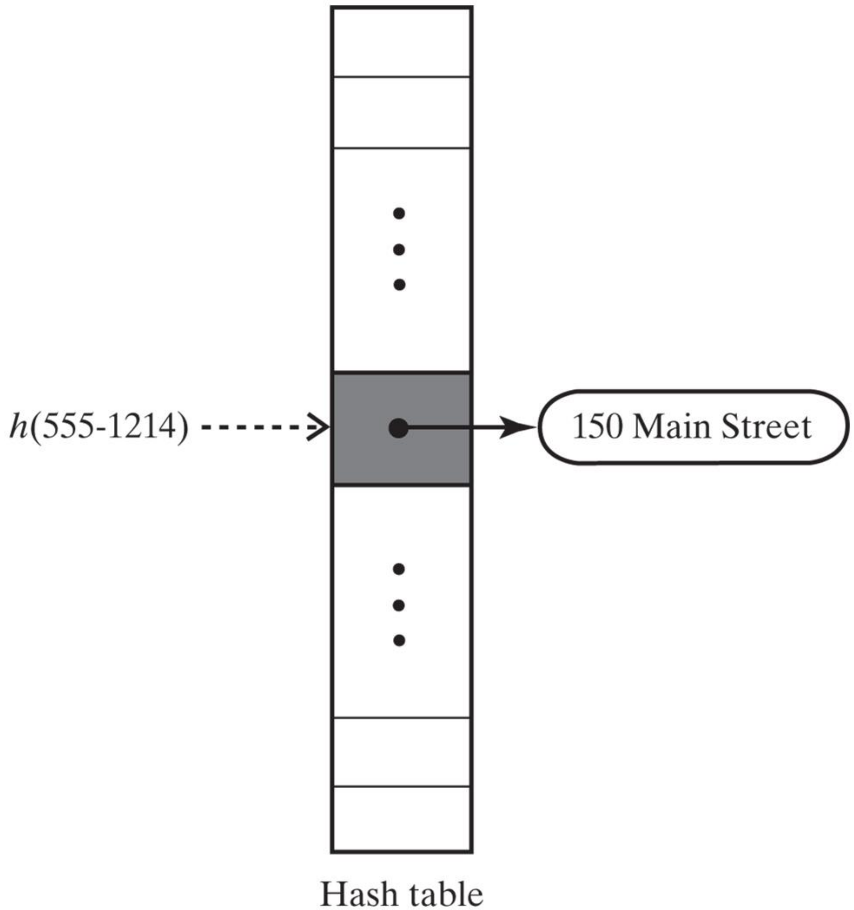
Class 09 - Hashing

CSIS 3475 Data Structures and Algorithms

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

A hash function indexes its hash table



Ideal Hashing

- Simple algorithms for the dictionary operations that add and retrieve

We can use Linked List for Hash Table but it is going to be really slow.

***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.
  - Typically done with modulo operator

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!!!***

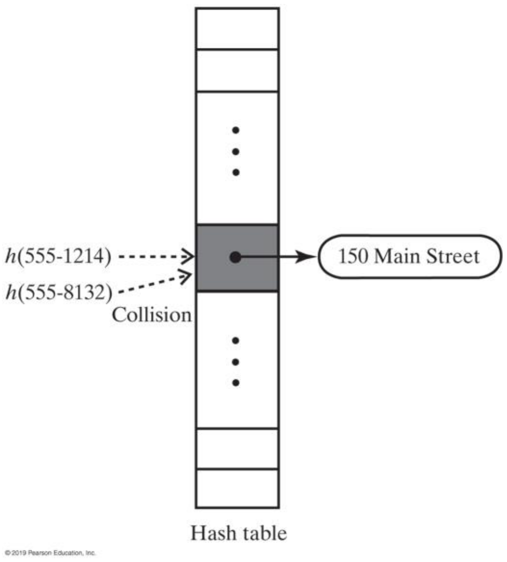


FIGURE 22-2 A collision caused by the hash function *h*

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:

  $$u_0 g^{n-1} + u_1 g^{n-2} + \ldots + u^{n-2} g + u_{n-1}$$

- Java code to do this:

```java
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
- Note that if the hashCode is negative, the index will be negative, so add the table length

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

  return hashIndex;
} // end getHashIndex
```

