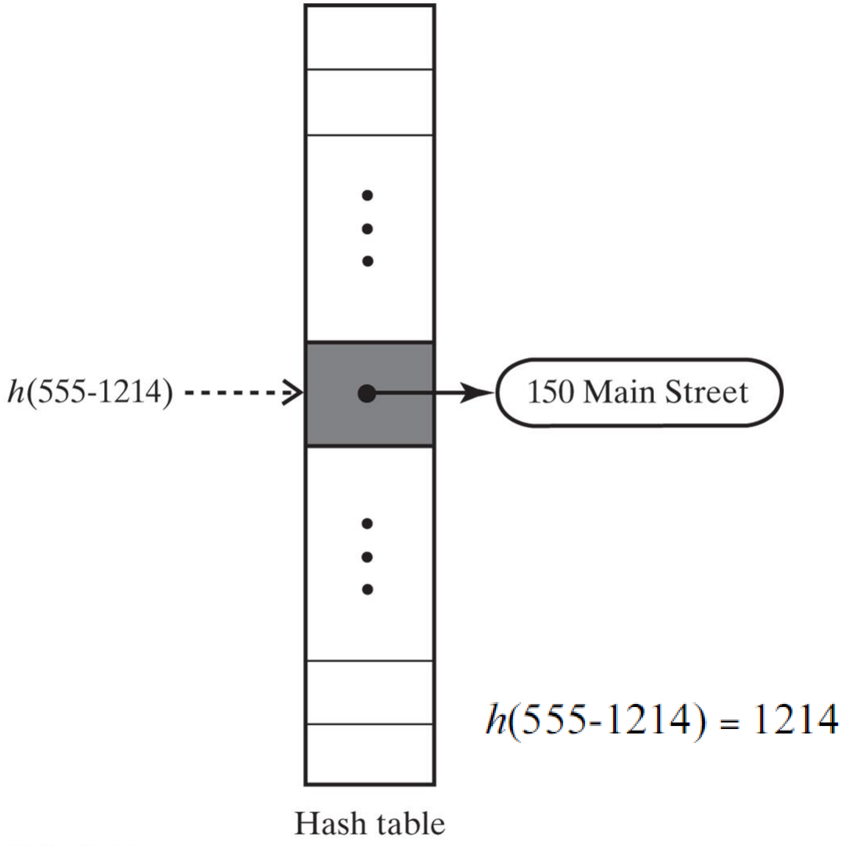
Hao Ji

Computer Science Department

Cal Poly Pomona

Hashing

- **Hashing** is a technique that ideally can result in O(1) search times.

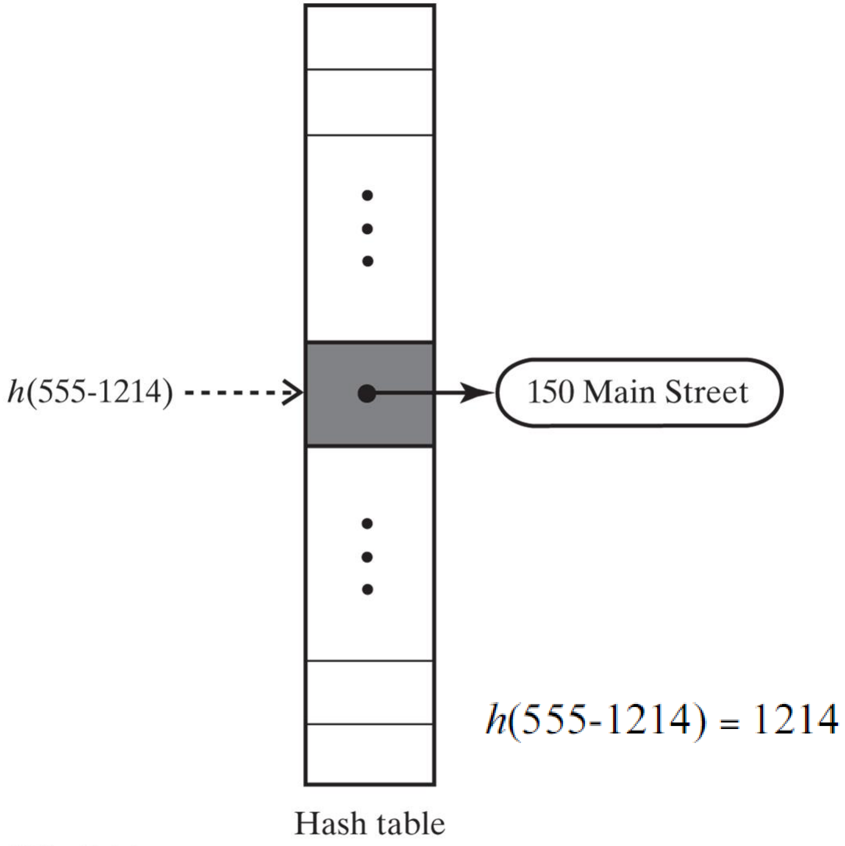


Hashing

- **Hashing** is a technique that ideally can result in O(1) search times.

- It uses **a hash function** to determine an index of an entry (in an array) using only the entry's search key, without searching.

```
index = getHashIndex(key)
```



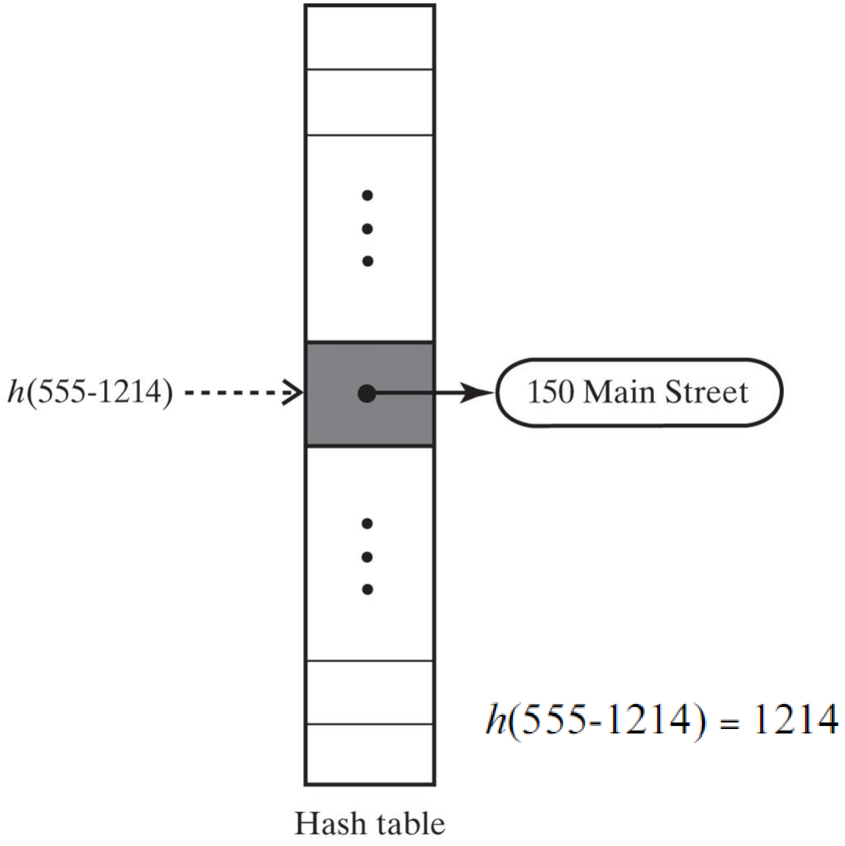
Hashing

- **Hashing** is a technique that ideally can result in O(1) search times.

- It uses **a hash function** to determine an index of an entry (in an array) using only the entry's search key, without searching.

```
index = getHashIndex(key)
```

- The array itself is called a **hash table**.



Hash Functions

- A perfect hash function maps each search key into a different integer that is suitable as an index to the hash table.

Algorithm **add(key, value)**

index = getHashIndex(key)

hashTable[index] = value

Algorithm **getValue(key)**

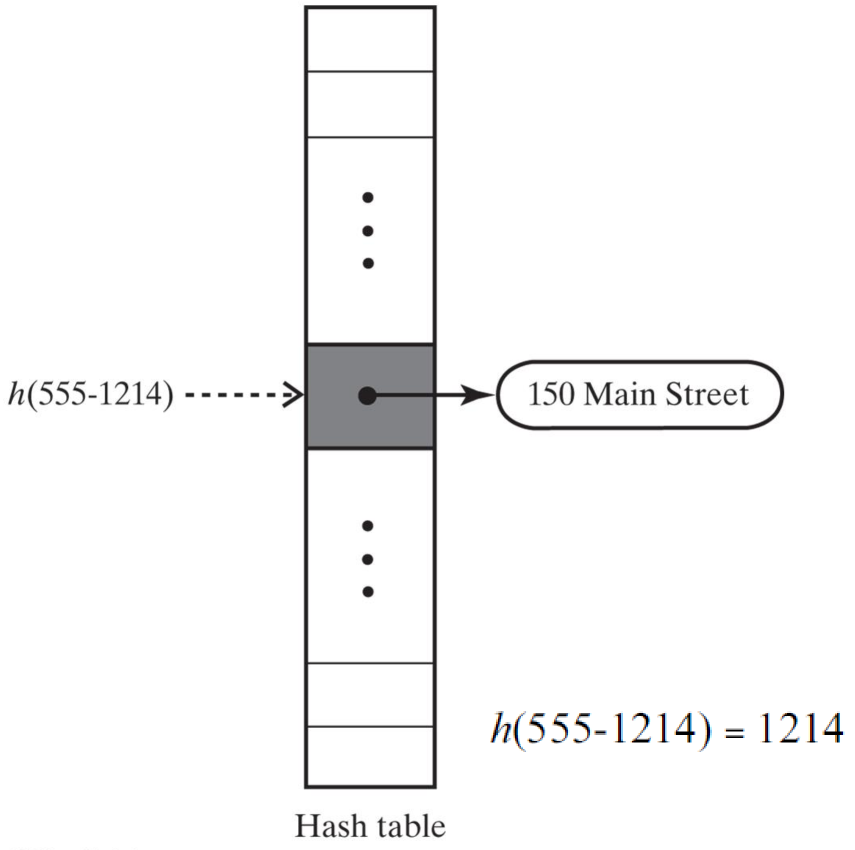
index = getHashIndex(key)

return hashTable[index]

Hash Functions

- However, a perfect hash function usually results in a full hash table, where

the table size = the number of data items.

