

Class 09 - Hashing

CSIS 3475 Data Structures and Algorithms

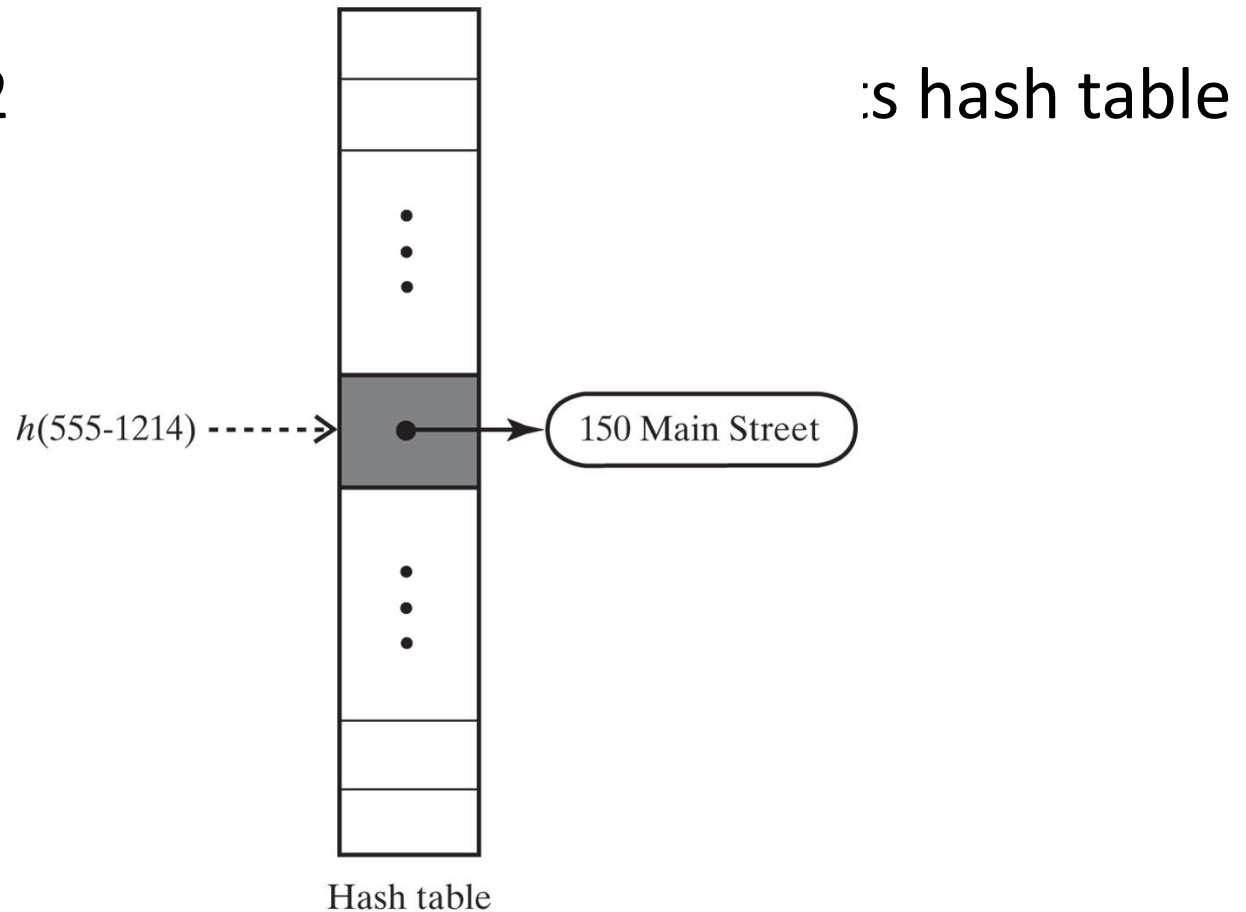
©Michael Hrybyk and others
NOT TO BE REDISTRIBUTED

Hashing

- A technique that determines an index into a table using only an entry's search key
- Hash function
 - Takes a search key and produces the integer index of an element in the hash table
 - Search key is mapped, or hashed, to the index

Hash Table

- FIGURE 22



© 2019 Pearson Education, Inc.

Ideal Hashing

- Simple algorithms for the dictionary operations that add and retrieve

Algorithm **add(key, value)**

index = h(key)

hashTable[index] = value

Algorithm **getValue(key)**

index = h(key)

return hashTable[index]

Typical Hashing

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

Algorithm `getHashIndex(phoneNumber)`

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

i = last four digits of `phoneNumber`

return `i % tableSize`

Typical Hashing

- Most hash functions are not perfect,
 - Can allow more than one search key to map into a single index
 - Causes a collision in the hash table
- Consider **tableSize = 101**
- **getHashIndex(555-1214) = 52**
- **getHashIndex(555-8132) = 52 *also!!!***

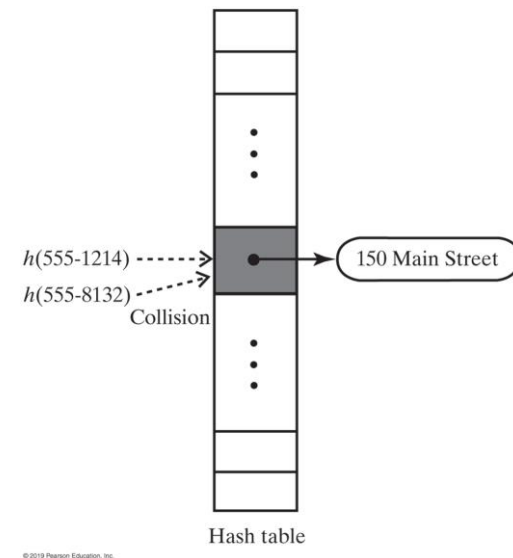


FIGURE 22-2 A collision caused by the hash function h

Hash Functions

- A good hash function should
 - Minimize collisions
 - Be fast to compute
- To reduce the chance of a collision
 - Choose a hash function that distributes entries uniformly throughout hash table.