Resolving Collisions

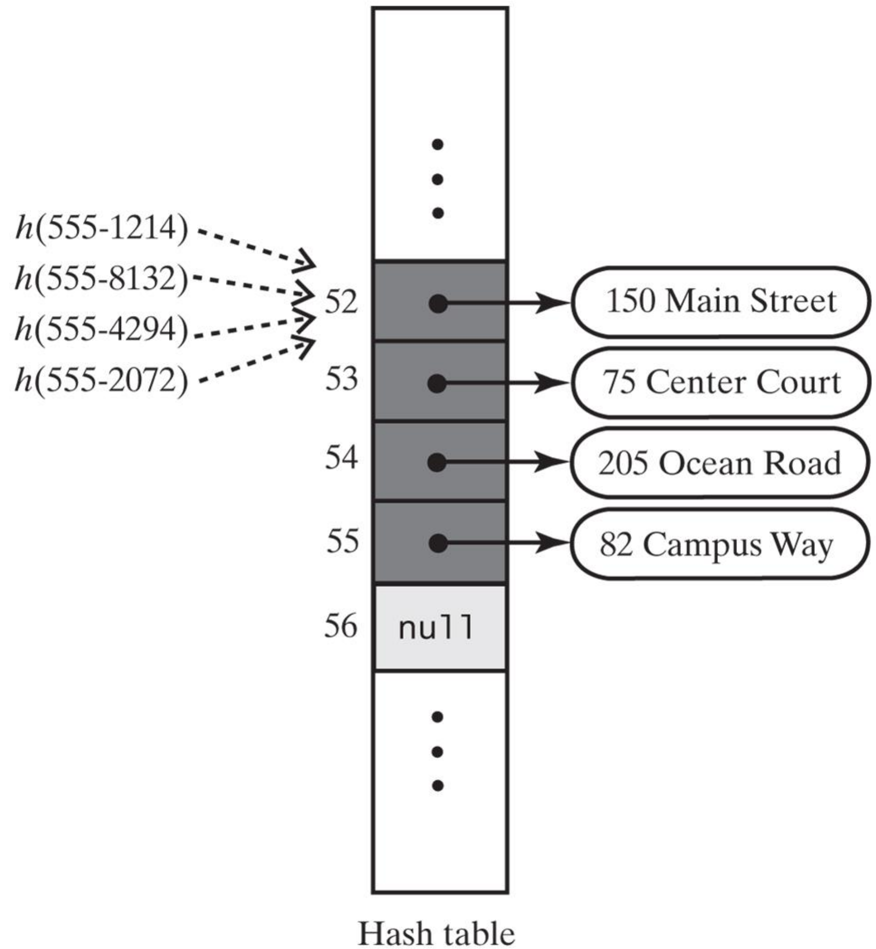
- Collision:
  - Hash function maps search key into a location in hash table already in use
- Two choices:
  - Use another location in the hash table
  - Change the structure of the hash table so that each array location can represent more than one value

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)