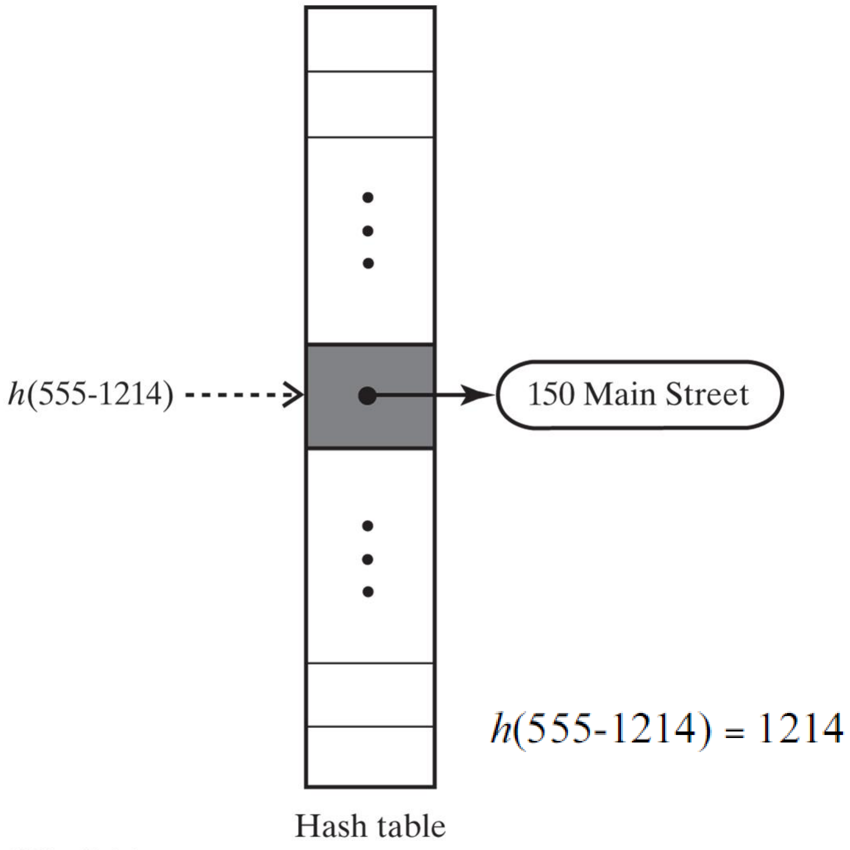


Hash Functions

- However, a perfect hash function usually results in a full hash table, where

  the table size = the number of data items.

- Issues
  - A full hash table tends to be expensive in terms of memory cost.
  - In practice, most hash tables are not full, with only a few of elements dynamically are in use.



Hash Functions

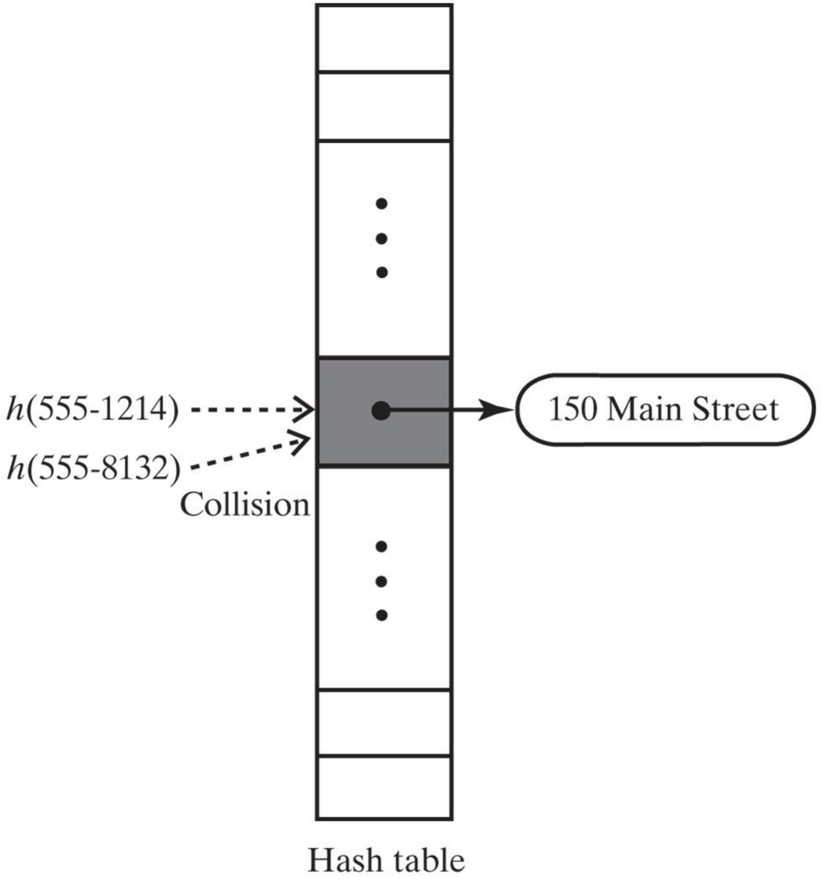
- With a smaller hash table, hash functions are not perfect,
  - allow more than one search key to map into a single index

Algorithm **getHashIndex(phoneNumber)**

// Returns an index to an array of tableSize *elements.*

i = last four digits of phoneNumber

return i % tableSize

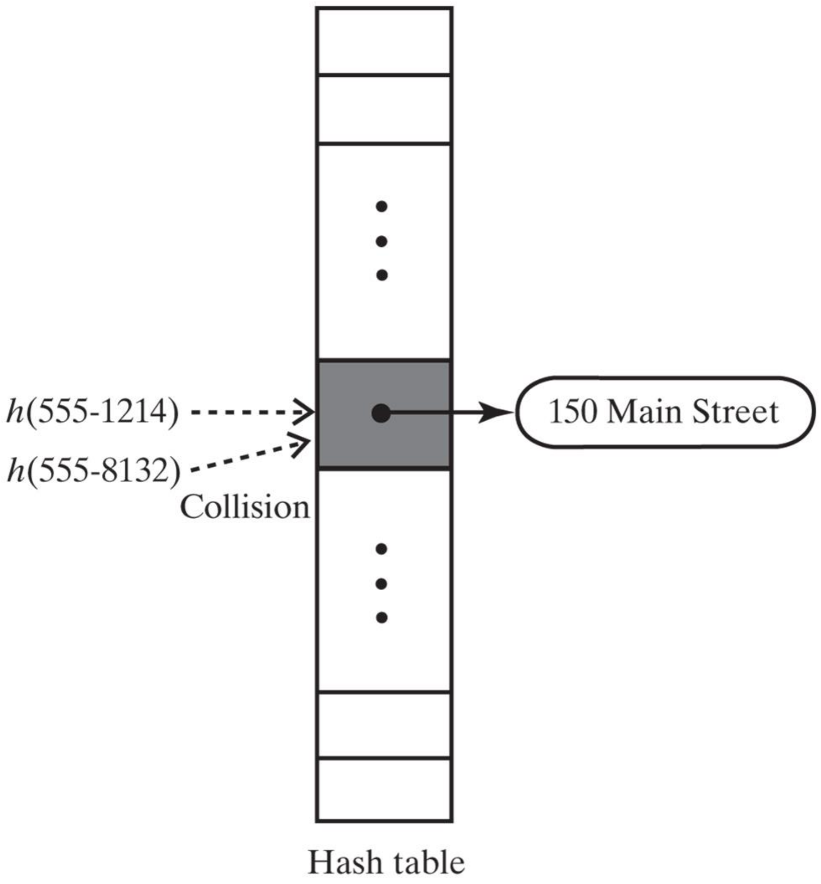


Hash Functions

- With a smaller hash table, hash functions are not perfect,
  - allow more than one search key to map into a single index

> ***Algorithm*** **getHashIndex(phoneNumber)**
>
> *// Returns an index to an array of* tableSize *elements.*
>
> *i = last four digits of* phoneNumber
>
> **return** i % tableSize

- For example, consider **`tableSize = 101`**
  - **`getHashIndex(555-1214) = 52`**
  - **`getHashIndex(555-8132) = 52`** ***also!!!***



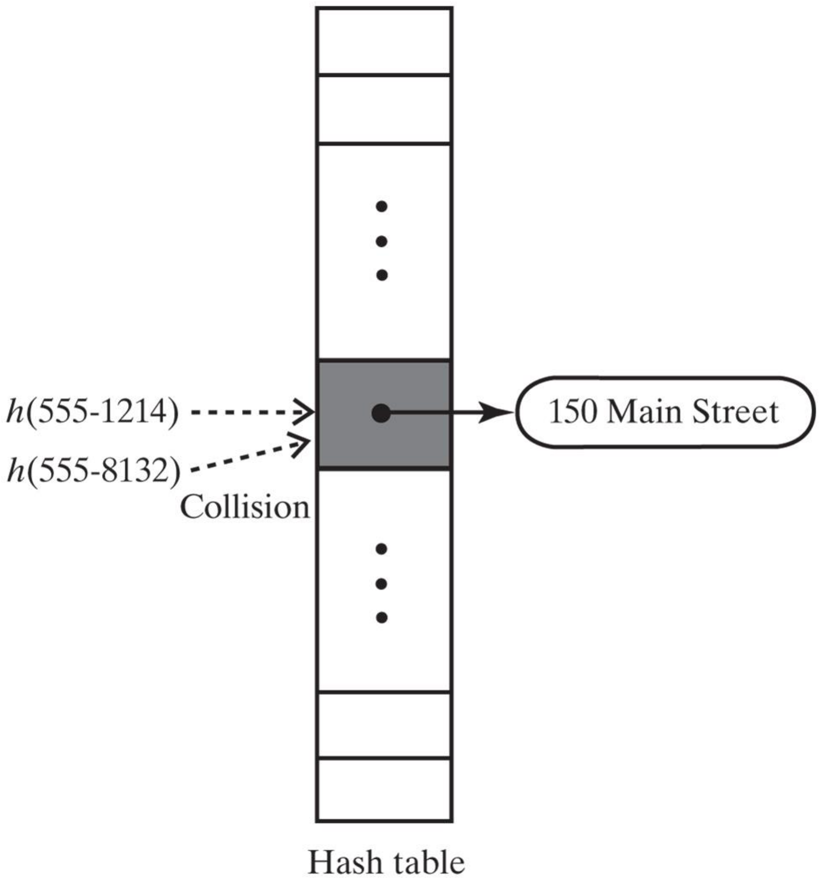
Hash Functions

- With a smaller hash table, hash functions are not perfect,
  - allow more than one search key to map into a single index

> ***Algorithm*** **getHashIndex(phoneNumber)**
>
> *// Returns an index to an array of* tableSize *elements.*
>
> *i = last four digits of* phoneNumber
>
> **return** i % tableSize

- For example, consider **`tableSize = 101`**
  - **`getHashIndex(555-1214) = 52`**
  - **`getHashIndex(555-8132) = 52`** ***also!!!***