Computing Hash Codes

- Java's base class **Object** has a method **hashCode** that returns an integer hash code
 - A class should/could define its own version of **hashCode**
- A hash code for a string
 - Using a character's Unicode integer is common
 - Better approach:
 - Multiply Unicode value of each character by factor based on character's position,
 - Then sum values

Computing Hash Codes

- Hash code for a string example:

$$u_0g^{n-1} + u_1g^{n-2} + \dots + u_{n-2}g + u_{n-1}$$

- Java code to do this:

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

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
 - Manipulate internal binary representations

Compressing a Hash Code

- 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

Compressing a Hash Code

- Hash function for the ADT dictionary

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

    return hashIndex;
} // end getHashIndex
```

Resolving Collisions

- Collision:
 - Hash function maps search key into a location in hash table already in use
- Two choices:
 - Use another location in the hash table
 - Change the structure of the hash table so that each array location can represent more than one value

Resolving Collisions

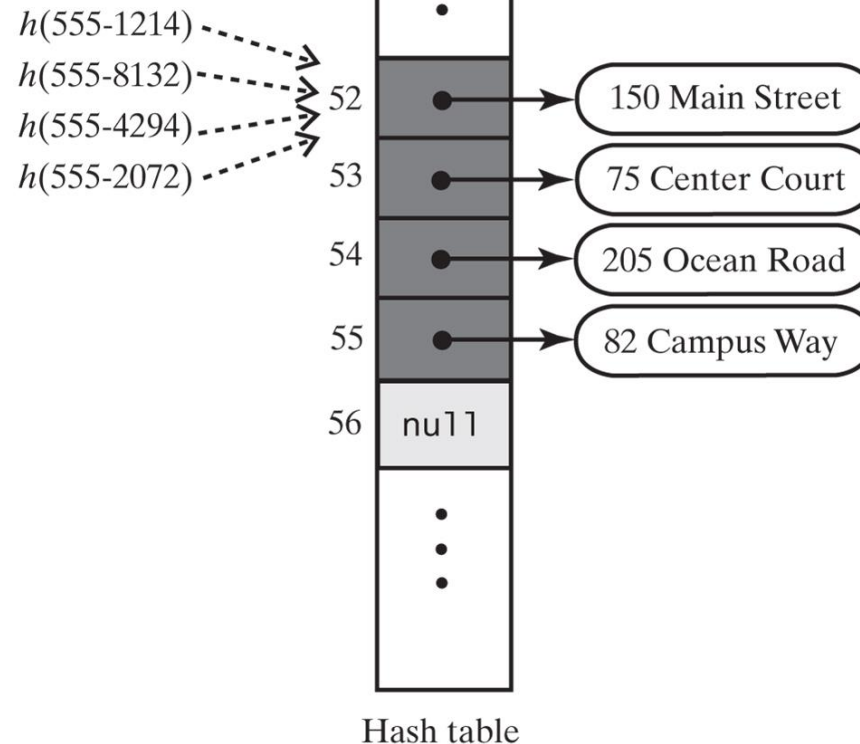
- **Linear probing**

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

- Table locations checked make up probe sequence
- If probe sequence reaches end of table, go to beginning of table (circular hash table)

Linear Probing

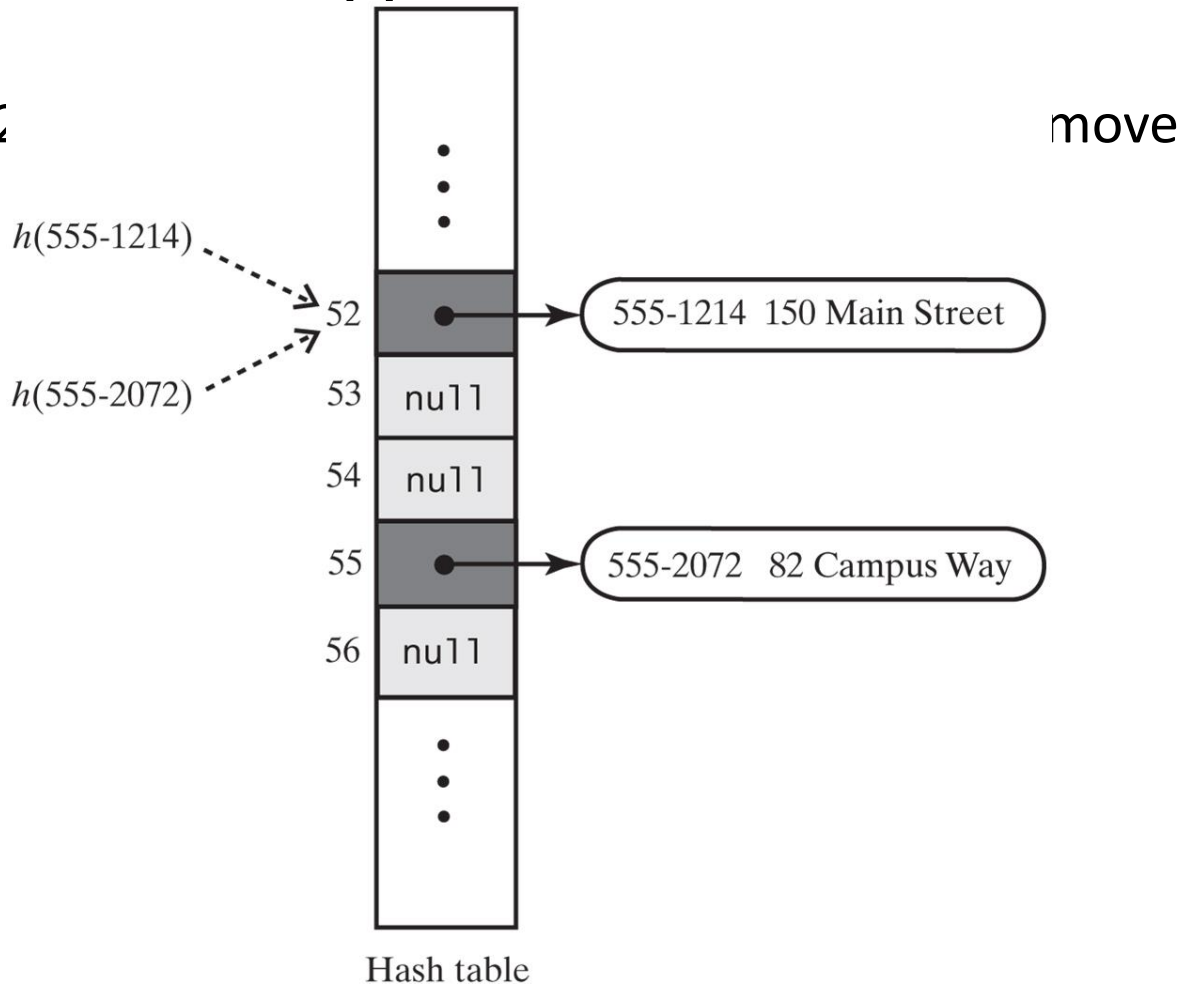
- FIGURE 22-3 T
whose search



© 2019 Pearson Education, Inc.

Linear Probing

- FIGURE 21
entries



© 2019 Pearson Education, Inc.

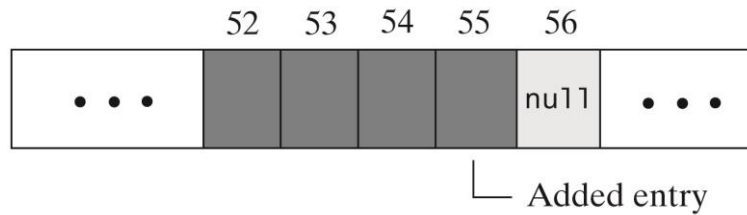
Resolving Collisions

- 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

Linear Probing

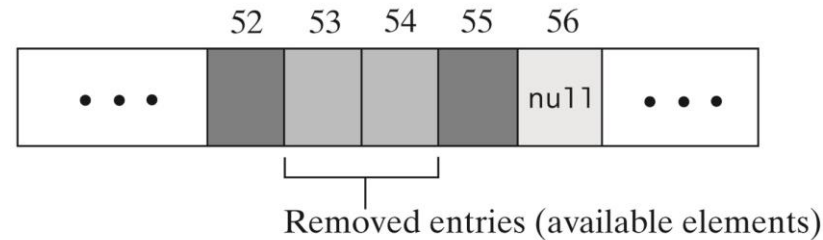
- FIGURE 22-6 The linear probe sequence in various situations

(a) After adding an entry



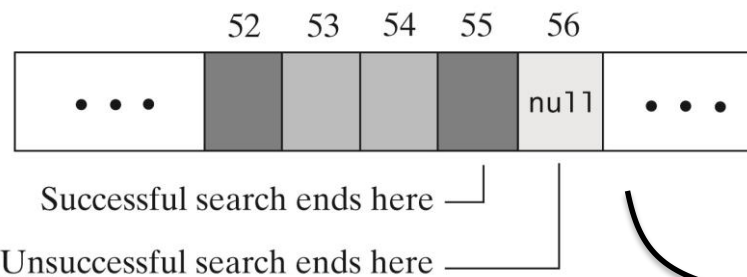
© 2019 Pearson Education, Inc.

(b) After removing two entries



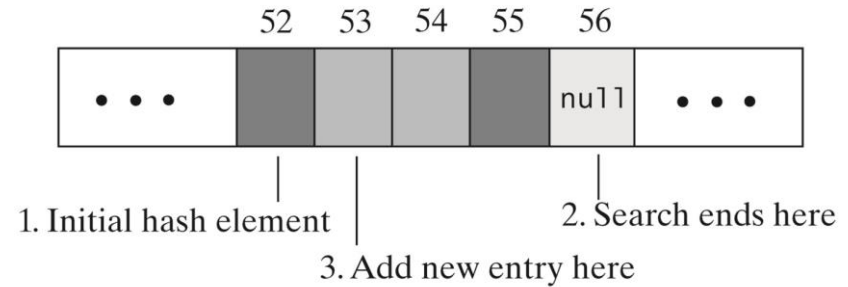
© 2019 Pearson Education, Inc.

(c) After a search



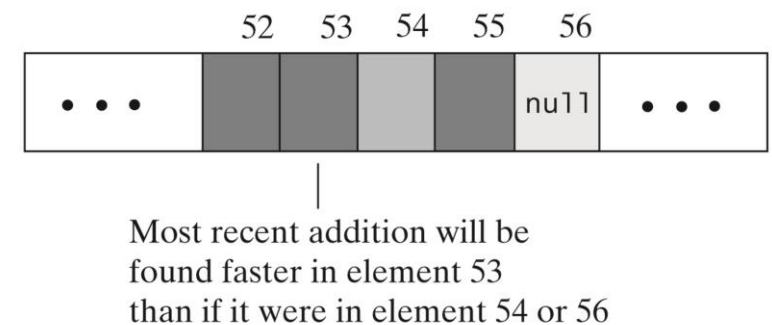
© 2019 Pearson Education, Inc.

(d) Searching for a place to add an entry



© 2019 Pearson Education, Inc.

(e) After an addition to a formerly occupied element



Dark gray = occupied with current entry
Medium gray = available element
Light gray = empty element (contains null)

© 2019 Pearson Education, Inc.

Linear Probing - Probe Algorithm

Algorithm probe(index, key)

// Searches the probe sequence that begins at index. Returns the index of either the element containing key or an available element in the hash table.