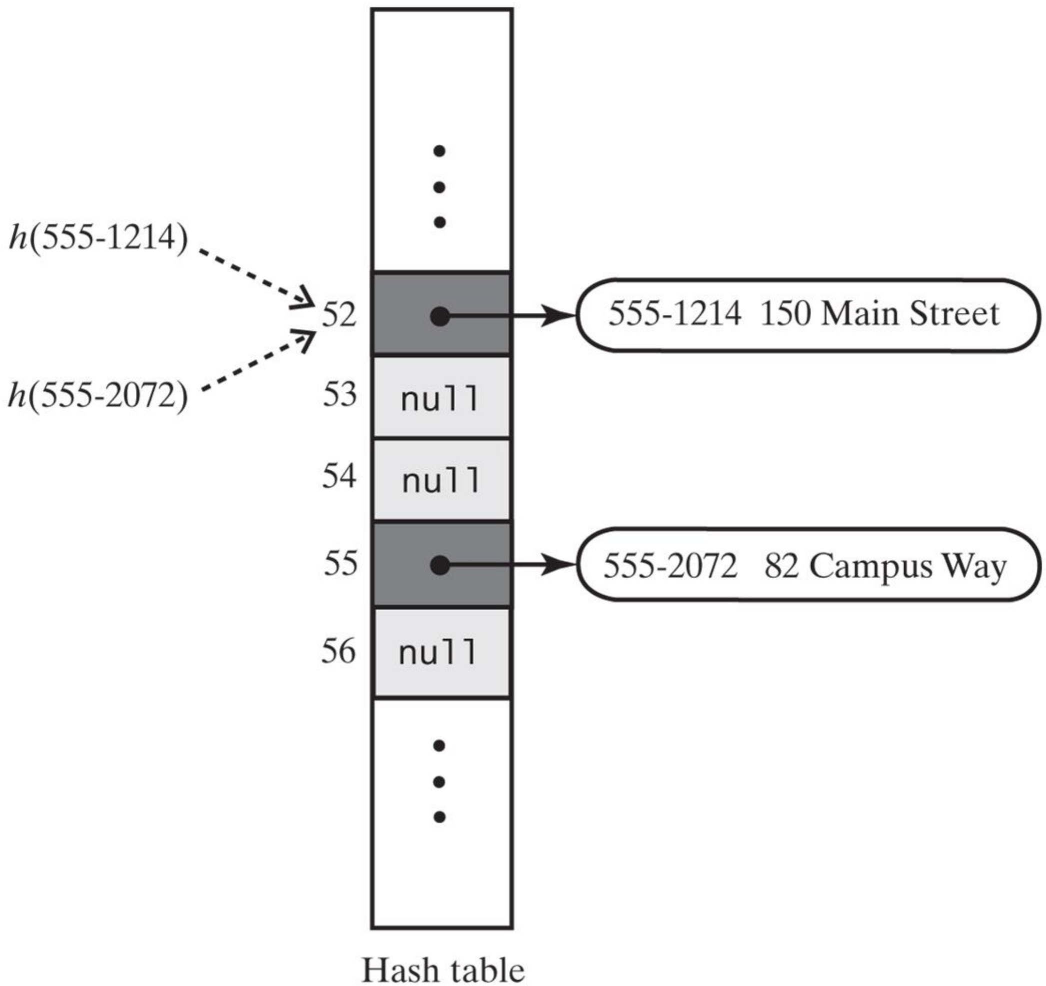
Linear Probing

- The effect of linear probing after adding four entries whose search keys hash to the same index