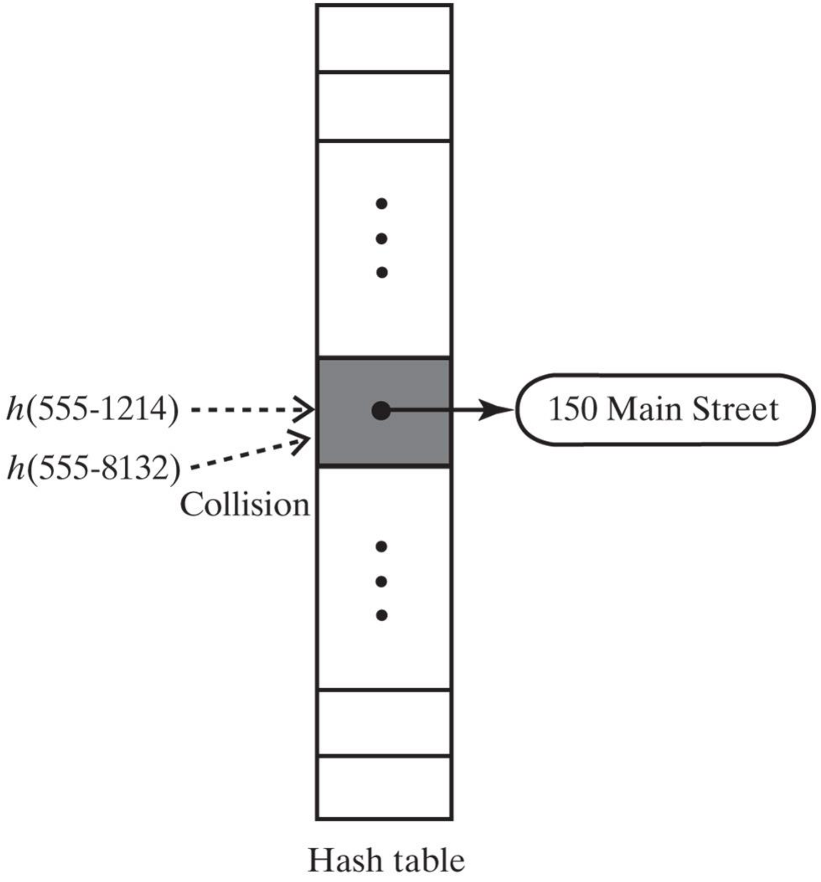
A collision caused by the hash function *h*



Hash Functions

- A **good** hash function should
  - **Be fast to compute**
  - **Minimize collisions**

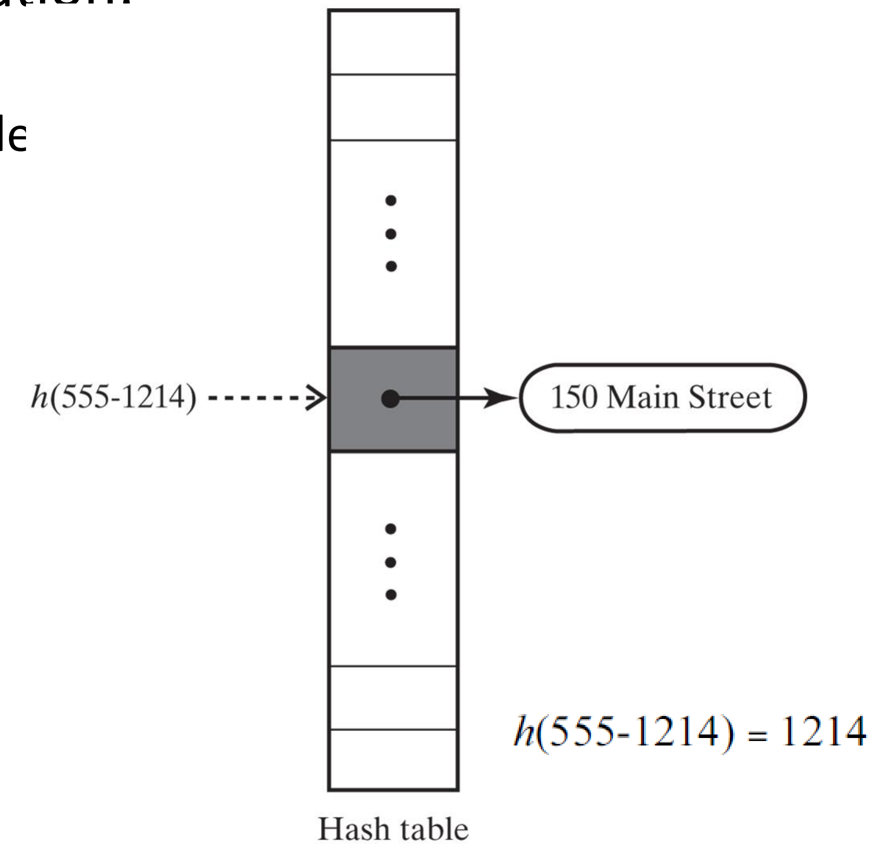
```
private int getHashIndex(K key)
{
    int hashIndex = key.hashCode() % hashTable.length;

    if (hashIndex < 0)
        hashIndex = hashIndex + hashTable.length;

    hashIndex = Probing(hashIndex, key);

    return hashIndex;
} // end getHashIndex
```



Hash Functions

- Typical hash functions perform two steps in computation:
  - **Convert** search key to an integer called the hash code.
  - **Compress** hash code into the range of indices for hash table

```
private int getHashIndex(K key)
{
    int hashIndex = key.hashCode() % hashTable.length;

    if (hashIndex < 0)
        hashIndex = hashIndex + hashTable.length;

    hashIndex = Probing(hashIndex, key);

    return hashIndex;
} // end getHashIndex
```



Computing Hash Codes

- Java's base class **`Object`** has a method **`hashCode`** that returns an integer hash code.

- However, a class should define its own version of **`hashCode`**. (This is because the method will return an *int* value based on the memory address of the object used to invoke it.)

Computing Hash Codes

- Java's base class **`Object`** has a method **`hashCode`** that returns an integer hash code.
- A class should define its own version of **`hashCode`**, as the method will return an *int* value based on the memory address of the object used to invoke it.

Note: Guidelines for the method `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 you call an object's `hashCode` more than once during the execution of a program, and if the object's data remains the same during this time, `hashCode` 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 Code for a String

- A hash code for a string
  - Using a character’s Unicode integer is common

Unicode Character Codes

The printable characters shown are a subset of the Unicode character set known as the ASCII character set. The numbering is the same whether the characters are considered to be members of the Unicode character set or members of the ASCII character set. (Character number 32 is the blank.)

32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g		

Hash Code for a String

- A hash code for a string
  - Using a character's Unicode integer is common
  - More robust approach:
    - Multiply Unicode value of each character by significance based on character's position,
    - Then sum values

$$u_0 g^{n-1} + u_1 g^{n-2} + \dots + u_{n-2}\, g + u_{n-1}$$

$$(\dots((u_0\, g + u_1)\, g + u_2)\, g + \dots + u_{n-2})\, g + u_{n-1}$$

```java
int hash = 0;
int n = s.length();
for (int i = 0; i < n; i++)
   hash = g * hash + s.charAt(i);
```

Unicode Character Codes

The printable characters shown are a subset of the Unicode character set known as the ASCII character set. The numbering is the same whether the characters are considered to be members of the Unicode character set or members of the ASCII character set. (Character number 32 is the blank.)

32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g		

Hash Code for a Primitive Type

- If data type is **`int`**,
  - Use the key itself

- For **`byte, short, char`**:
  - Cast as **`int`**

- Other primitive types
  - Work with their internal binary representations

Compressing a Hash Code into an Index for the Hash Table

- Common way to scale an integer
  - Use Java mod operator **%**: **`code % n`**

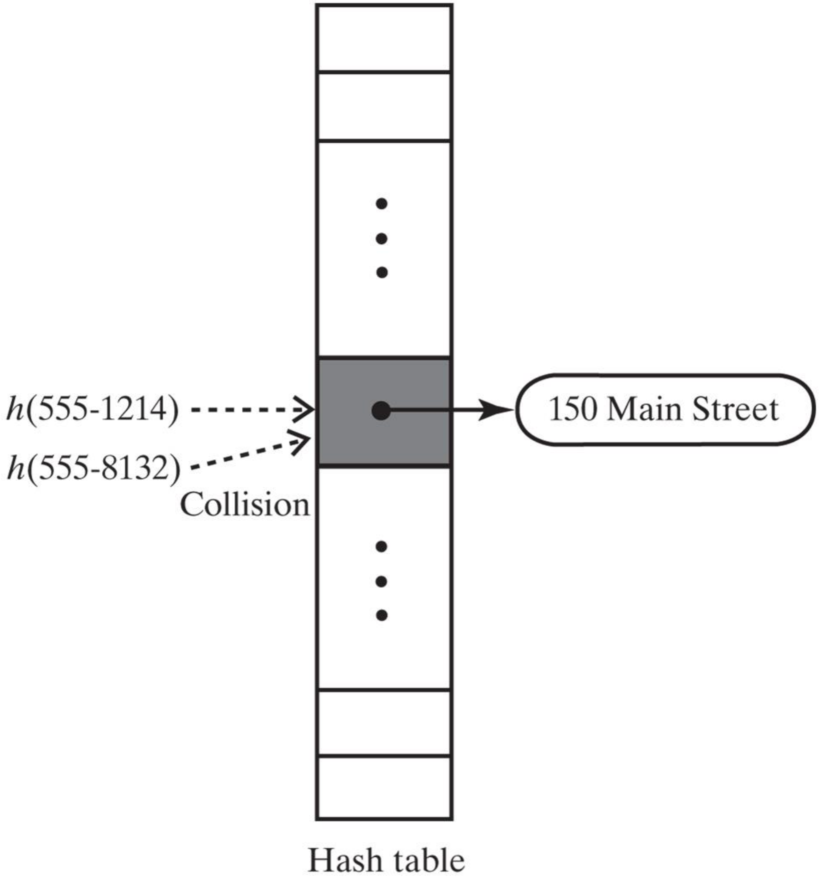
- Best to use an odd number for n; Prime numbers often give good distribution of hash values

```
private int getHashIndex(K key)
{
    int hashIndex = key.hashCode() % hashTable.length;

    if (hashIndex < 0)
        hashIndex = hashIndex + hashTable.length;

    hashIndex = Probing(hashIndex, key);

    return hashIndex;
} // end getHashIndex
```

Hash Functions

- A **good** hash function should
  - Be fast to compute
  - **Minimize collisions**