```
while (key is not found and hashTable[index] is not null)
{
    if (hashTable[index] references an entry in the dictionary)
    {
        if (the entry in hashTable[index] contains key)
            Exit loop
        else
            index = next probe index
    }
    else // hashTable[index] is available
    {
        if (this is the first available element encountered)
            availableStateIndex = index
        index = next probe index
    }
}
if (key is found or an available element was not encountered)
    return index
else
    return availableStateIndex // Index of first entry removed
```

Linear Probe Algorithm

// Precondition: checkIntegrity has been called.

```
private int linearProbe(int index, K key)
{
    boolean found = false;
    int availableStateIndex = -1; // Index of first element in available state
    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 // Element in available state; skip it, but mark the first one encountered
        {
            // Save index of first element in available state
            if (availableStateIndex == -1)
                availableStateIndex = index;
            index = (index + 1) % hashTable.length; // Linear probing
        } // end if
    } // end while
    // Assertion: Either key or null is found at hashTable[index]
    if (found || (availableStateIndex == -1) )
        return index; // Index of either key or null
    else
        return availableStateIndex; // Index of an available element
} // end linearProbe
```

Revised version of linear probe

```
private int linearProbe(int index, K key) {
    boolean found = false;

    // Index of first available location (from which an entry was removed)
    int availableIndex = -1;

    // start looking at keys at the index location, then increment until key is found
    while (!found && (hashTable[index] != null)) {
        // if there is an entry in the location test for equality
        if (hasAnEntry(index)) {
            if (key.equals(hashTable[index].getKey()))
                found = true; // Key found
        } else {
            // Skip entries that were removed
            // but save index of first location in removed state
            if (availableIndex == -1)
                availableIndex = index;
        }

        // if there was an entry but it wasn't the key, increment th
        // index and try again

        if (!found)
            index = setHashIndex(index + 1); // Linear probing
    }

    // if the key is found return the location
    // if we didn't find the key and there are only null entries, return the first
    // null entry