Linear Probing

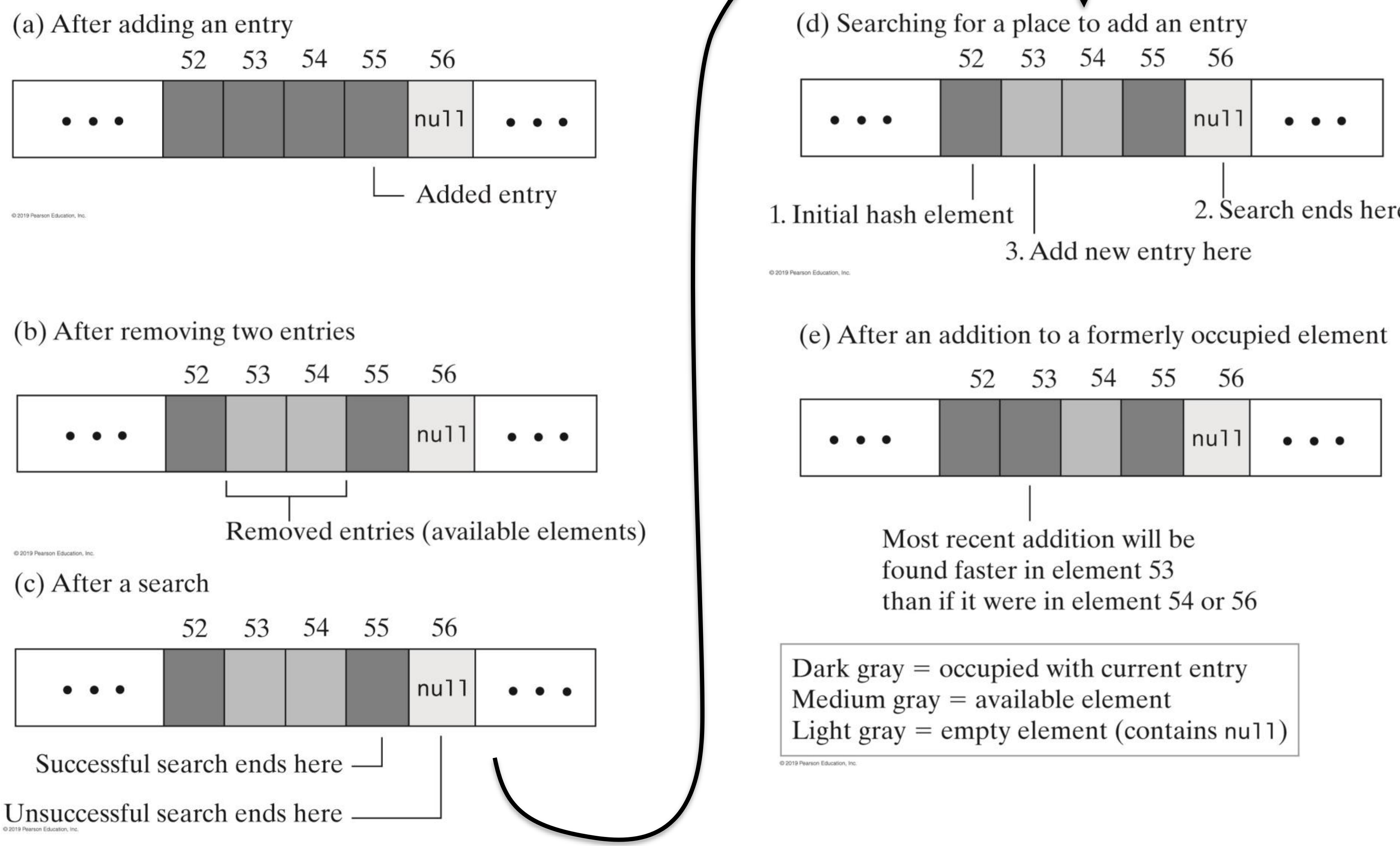
- A hash table if remove used `null` to remove entries



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

The linear probe sequence in various situations



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 implementation

```
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
}
```