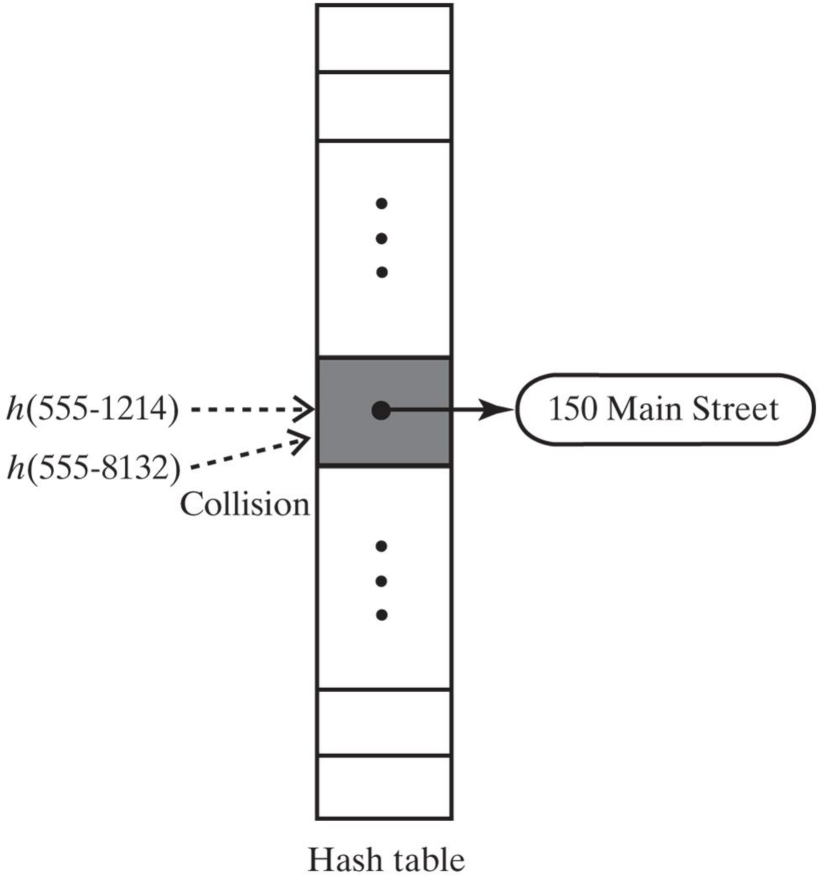
```
private int getHashIndex(K key)
{
     int hashIndex = key.hashCode() % hashTable.length;

     if (hashIndex < 0)
          hashIndex = hashIndex + hashTable.length;

     hashIndex = Probing(hashIndex, key);

     return hashIndex;
} // end getHashIndex
```



Resolving Collisions

- **Collision**:
  - Hash function maps search key into a location in hash table already in use

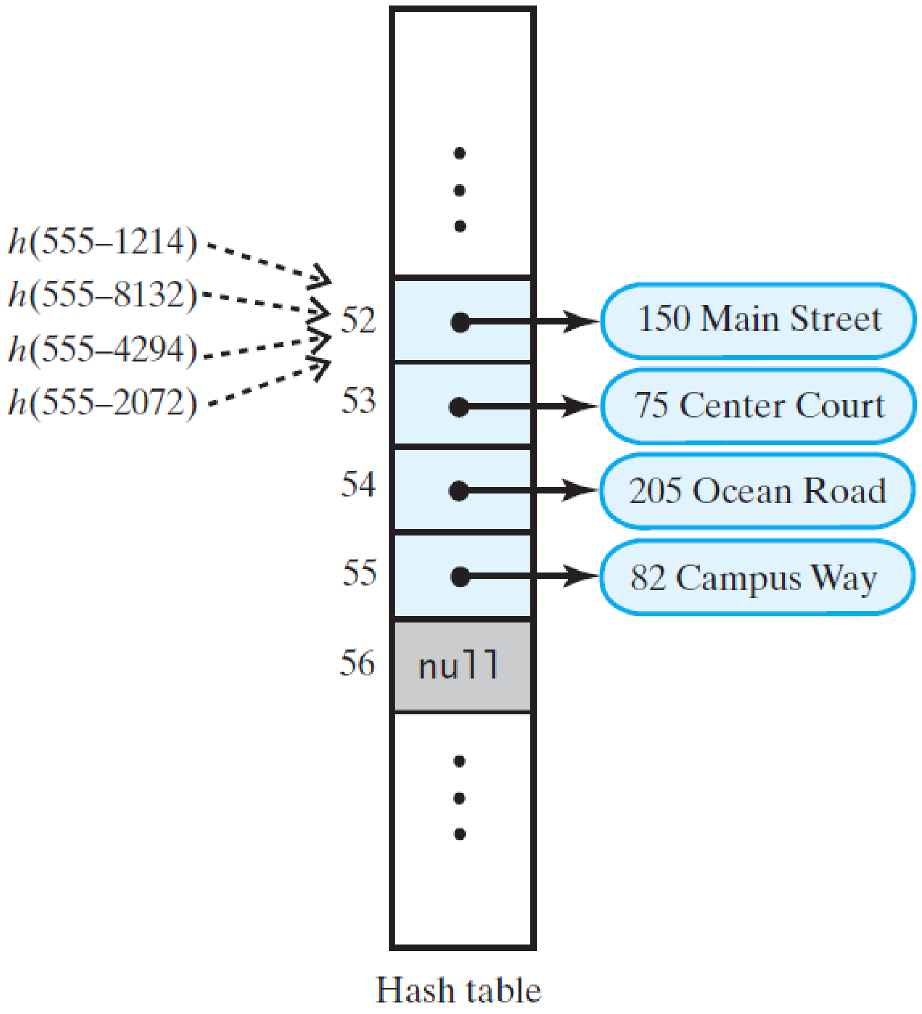
- Two choices:
  - (**Open Addressing**) Use another location in the hash table
  - (**Separate Chaining**) Change the structure of the hash table so that each array location can represent more than one value



Resolving Collisions

- **Linear probing**
  - Resolves a collision during hashing by examining consecutive locations in hash table
  - Beginning at original hash index
  - Find the next available one

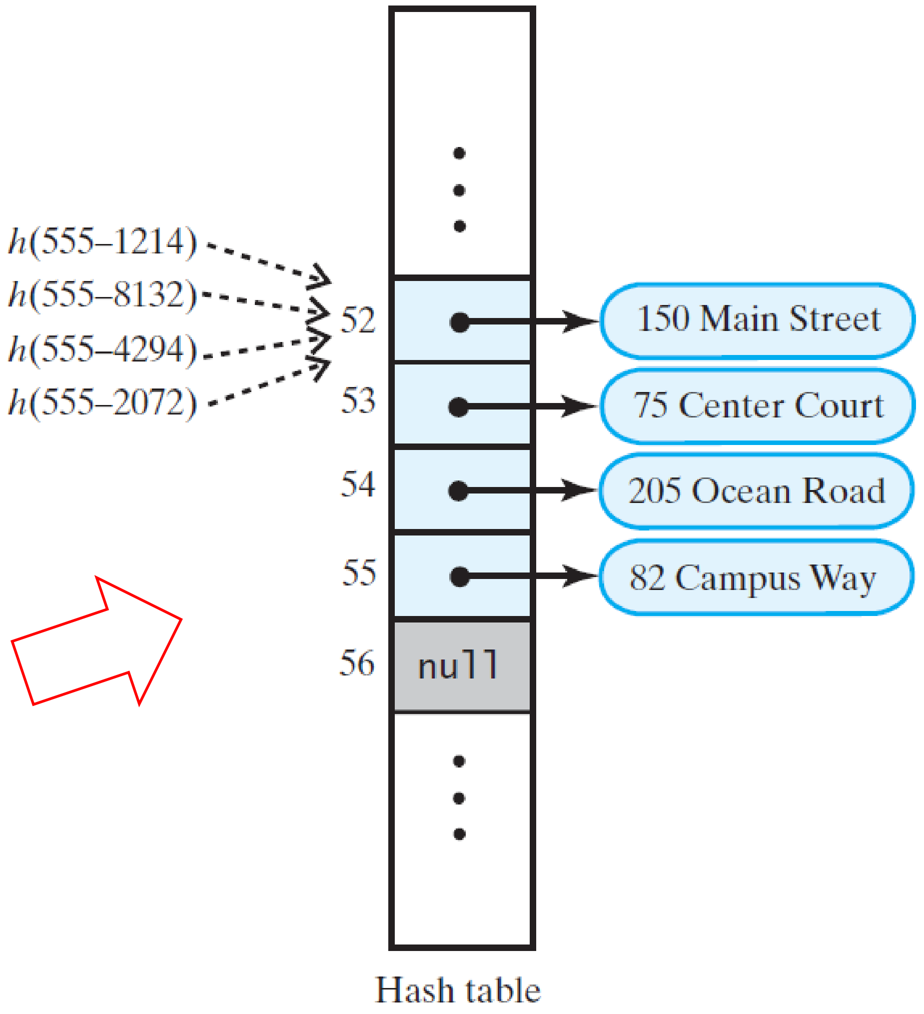
- If probe sequence reaches end of table, go to beginning of table (circular hash table)



Resolving Collisions

- **Add** **operation** in Linear probing

```
addressBook.add("555-1214", "150 Main Street");
addressBook.add("555-8132", "75 Center Court");
addressBook.add("555-4294", "205 Ocean Road");
addressBook.add("555-2072", "82 Campus Way");
```



In-Class Exercise

- Given a table size of 13 with the hash function

$$h(k) = k \; \% \; table\ size$$

for a sequence of the entries 14, 2, 15, 26 and 29, show the hash table after the five entries are inserted into the table using open addressing with linear probing

In-Class Exercise

- Given a table size of 13 with the hash function

$$h(k) = k \% \text{ table size}$$

for a sequence of the entries 14, 2, 15, 26 and 29, show the hash table after the five entries are inserted into the table using open addressing with linear probing