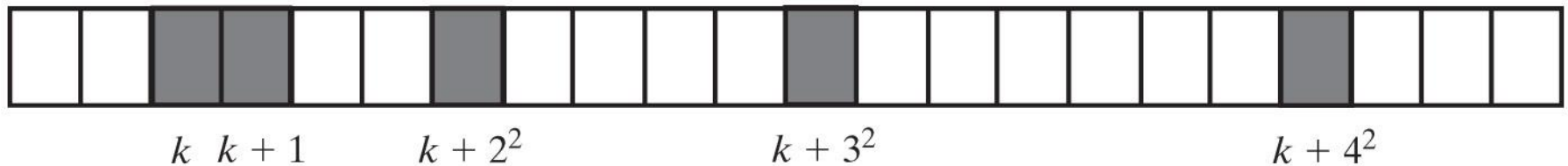
    // otherwise, return the first available index
    if (found || (availableIndex == -1))
        return index; // Index of either key or null
    else
        return availableIndex; // Index of an available location
} // end linearProbe
```

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

Open Addressing with Quadratic Probing

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



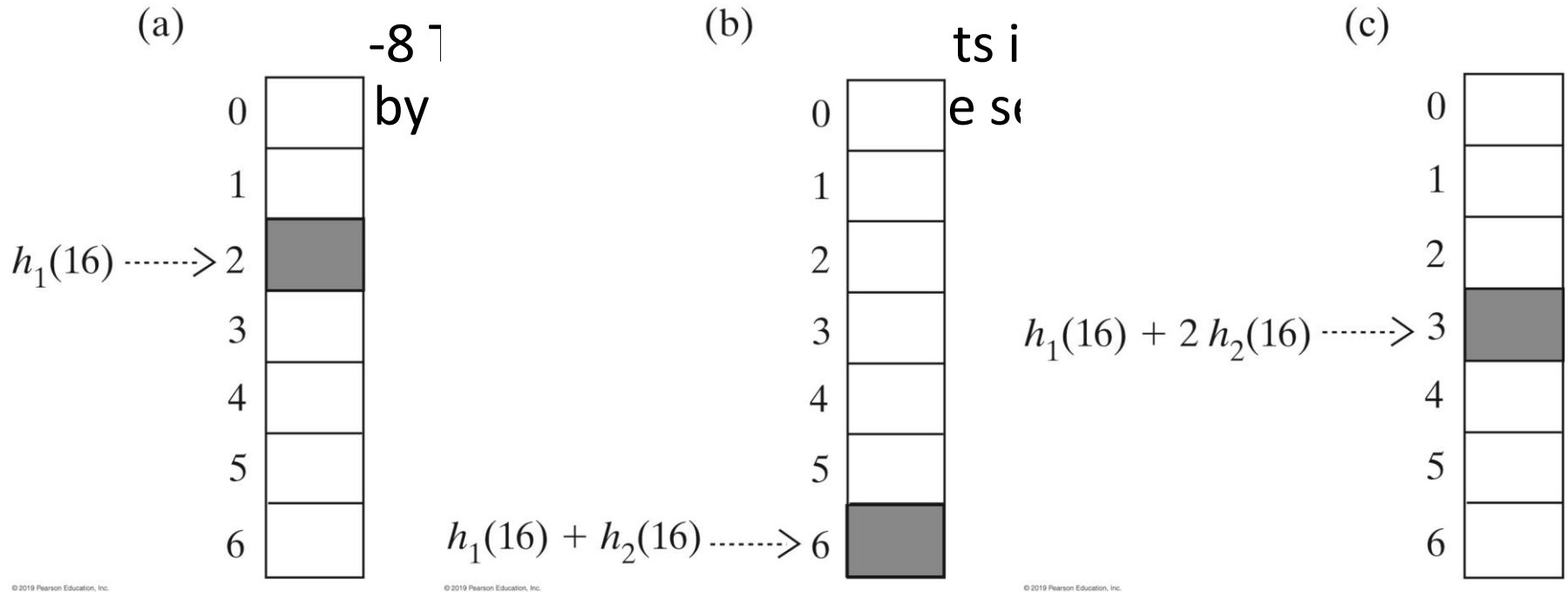
© 2019 Pearson Education, Inc.

FIGURE 22-7 A probe sequence of length five using quadratic probing

Open Addressing with Double Hashing

- Linear probing and quadratic probing add increments to k to define a probe sequence
 - Both are independent of the search key
- Double hashing uses a second hash function to compute these increments
 - This is a key-dependent method.

Open Addressing with Double Hashing



- FIGURE 22–8 The first three elements in a probe sequence generated by double hashing for the search key 16

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
- Thus searching a probe sequence will not work