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, …`

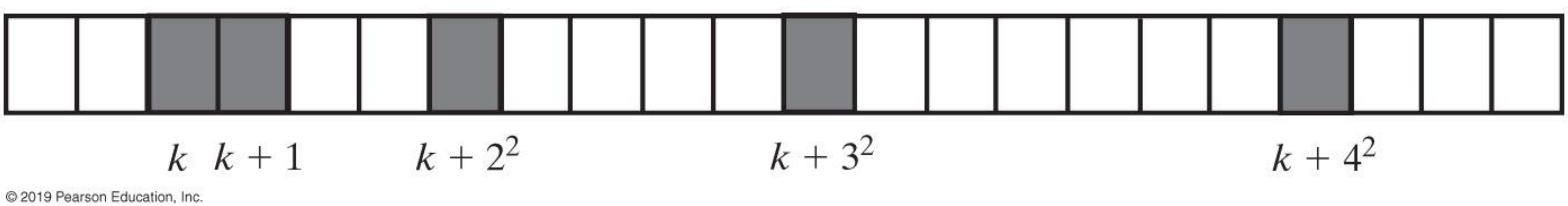


FIGURE 22-7 A probe sequence of length five using quadratic probing

Quadratic probing implementation

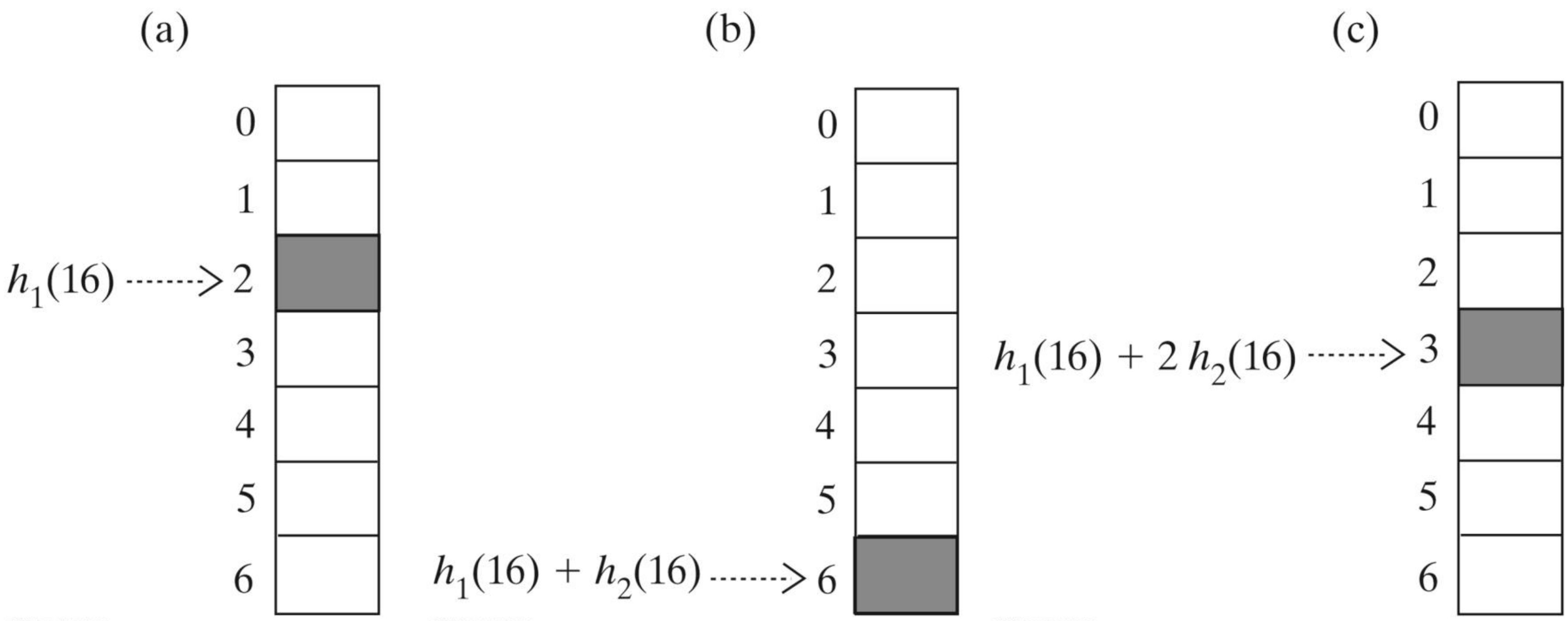
```
    private int quadraticProbe(int index, K key) {
        boolean found = false;
        int availableIndex = -1; // Index of first available location (from which an entry was removed)
        int increment = 1; // For quadratic probing

        // start looking at keys at the index location, then increment
        while (!found && (hashTable[index] != null)) {

            // if the location is not null and the entry key/value is not null
            // then test for equality