14 % 13 = 1

2 % 13 = 2

15 % 13 = 2 << collision

26 % 13 = 0

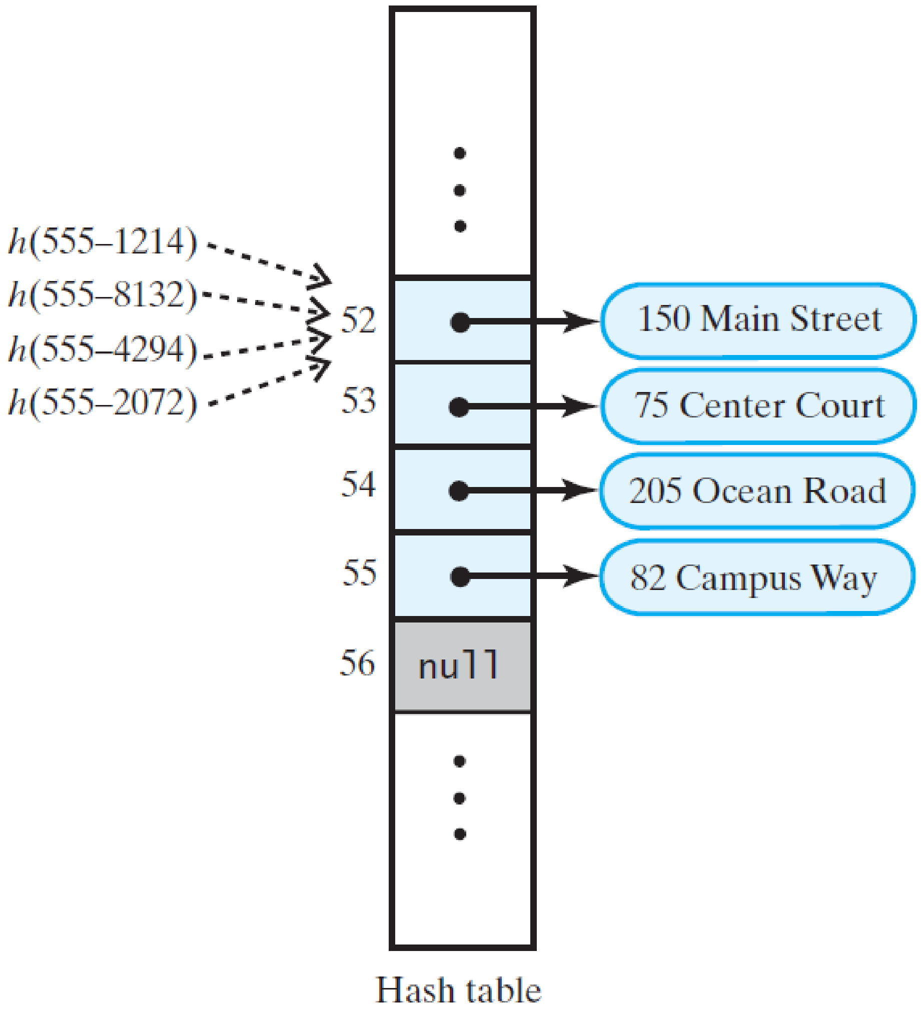
29 % 13 = 3 << collision

index	0	1	2	3	4	5	6	7	8	9	10	11	12
entry	26	14	2	15	29								

Resolving Collisions

- **(Issue)** in implementing getVaule() operation
  - Given a key, we can not tell which entry is the right one, as some entries whose search keys hash to the same index

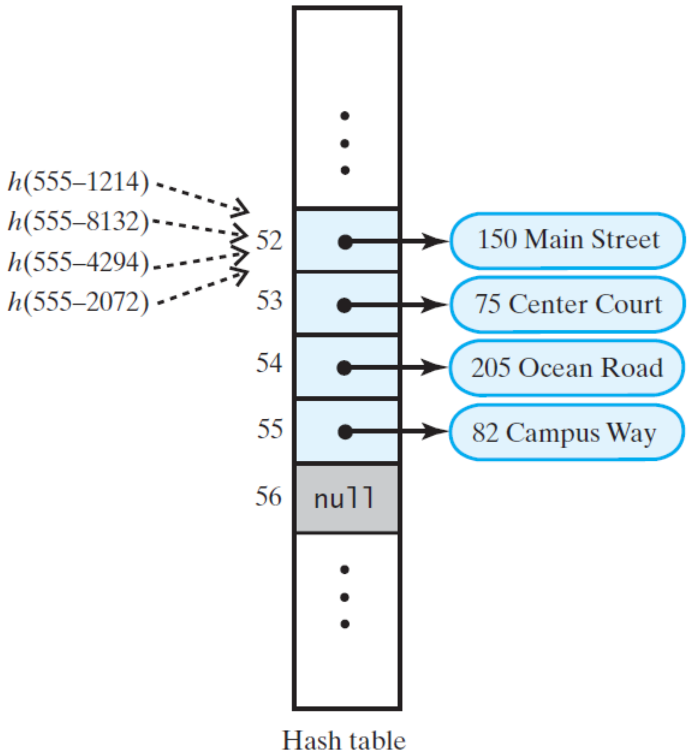
```
String streetAddress = addressBook.getValue("555-2072");
```

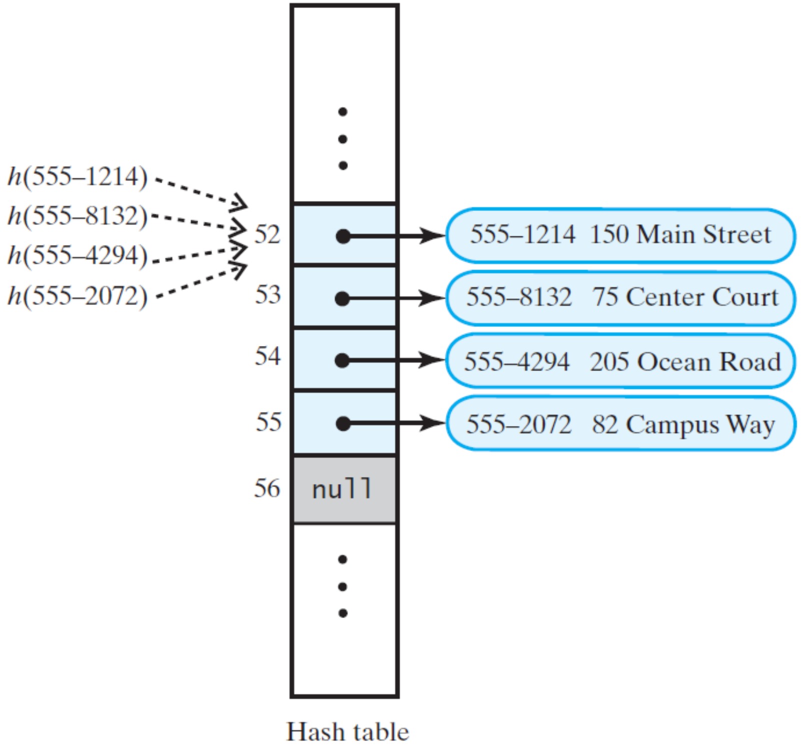


Resolving Collisions

- Revised hash table for linear probing
  - **Each entry contains a search key and its associated value**

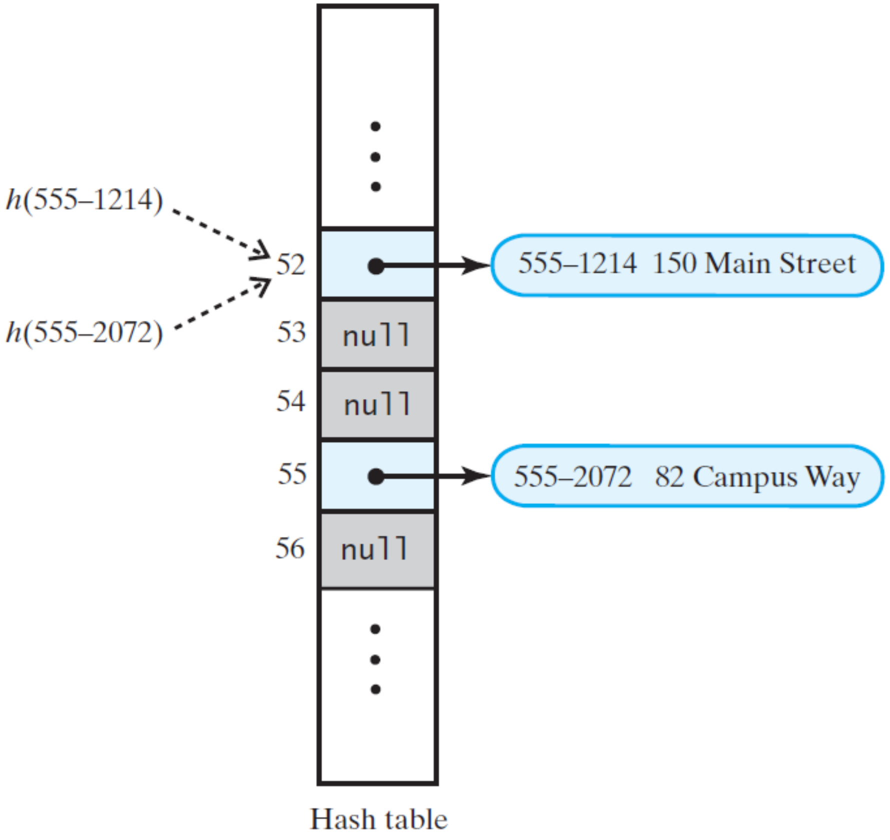
```
String streetAddress = addressBook.getValue("555-2072");
```





Resolving Collisions

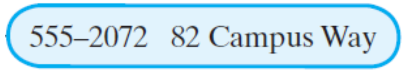
- Removal **operation** in Linear probing
  - What if we simply remove an entry by placing **null** in that location?

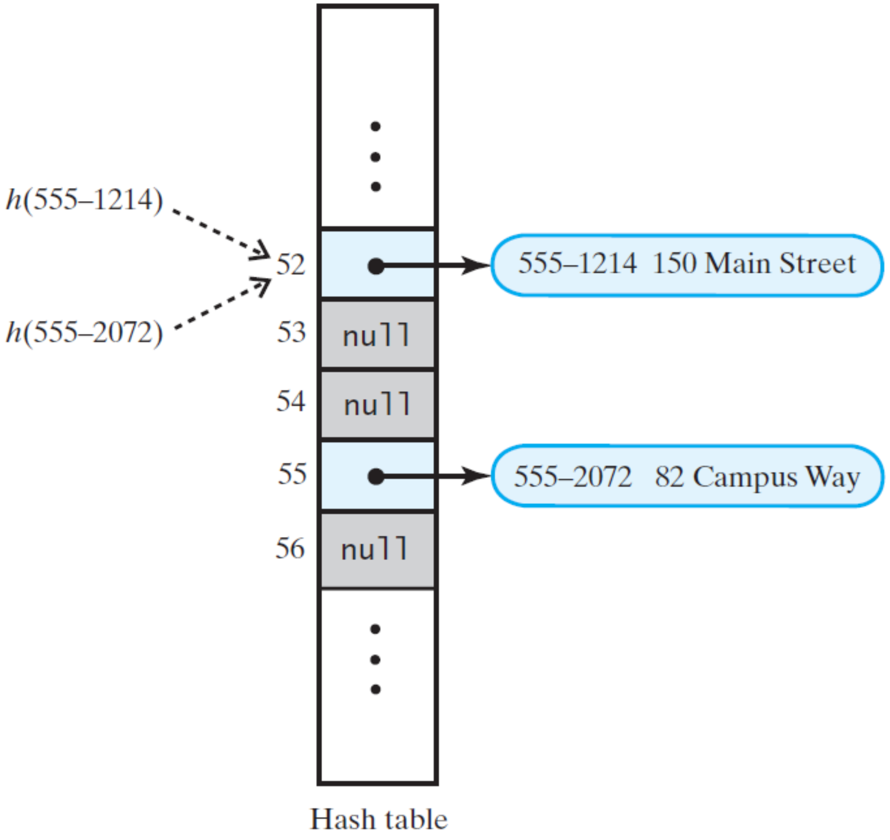


Resolving Collisions

- **Removal operation** in Linear probing
  - What if we simply remove an entry by placing **null** in that location?