- Consider separate chaining as a solution

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**

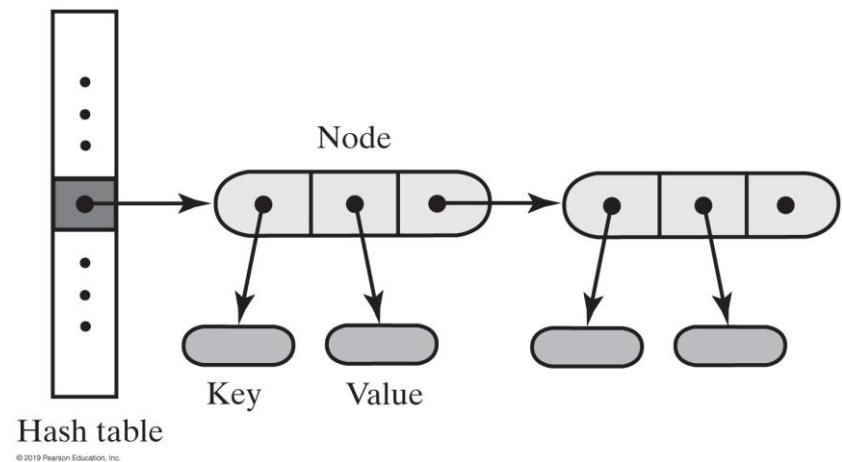
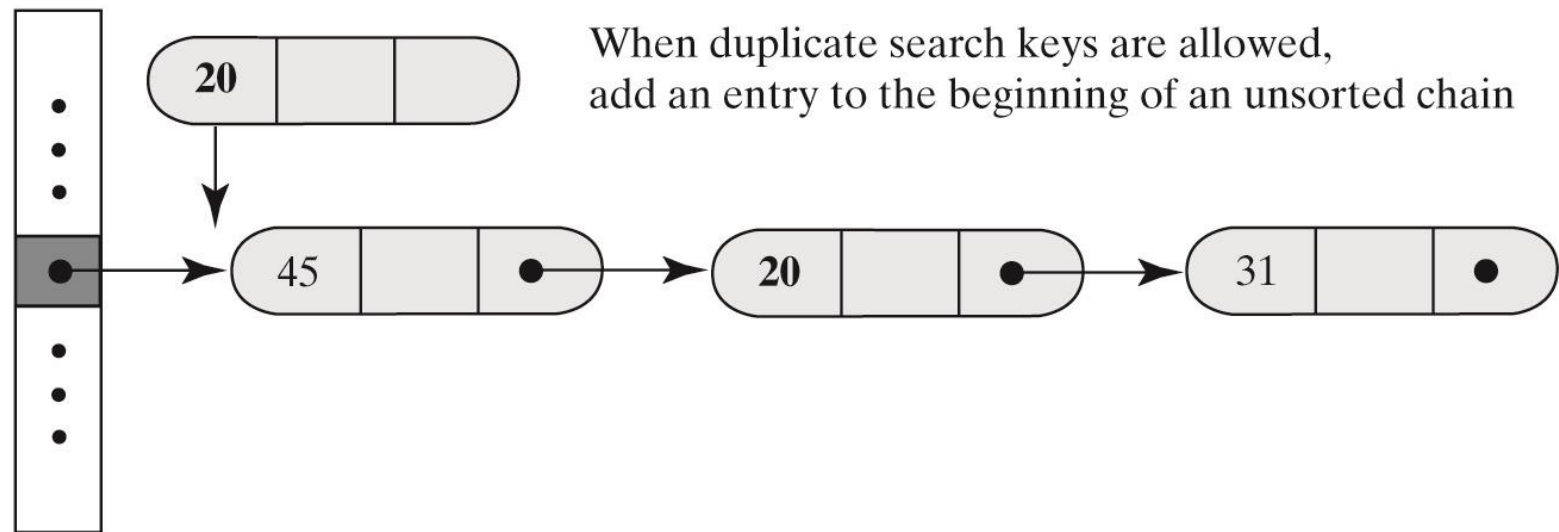


FIGURE 22-9 A hash table for use with separate chaining; each bucket is a chain of linked nodes

Separate Chaining

- FIGURE 22-10a Inserting a new entry into a linked bucket according to the nature of the integer search keys

(a) Unsorted, and possibly duplicate, keys



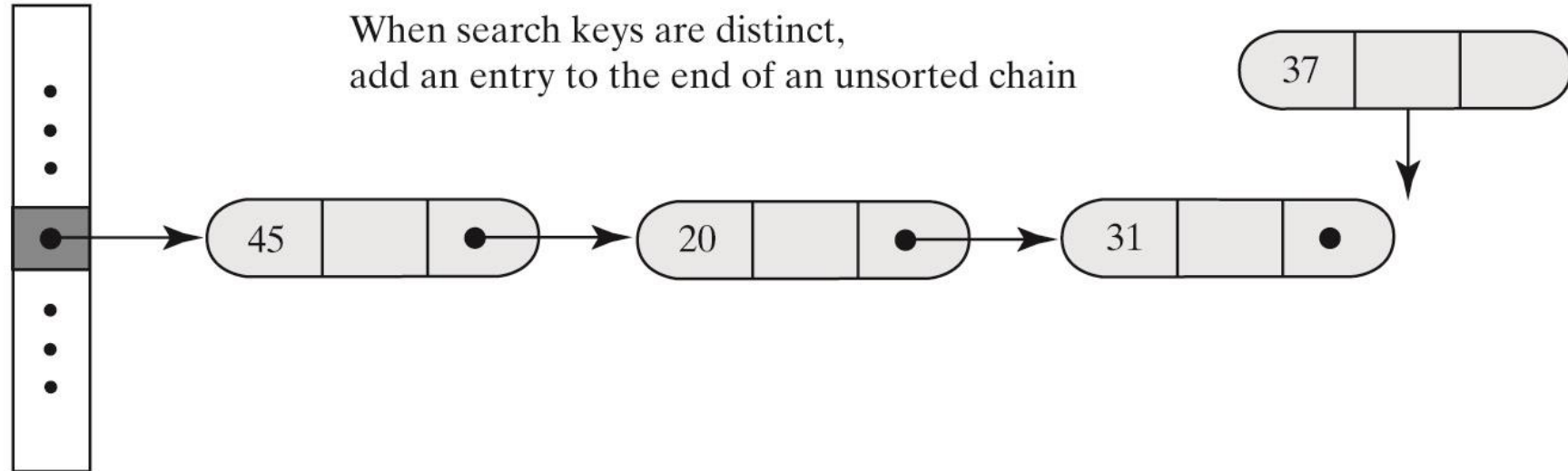
Hash table

© 2019 Pearson Education, Inc.

Separate Chaining

- FIGURE 22-10b Inserting a new entry into a linked bucket according to the nature of the integer search keys

(b) Unsorted and distinct keys



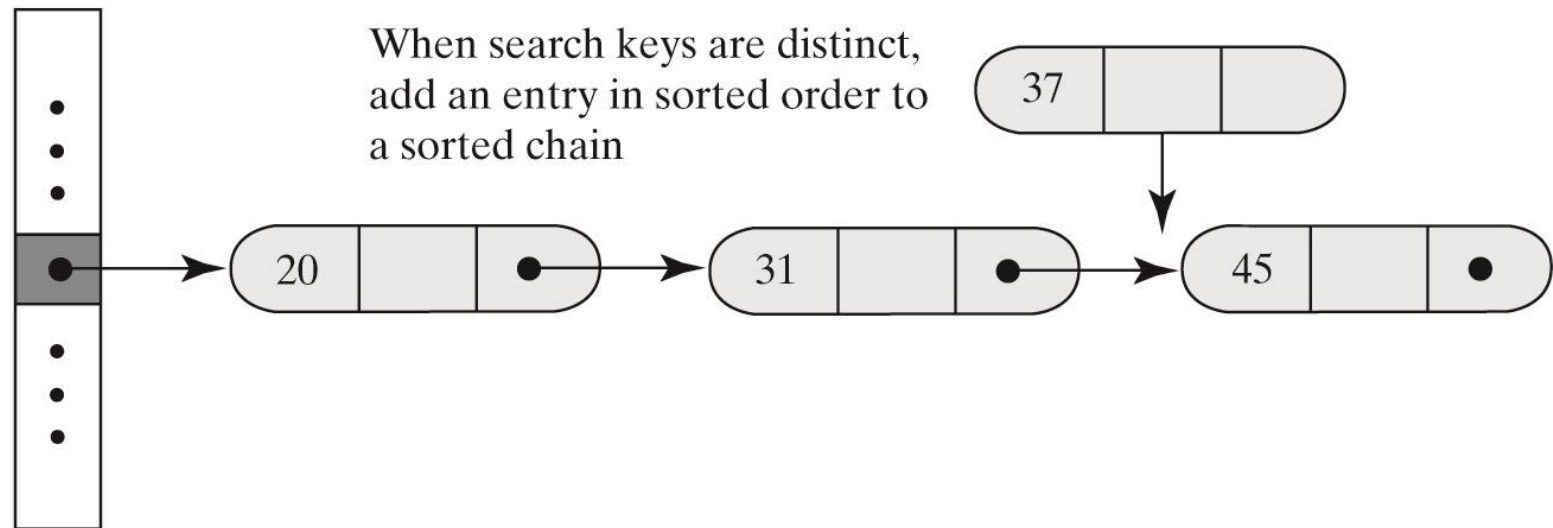
Hash table

© 2019 Pearson Education, Inc.

Separate Chaining

- FIGURE 22-10c Inserting a new entry into a linked bucket according to the nature of the integer search keys

(c) Sorted and distinct keys



Hash table

© 2019 Pearson Education, Inc.

Separate Chaining

Algorithm for the dictionary's `add` method.