            if (hasAnEntry(index)) {
                if (key.equals(hashTable[index].getKey()))
                    found = true; // Key found
            } else {
                // Skip entries that were removed
                // Save index of first location in removed state
                if (availableIndex == -1)
                    availableIndex = index;
            }

            // not found, set index to next offset by increment (wrap around if nec)
            // then increase the increment by 2
            // this is quadratic probing

            if (!found) {
                index = setHashIndex(index + increment);
                increment = increment + 2; // Odd values for quadratic probing
            }
        }

        // if we have found it or the table is empty return the index

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

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

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

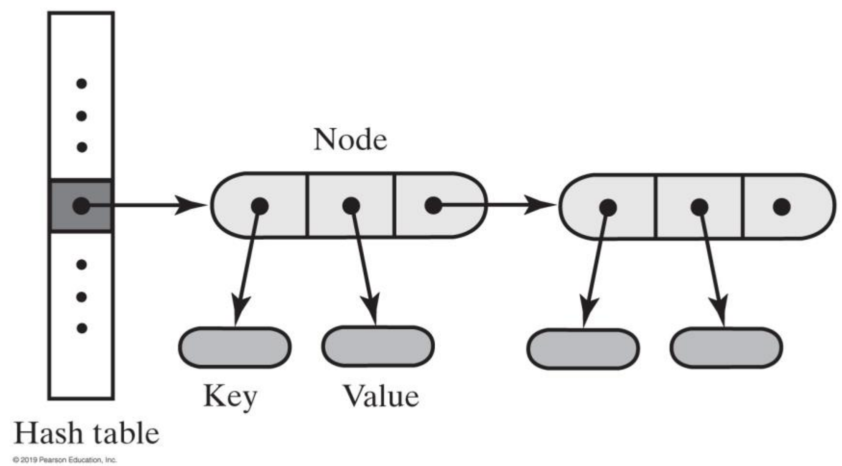


Potential Problem with Open Addressing

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

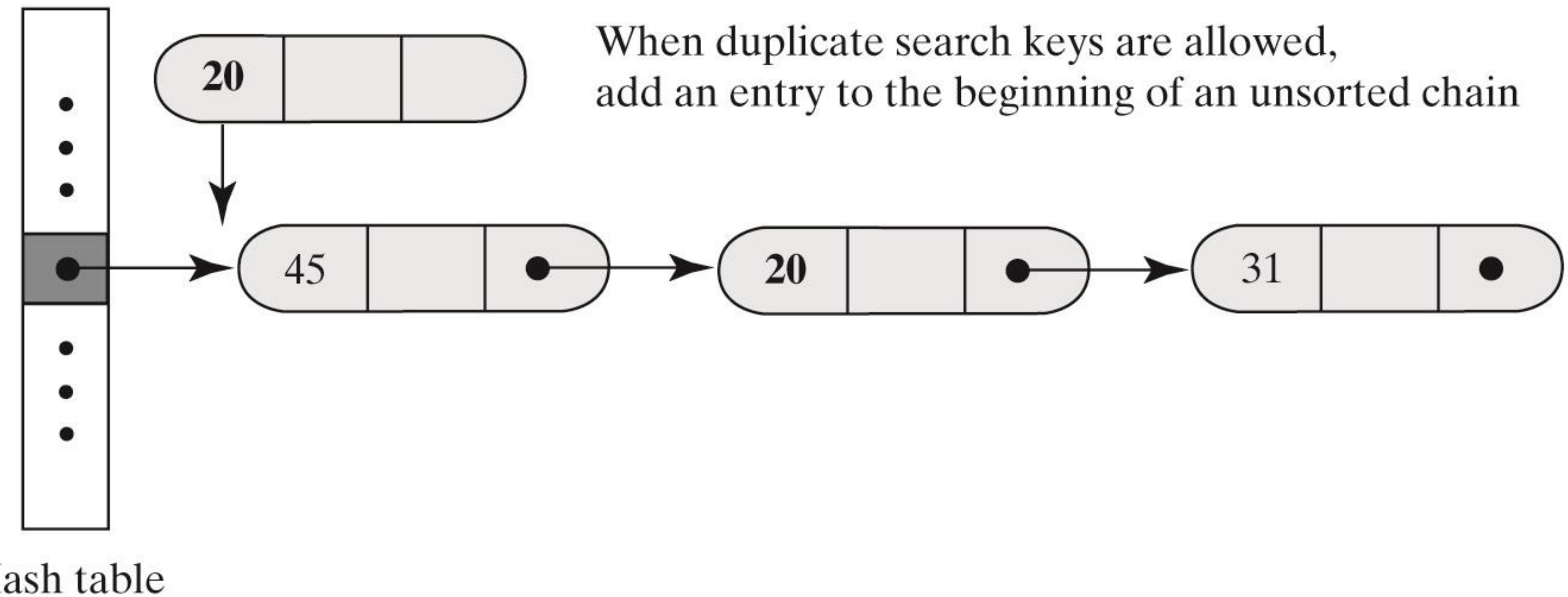


A hash table for use with separate chaining; each bucket is a chain of linked nodes

Separate Chaining

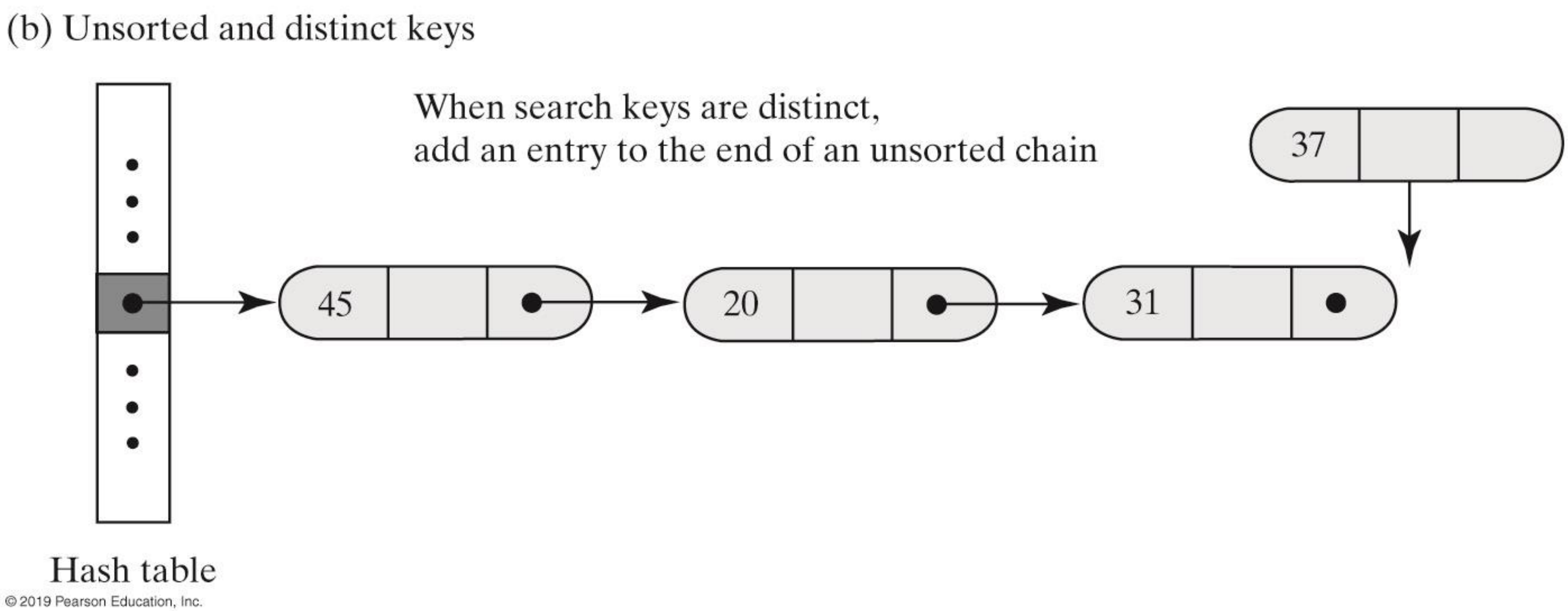
- Inserting a new entry into a linked bucket according to the nature of the integer search keys

(a) Unsorted, and possibly duplicate, keys