  - **(Issue)** we then have difficulty in finding the following item in linear probing

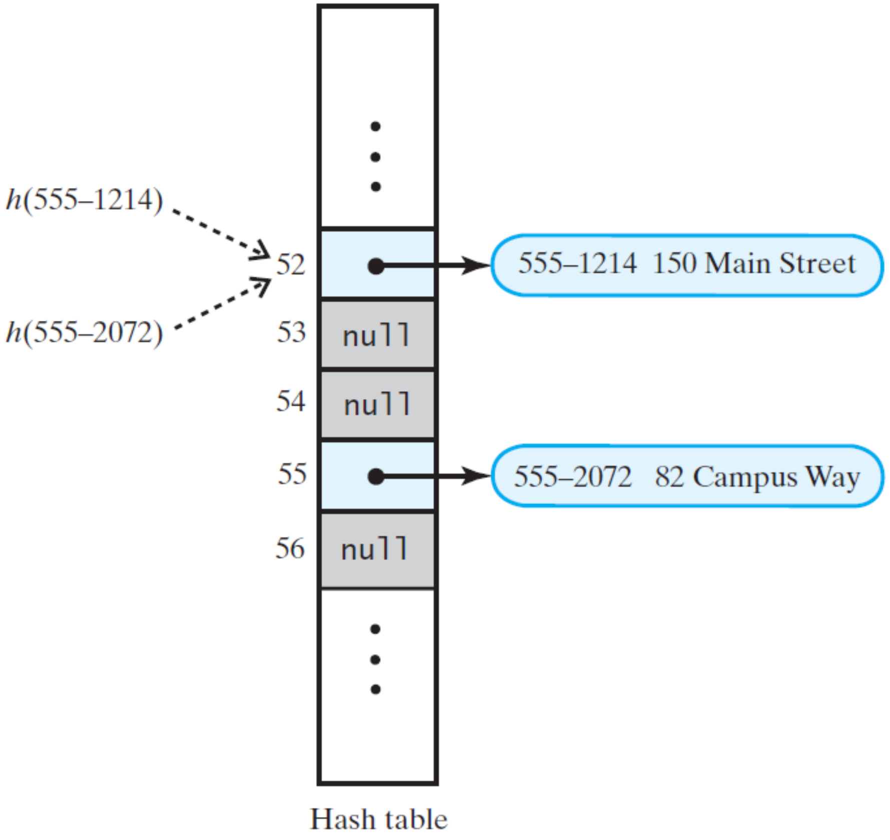




Resolving Collisions

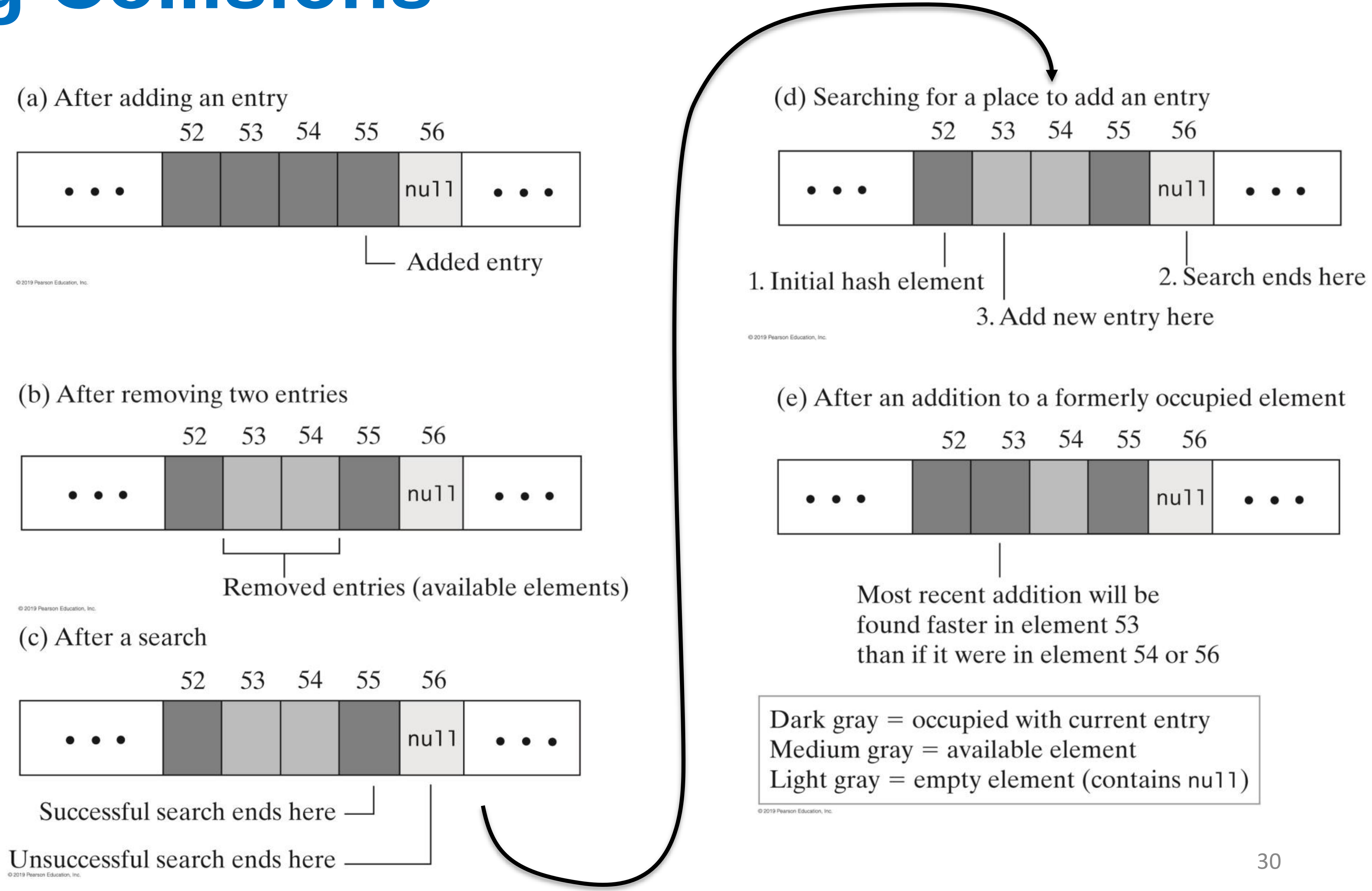
- **Removal operation** in Linear probing

- **Solution**
  - Need to distinguish among three kinds of locations in the hash table
    - `Occupied`
      - location references an entry in the dictionary
    - `Empty`
      - location contains null and always has
    - `Available`
      - location's entry was removed from the dictionary



Resolving Collisions

- **Linear probing**



HashedDictionary

HashedDictionary
-hashTable: Entry<K, V>[] -numberOfEntries: int
+add(key : K, value : V) : void +remove(key : K) : V +getValue(key : K) : V +contains(key : K) : Boolean +isEmpty() : Boolean +getSize() : integer +clear() : void

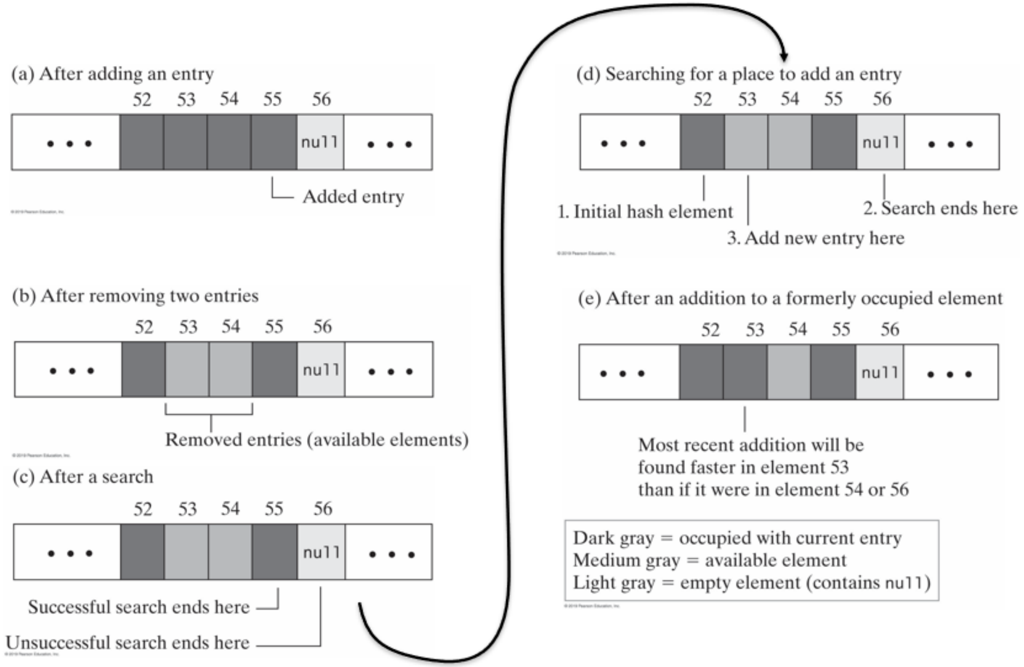
```
private class Entry<S, T>
{
   private S key;
   private T value;
   private boolean inTable; // true if entry is in hash table

   private      Entry(S searchKey, T dataValue)
   {
      key = searchKey;
      value = dataValue;
      inTable = true;
   } // end constructor
   . . .
```

LISTING 22-1 An outline of the class `HashedDictionary`

```
import java.util.Iterator;
import java.util.NoSuchElementException;

/**
   A class that implements a dictionary by using hashing.
   @author Frank M. Carrano
*/
public class HashedDictionary<K, V>
             implements DictionaryInterface<K, V>
{
   private      Entry<K, V>[] hashTable; // dictionary entries
   private int numberOfEntries;
   private int locationsUsed; // number of table locations not null
   private static final int DEFAULT_SIZE = 101;       // must be prime
   private static final double MAX_LOAD_FACTOR = 0.5; // fraction of
                                                   // hash table that can be filled

   public HashedDictionary()
   {
      this(DEFAULT_SIZE); // call next constructor
   } // end default constructor

   public HashedDictionary(int tableSize)
   {
      int primeSize = getNextPrime(tableSize);

      hashTable = new TableEntry[primeSize];
      numberOfEntries = 0;
      locationsUsed = 0;
   } // end constructor

   < Implementations of methods in DictionaryInterface >
   . . .

   < Implementations of private methods >
   . . .

   private class Entry<S, T>
   {
      < See Segment 22.9 >
   } // end TableEntry
} // end HashedDictionary
```