```
Algorithm add(key, value)
index = getHashIndex(key)
if (hashTable[index] == null)
{
    hashTable[index] = new Node(key, value)
    numberOfEntries++
    return null
}
else
{
    Search the chain that begins at hashTable[index] for a node that contains key
    if (key is found)
    { // Assume currentNode references the node that contains
        key oldValue = currentNode.getValue()
        currentNode.setValue(value)
        return oldValue
    }
    else // Add new node to end of chain
    { // Assume nodeBefore references the last node
        newNode = new Node(key, value)
        nodeBefore.setNextNode(newNode) numberOfEntries++
        return null
    }
}
```

Separate Chaining

- Algorithm for the dictionary's `remove` method.

Algorithm `remove(key)`

`index = getHashIndex(key)`

Search the chain that begins at `hashTable[index]` for a node that contains `key`

if (*key is found*)

```
{  
    Remove the node that contains key from the chain  
    numberOfEntries--  
    return value in removed node  
}
```

}

else

return null

Separate Chaining

- Algorithm for the dictionary's `getValue` method.

Algorithm `getValue(key)`

`index = getHashIndex(key)`

Search the chain that begins at `hashTable[index]` for a node that contains key

if (*key is found*)

return *value in found node*

else

return `null`

Efficiency of Hashing

- Observations about the time efficiency of these operations
 - Successful retrieval/removal has same efficiency as successful search
 - Unsuccessful retrieval/removal has same efficiency as unsuccessful search
 - Successful addition has same efficiency as unsuccessful search
 - Unsuccessful addition has same efficiency as successful search

Load Factor

- Definition of load factor:

$$\lambda = \frac{\textit{Number of entries in the dictionary}}{\textit{Number of locations in the hash table}}$$

- Never negative
- For open addressing, $1 \geq \lambda$
- For separate chaining, λ has no maximum value
- Restricting size of λ improves performance

Cost of Open Addressing

- Average number of searches for linear probing

For unsuccessful search: $\frac{1}{2} \left\{ 1 + \frac{1}{(1 - \lambda)^2} \right\}$

For successful search: $\frac{1}{2} \left\{ 1 + \frac{1}{(1 - \lambda)} \right\}$

Cost of Open Addressing

- FIGURE 23-1 The average number of comparisons required by a search of the hash table for given values of the load factor λ when using linear probing

λ	Unsuccessful Search	Successful Search
0.1	1.1	1.1
0.3	1.5	1.2
0.5	2.5	1.5
0.7	6.1	2.2
0.9	50.5	5.5

Quadratic Probing and Double Hashing

- Average number of comparisons needed

For unsuccessful search: $\frac{1}{(1 - \lambda)}$

For successful search: $\frac{1}{\lambda} \log\left(\frac{1}{1 - \lambda}\right)$

Quadratic Probing and Double Hashing

- FIGURE 23-2 The average number of comparisons required by a search of the hash table for given values of the load factor λ when using either quadratic probing or double hashing

λ	Unsuccessful Search	Successful Search
0.1	1.1	1.1
0.3	1.5	1.2
0.5	2.0	1.4
0.7	3.3	1.7
0.9	10.0	2.6

Cost of Separate Chaining

- Average number of comparisons during a search when separate chaining is used

For unsuccessful search: λ

For successful search: $1 + \lambda/2$

To maintain reasonable efficiency, you should keep $\lambda < 1$.

Cost of Separate Chaining

- FIGURE 23-3 The average number of comparisons required by a search of the hash table for given values of the load factor λ when using separate chaining

λ	Unsuccessful Search	Successful Search
0.1	0.1	1.1
0.3	0.3	1.2
0.5	0.5	1.3
0.7	0.7	1.4
0.9	0.9	1.5
1.1	1.1	1.6
1.3	1.3	1.7
1.5	1.5	1.8
1.7	1.7	1.9
1.9	1.9	2.0
2.0	2.0	2.0

Maintaining the Performance of Hashing

- To maintain efficiency, restrict the size of λ as follows:
 - $\lambda < 0.5$ for open addressing
 - $\lambda < 1.0$ for separate chaining
- Should the load factor exceed these bounds
 - Increase the size of the hash table

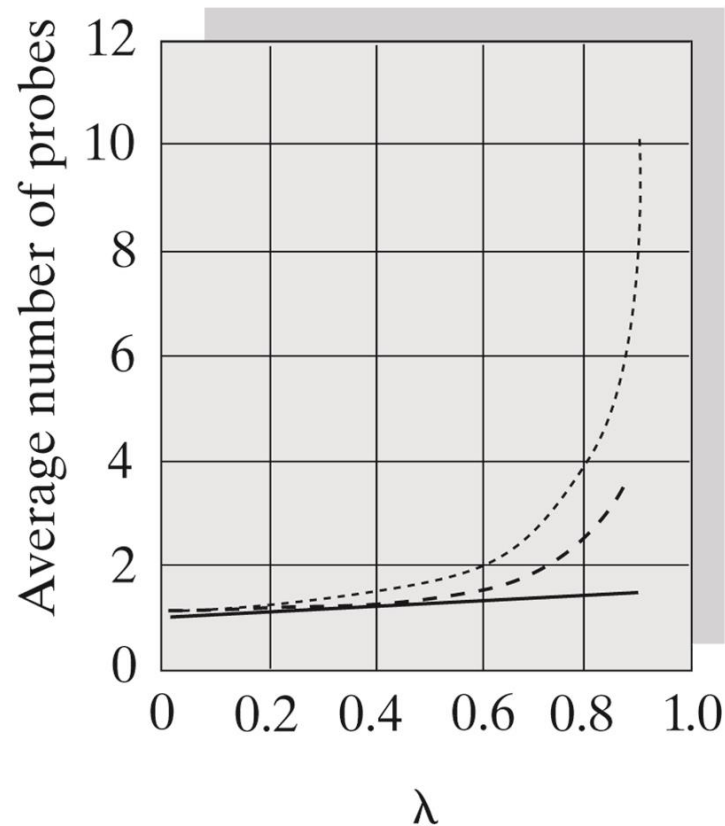
Rehashing

- When the load factor λ becomes too large must resize the hash table
- Compute the table's new size
 - Double its present size
 - Increase the result to the next prime number
 - Use method `add` to add the current entries in dictionary to new hash table

Comparing Schemes for Collision Resolution

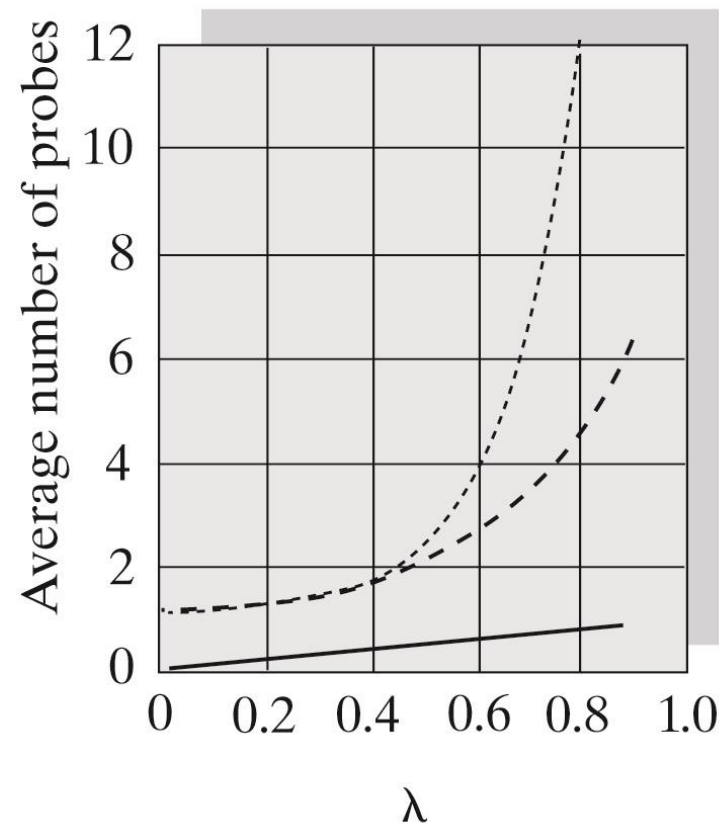
- FIGURE 23-4 The average number of comparisons required by a search of the hash table versus the load factor λ for four collision resolution techniques

(a) Successful search



© 2019 Pearson Education, Inc.

(b) Unsuccessful search

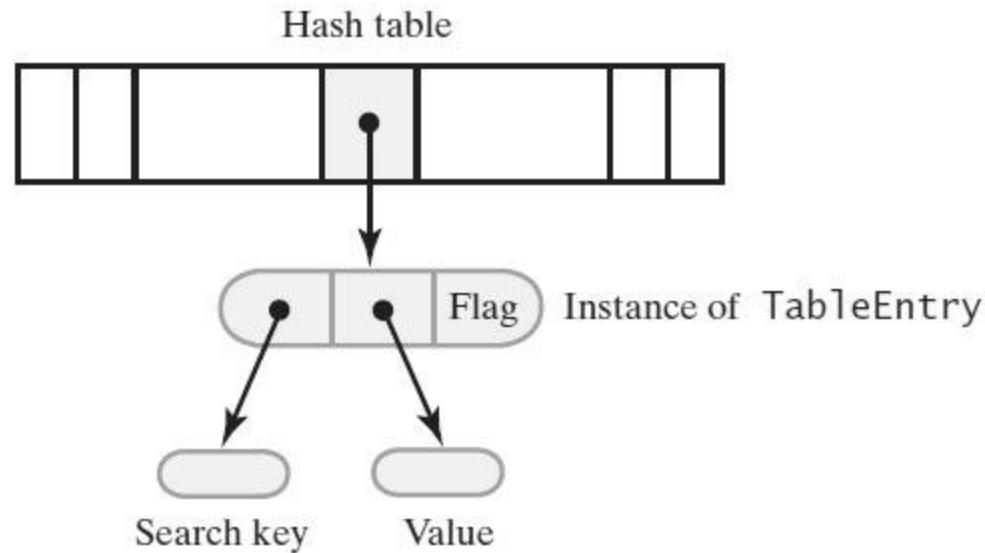


© 2019 Pearson Education, Inc.

- Linear probing
- Quadratic probing or double hashing
- Separate chaining

Dictionary Implementation That Uses Hashing

- FIGURE 23-5 A hash table and one of its entry objects



HashedDictionary

- Keep an internal table of entries
- Mark an entry as AVAILABLE as an entry with null values.
 - Note this can't be changed
- Define a load factor above which the table will be increased in size