Linear Probing - Probe Algorithm

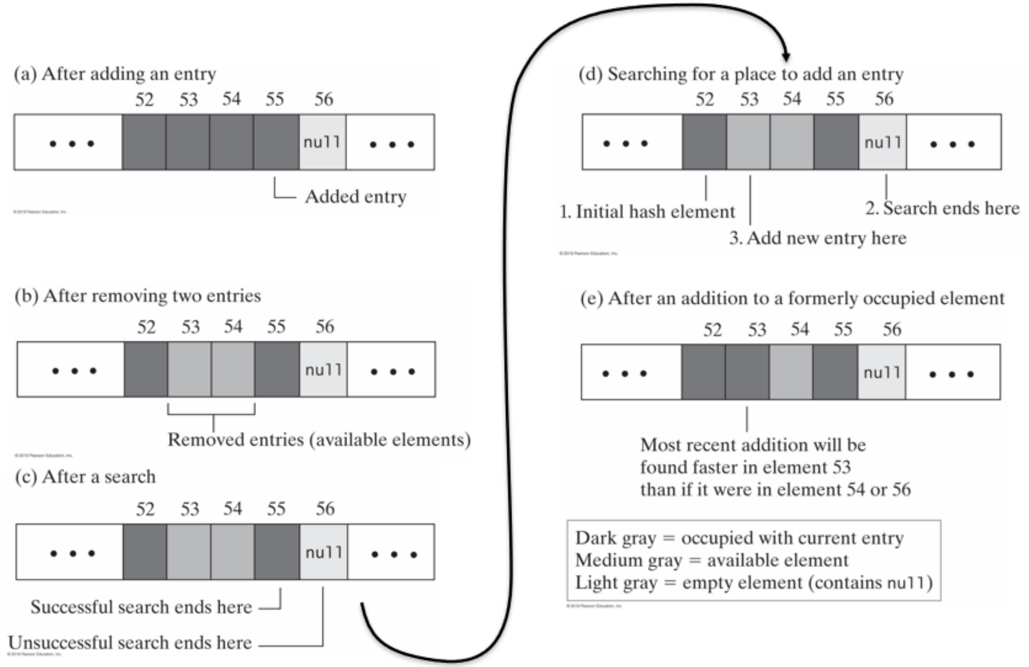
```
// Precondition: checkIntegrity has been called.
private int linearProbe(int index, K key) {
    boolean found = false;
    int availableIndex = -1; // Index of first available location (from which an entry was removed)

    while (!found && (hashTable[index] != null)) {
        if (hashTable[index] != AVAILABLE) {
            if (key.equals(hashTable[index].getKey()))
                found = true; // Key found
            else // Follow probe sequence
                index = (index + 1) % hashTable.length; // Linear probing
        }
        else // Skip entries that were removed
        {
            // Save index of first location in removed state
            if (availableIndex == -1)
                availableIndex = index;

            index = (index + 1) % hashTable.length; // Linear probing
        } // end if
    } // end while
        // Assertion: Either key or null is found at hashTable[index]

    if (found || (availableIndex == -1))
        return index; // Index of either key or null
    else
        return availableIndex; // Index of an available location
} // end linearProbe
```




```
Entry<K, V> AVAILABLE = new Entry<>(null, null);
```

Add() Method

```java
public V add(K key, V value) {
    checkIntegrity();
    if ((key == null) || (value == null))
        throw new IllegalArgumentException("Cannot add null to a dictionary.");
    else {
        V oldValue; // Value to return

        int index = getHashIndex(key);

        // Assertion: index is within legal range for hashTable
        assert (index >= 0) && (index < hashTable.length);

        if ((hashTable[index] == null) || (hashTable[index] == AVAILABLE)) { // Key not found, so insert new entry
            hashTable[index] = new Entry<>(key, value);
            numberOfEntries++;
            oldValue = null;
        } else { // Key found; get old value for return and then replace it
            oldValue = hashTable[index].getValue();
            hashTable[index].setValue(value);
        } // end if

        // Ensure that hash table is large enough for another add
        if (isHashTableTooFull())
            enlargeHashTable();

        return oldValue;
    } // end if
} // end add
```

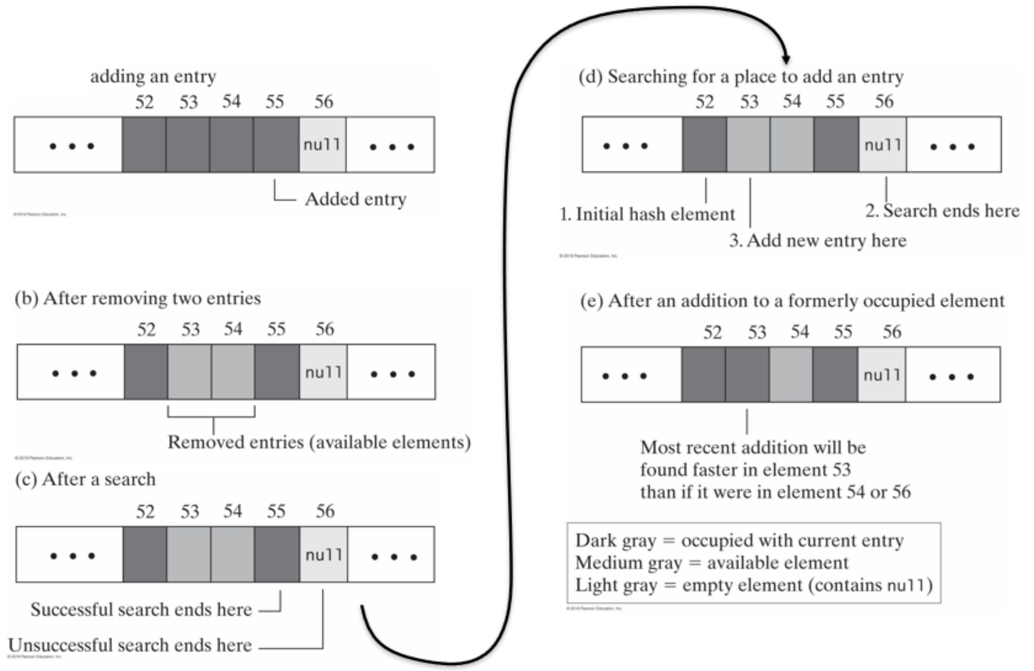


remove() Method

```java
public V remove(K key) {
    checkIntegrity();
    V removedValue = null;

    int index = getHashIndex(key);

    if ((hashTable[index] != null) && (hashTable[index] != AVAILABLE)) {
        // Key found; flag entry as removed and return its value
        removedValue = hashTable[index].getValue();
        hashTable[index] = AVAILABLE;
        numberOfEntries--;
    } // end if
        // Else not found; result is null

    return removedValue;
} // end remove
```

getValue() Method

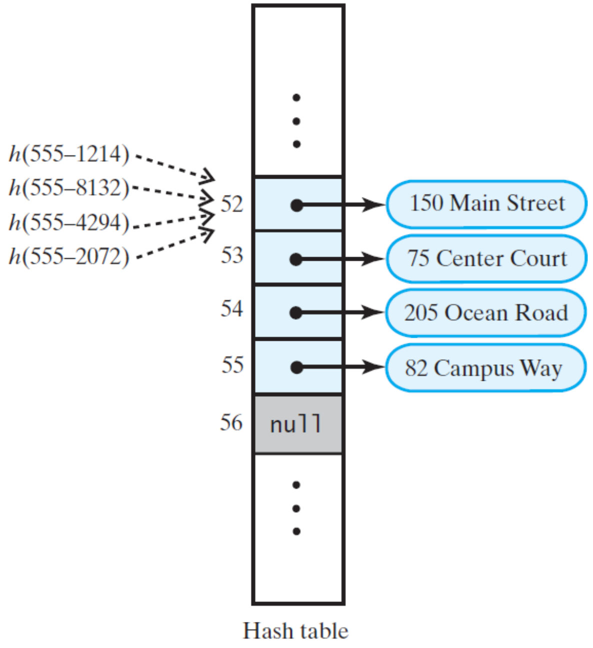
```java
public V getValue(K key) {
    checkIntegrity();
    V result = null;

    int index = getHashIndex(key);

    if ((hashTable[index] != null) && (hashTable[index] != AVAILABLE))
        result = hashTable[index].getValue(); // Key found; get value
    // Else not found; result is null

    return result;
} // end getValue
```



Issue of Linear Probing: Clustering

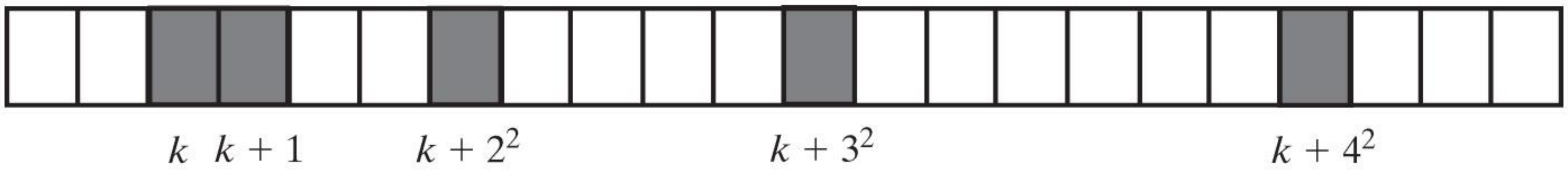
- Collisions resolved with linear probing cause groups of consecutive locations in hash table to be occupied
  - Each group is called a ***cluster***

- Bigger clusters mean longer search times following collision



Quadratic Probing

- Linear probing looks at consecutive locations beginning at index $\boldsymbol{k}$
- Quadratic probing:
  - Considers the locations at indices $\boldsymbol{k + j^2}$
  - Uses the indices `k, k + 1, k + 4, k + 9, …`

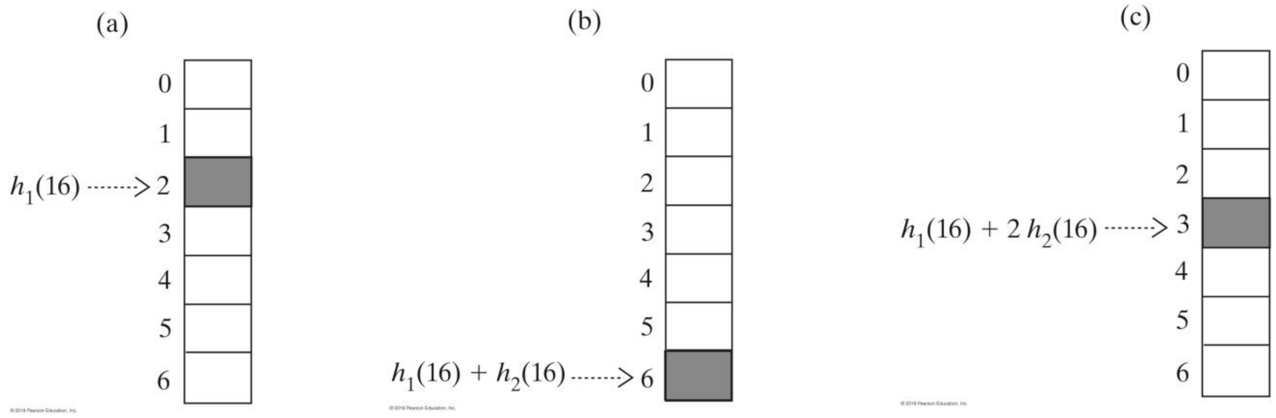


A probe sequence of length five using quadratic probing

Open Addressing with Double Hashing

- Linear probing looks at consecutive locations beginning at index $\boldsymbol{k}$
- Quadratic probing considers the locations at indices $\boldsymbol{k + j^2}$
- Double hashing uses a second hash function to compute these increments

$h_1(key) = key \text{ modulo } 7$
$h_2(key) = 5 - key \text{ modulo } 5$



The probe sequence has the following indices: 2, 6, 3, 0, 4, 1, 5, 2, ….