```
public class HashedDictionary<K, V> implements DictionaryInterface<K, V> {  
    // The dictionary:  
    private int numberOfEntries;  
  
    // capacity must be prime, which checkCapacity will automatically set  
    // if this is set too low with quadratic probe, search time increases.  
    private static final int DEFAULT_CAPACITY = 5;  
    private static final int MAX_CAPACITY = 10000;  
  
    // The hash table:  
    private Entry<K, V>[] hashTable;  
  
    private static final int MAX_SIZE = 2 * MAX_CAPACITY; // Max size of hash table  
    private static final double MAX_LOAD_FACTOR = 0.5; // Fraction of hash table that can be filled  
  
    private final Entry<K, V> AVAILABLE = new Entry<>(null, null);
```

Use a key/value Entry class

- Simple generic class to keep keys and values

```
protected final class Entry<K, V> {  
    private K key;  
    private V value;  
  
    private Entry(K searchKey, V dataValue) {  
        key = searchKey;  
        value = dataValue;  
    } // end constructor  
  
    private K getKey() {  
        return key;  
    } // end getKey  
  
    private V getValue() {  
        return value;  
    } // end getValue  
  
    private void setValue(V newValue) {  
        value = newValue;  
    } // end setValue  
} // end Entry
```

Constructor

- Make the table size the next prime number up from what was asked.
- Also that the table has enough space re load factor (checkCapacity).

```
public HashedDictionary(int initialCapacity) {
    initialCapacity = checkCapacity(initialCapacity);
    numberOfEntries = 0; // Dictionary is empty

    // Set up hash table:
    // Initial size of hash table is same as initialCapacity if it is prime;
    // otherwise increase it until it is prime size
    int tableSize = getNextPrime(initialCapacity);
    checkSize(tableSize); // Check that the prime size is not too large

    // The cast is safe because the new array contains null entries
    @SuppressWarnings("unchecked")
    Entry<K, V>[] temp = (Entry<K, V>[]) new Entry[tableSize];
    hashTable = temp;
} // end constructor
```


Getting an index in the hash table

- Hash the key, and get an integer.
 - Adjust this so it is modulo the table length, so the hashes are distributed. (setHashIndex()).
 - This will be the starting index
- Then probe starting at the index and look for the key (see next slide)
- Returns a new index of an available entry or key itself.
- Remember, an available entry is either null or the AVAILABLE entry (null key and value).

```
/**
 * Get the next available hash index for the key
 * @param key
 * @return
 */
private int getHashIndex(K key) {
    int hashIndex = setHashIndex(key.hashCode());

    // Check for and resolve collision
    hashIndex = linearProbe(hashIndex, key);

    return hashIndex;
} // end getHashIndex

/**
 * Take a hashcode and make sure it fits in the hash table.
 * Wraparound if necessary using mod tablelength.
 * If the resulting index is < 0, add the table length to it.
 * @param index
 * @return index % tablelength
 */
private int setHashIndex(int index) {
    index = index % hashCode.length;
    if(index < 0)
        index = index + hashCode.length;

    return index;
}
```

Revised version of linear probe

```
private int linearProbe(int index, K key) {
    boolean found = false;

    // Index of first available location (from which an entry was removed)
    int availableIndex = -1;

    // start looking at keys at the index location, then increment until key is found
    while (!found && (hashTable[index] != null)) {
        // if there is an entry in the location test for equality
        if (hasAnEntry(index)) {
            if (key.equals(hashTable[index].getKey()))
                found = true; // Key found
        } else {
            // Skip entries that were removed
            // but save index of first location in removed state
            if (availableIndex == -1)
                availableIndex = index;
        }

        // if there was an entry but it wasn't the key, increment th
        // index and try again

        if (!found)
            index = setHashIndex(index + 1); // Linear probing
    }

    // if the key is found return the location
    // if we didn't find the key and there are only null entries, return the first
    // null entry

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

Add method

- Hash the key and get the next available index
- If the slot is free, add the entry to the hash table
- If the key already exists, replace the value

```
public V add(K key, V value) {  
    if ((key == null) || (value == null))  
        throw new IllegalArgumentException("Cannot add null to a dictionary.");  
    else {  
        V oldValue; // Value to return  
  
        // get the next available hash index for the key  
        int index = getHashIndex(key);  
  
        if (!hasAnEntry(index)) { // 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
```

getValue algorithm for retrieval

Algorithm getValue(key)

// Returns the value associated with the given search key, if it is in the dictionary.