Potential Problem with Open Addressing

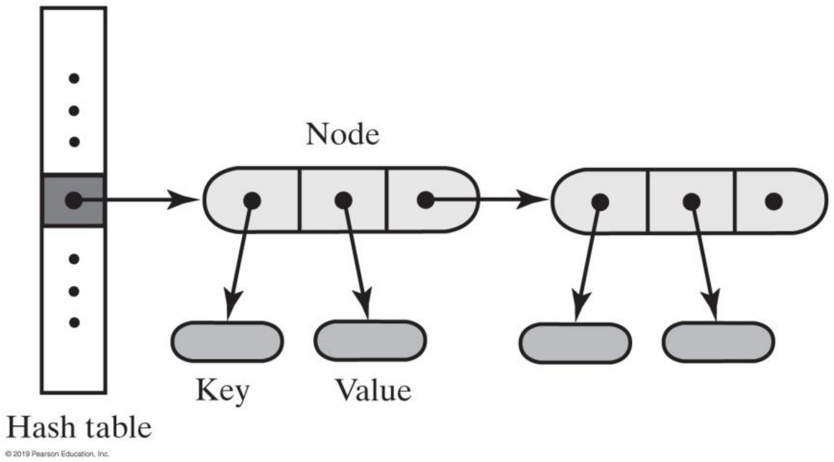
- Recall each location is either occupied, empty, or available
  - Frequent additions and removals can result in **no locations that are null**

- An alternative approach is to use separate chaining

Separate Chaining

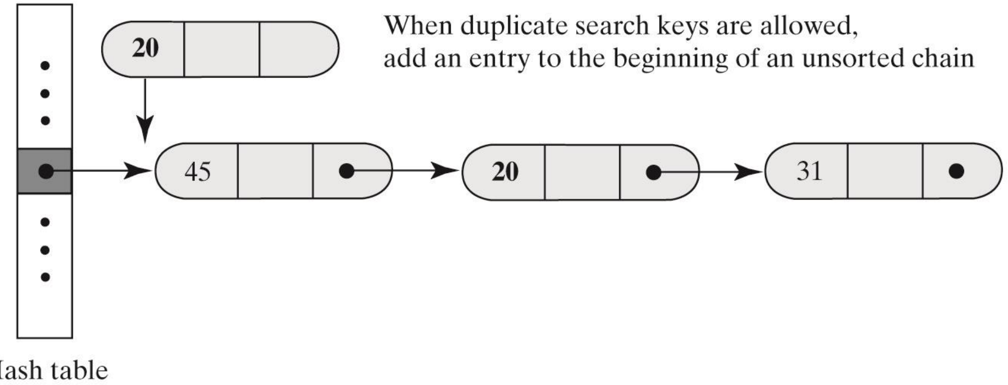
- Alter the structure of the hash table
  - Each location can represent more than one value.
  - Such a location is called a bucket

- Decide how to represent a bucket
  - `list, sorted list`
  - `array`
  - `linked nodes`
  - `vector`

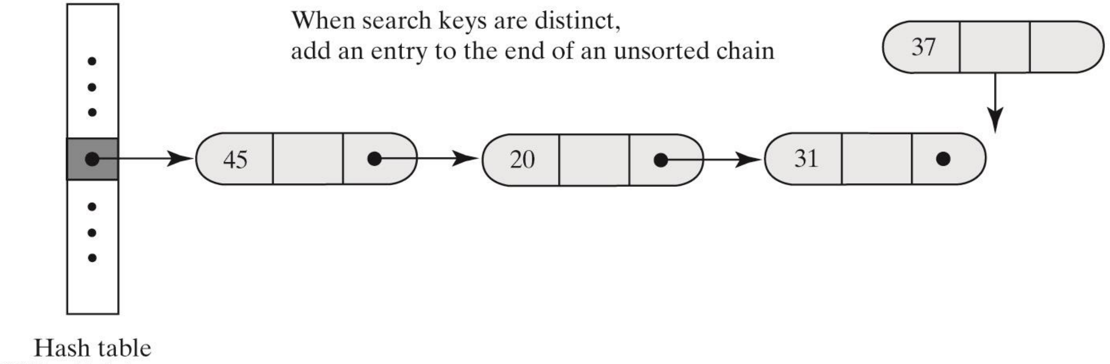


Separate Chaining

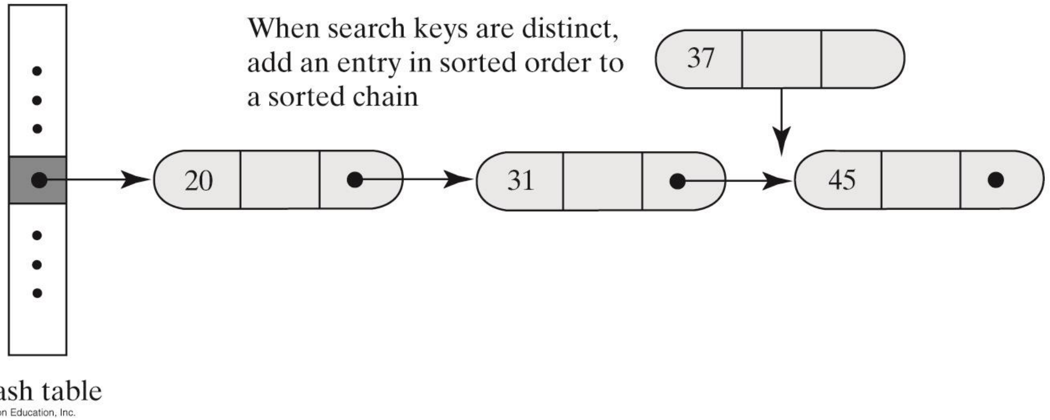
(a) Unsorted, and possibly duplicate, keys



(b) Unsorted and distinct keys



(c) Sorted and distinct keys



In-Class Exercise

- Given a table size of 19, the hash function

$$h(k) = k \ \% \ table\ size$$

For a sequence of the entries 19, 38, 20, 39 and 21, show the hash table after the five entries are inserted into the table using buckets.

In-Class Exercise

- Given a table size of 19, the hash function

$$h(k) = k \% \text{ table size}$$

For a sequence of the entries 19, 38, 20, 39 and 21, show the hash table after the five entries are inserted into the table using buckets.

19 % 19 = 0
38 % 19 = 0 << collision
20 % 19 = 1 << collision
39 % 19 = 1 << collision
21 % 19 = 2 << collision

index	0	1	2	3	4	5	6	7	8	9	10	11	12
entry	19	20	21										
	38	39											

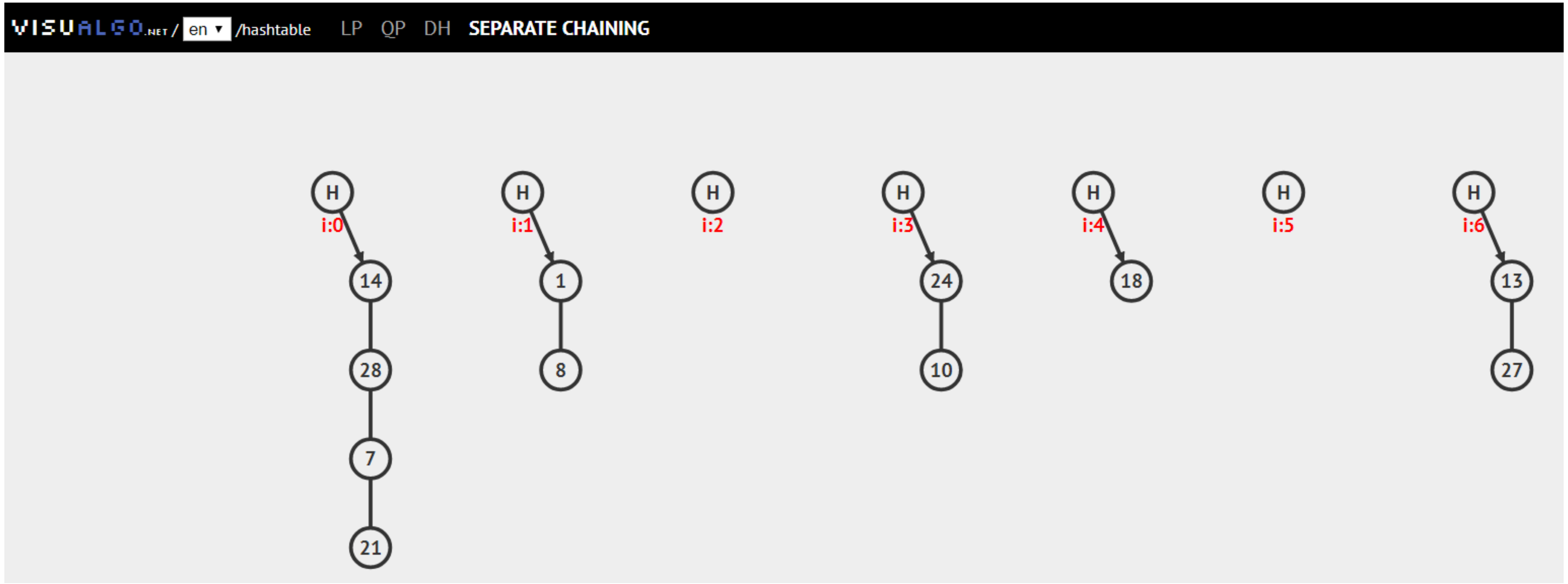
Java Class Library: HashMap and HashSet

- The standard package java.util contains the class **HashMap**<K, V>.
  - This class implements the interface java.util.Map

- The package java.util of the Java Class Library also contains the class **HashSet**<T>.
  - This class implements the interface java.util.Set

Interactive and Visualization Demos

- https://visualgo.net/en/hashtable



Summary

- Hashing
- Hashing as a Dictionary Implementation

What I Want You to Do

- Review Chapters 22 and 23