// Otherwise, returns null.

if (key *is found*)

return *value in found entry*

else

return null

getValue implementation

- Get the index by hashing the key
- If it exists, then just get the value using Entry getValue() method.

```
public V getValue(K key) {  
    V result = null;  
  
    int index = getHashIndex(key);  
  
    if (hasAnEntry(index))  
        result = hashTable[index].getValue(); // Key found; get value  
  
    return result;  
}
```

Pseudocode for method `remove`

Algorithm `remove(key)`

// Removes a specific entry from the dictionary, given its search key.

*// Returns either the value that was associated with the search key or null if no such object
// exists.*

`removedValue = null`

`index = getHashIndex(key)`

if (*key is found*) *// hashCode[index] is not null and does not equal AVAILABLE*

```
{  
    removedValue = hashCode[index].getValue()  
    hashCode[index] = AVAILABLE  
    numberOfEntries--  
}
```

return removedValue

remove method

- Hash the key and get the index.
- If it exists, get the value, then set the slot to AVAILABLE
 - key and value are null

```
public V remove(K key) {  
    V removedValue = null;  
  
    int index = getHashIndex(key); // get the location in the table  
  
    if (hasAnEntry(index)) {  
        // Key found; flag entry as removed and return its value  
        removedValue = hashTable[index].getValue();  
        hashTable[index] = AVAILABLE;  
        numberOfEntries--;  
    }  
  
    return removedValue;  
}
```

Algorithm for adding a new entry

Algorithm **add(key, value)**

*// Adds a new key-value entry to the dictionary. If key is already in the dictionary,
// returns its corresponding value and replaces it in the dictionary with value.*

if ((key == **null**) or (value == **null**))

Throw an exception

index = getHashIndex(key)

if (key is not found)

{ *// Add entry to hash table*

hashTable[index] = **new** Entry(key, value)

numberOfEntries++

oldValue = **null**

}

else *// Search key is in table; replace and return entry's value*

{

oldValue = hashTable[index].getValue()

hashTable[index].setValue(value)

}

// Ensure that hash table is large enough for another addition

if (hash table is too full)

Enlarge hash table

return oldValue

Increase hash table size

- Create a new table with a set of null entries
- Save the old one.
- Important: copy by iterating through the old table and **add()** the key/value pairs to the new one.
 - This **rehashes** all of the keys, so it is not an exact duplicate.

```
private void enlargeHashTable() {
    Entry<K, V>[] oldTable = hashTable;
    int oldSize = hashTable.length;
    int newSize = getNextPrime(oldSize + oldSize);
    checkSize(newSize); // Check that the prime size is not too large

    // The cast is safe because the new array contains null entries
    // increase the size of the array
    @SuppressWarnings("unchecked")
    Entry<K, V>[] tempTable = (Entry<K, V>[]) new Entry[newSize];

    // the internal table is now a larger array, but empty

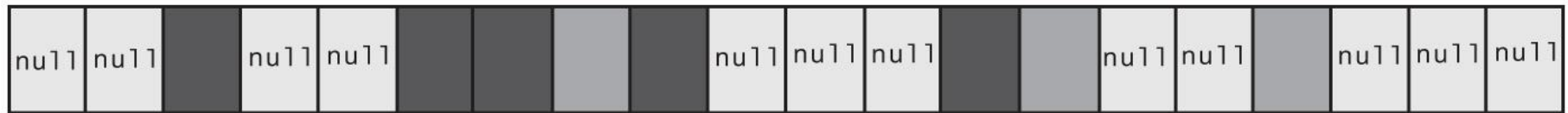
    hashTable = tempTable;
    numberOfEntries = 0; // Reset number of dictionary entries, since
                        // it will be incremented by add during rehash

    // Rehash dictionary entries from old array to the new and bigger array;
    // skip both null locations and removed entries
    // note use of add() to do this which rehashes keys

    for (int index = 0; index < oldSize; index++) {
        if ((oldTable[index] != null) && (oldTable[index] != AVAILABLE))
            add(oldTable[index].getKey(), oldTable[index].getValue());
    }
}
```

Hash Tables and Iterators

- FIGURE 23-5 A hash table containing occupied elements, available elements, and null values



Dark gray = occupied with current entry
Medium gray = available location
Light gray = empty location (null)

© 2019 Pearson Education, Inc.

Key iterator

```
private class KeyIterator implements Iterator<K> {
    private int currentIndex; // Current position in hash table
    private int numberLeft; // Number of entries left in iteration

    private KeyIterator() {
        currentIndex = 0;
        numberLeft = numberOfEntries;
    } // end default constructor

    public boolean hasNext() {
        return numberLeft > 0;
    } // end hasNext

    public K next() {
        K result = null;

        if (hasNext()) {
            // Skip table locations that do not contain a current entry
            while (!hasAnEntry(currentIndex)) {
                currentIndex++;
            } // end while

            result = hashTable[currentIndex].getKey();
            numberLeft--;
            currentIndex++;
        } else
            throw new NoSuchElementException();

        return result;
    } // end next

    public void remove() {
        throw new UnsupportedOperationException();
    } // end remove
} // end KeyIterator
```

Value iterator

```
private class ValueIterator implements Iterator<V> {
    private int currentIndex;
    private int numberLeft;

    private ValueIterator() {
        currentIndex = 0;
        numberLeft = numberOfEntries;
    } // end default constructor

    public boolean hasNext() {
        return numberLeft > 0;
    } // end hasNext

    public V next() {
        V result = null;

        if (hasNext()) {
            // Skip table locations that do not contain a current entry
            while (!hasAnEntry(currentIndex)) {
                currentIndex++;
            } // end while

            result = hashTable[currentIndex].getValue();
            numberLeft--;
            currentIndex++;
        } else
            throw new NoSuchElementException();

        return result;
    } // end next

    public void remove() {
        throw new UnsupportedOperationException();
    } // end remove
} // end ValueIterator
```

Java Class Library: The Class HashMap

- Hash table is a collection of buckets
 - Each bucket contains several entries
- Variety of constructors provided
- Default maximum load factor of 0.75
 - When limit exceeded, size of table increased by rehashing
- Possible to avoid rehashing by setting number of buckets initially larger

Java Class Library: The Class `HashSet`

- Implements the interface **`java.util.Set`**
- **`HashSet`** uses an instance of the class **`HashMap`**
- Variety of constructors provided in class