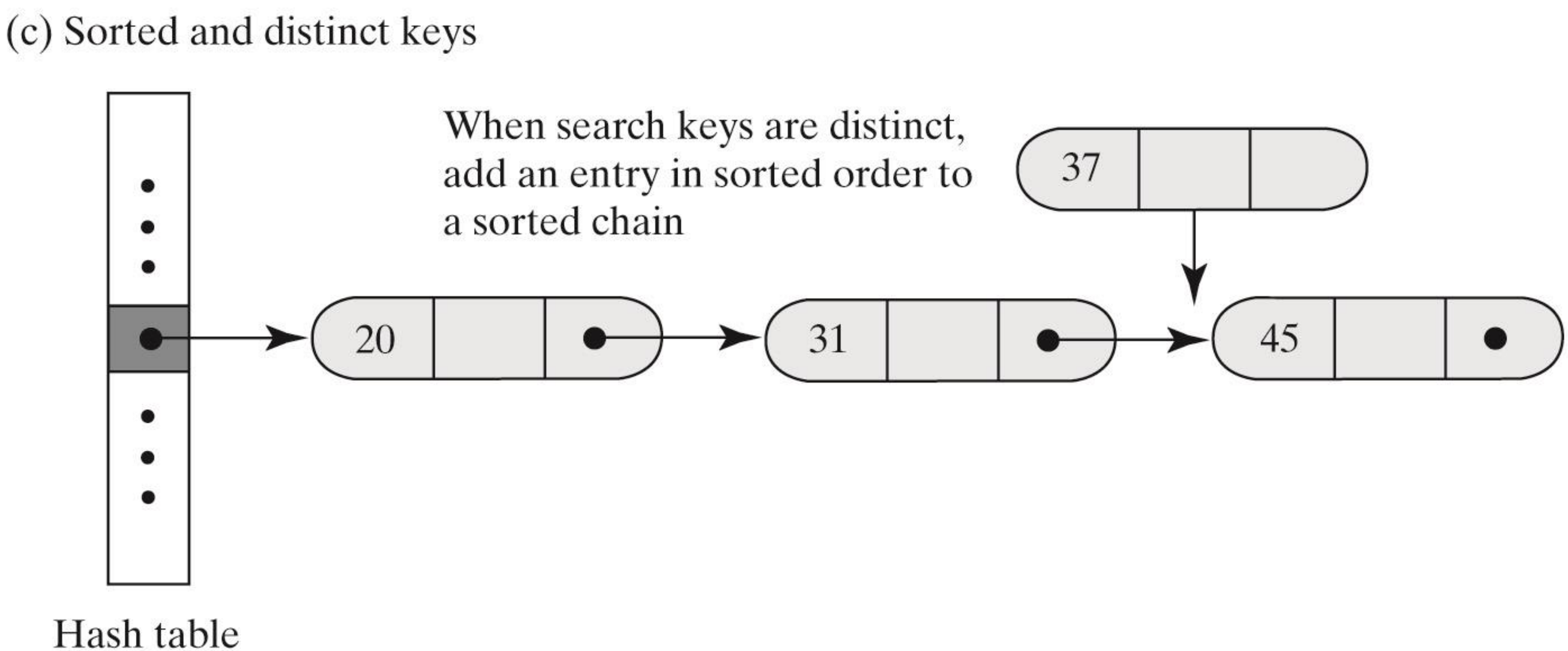
Separate Chaining

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



Separate Chaining

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



Separate Chaining - add()

```
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 – remove()

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 – getValue()

***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
- Example load factor .5
  - unsuccessful: 2.5
  - successful: 1.5

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

- 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

- 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

- 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

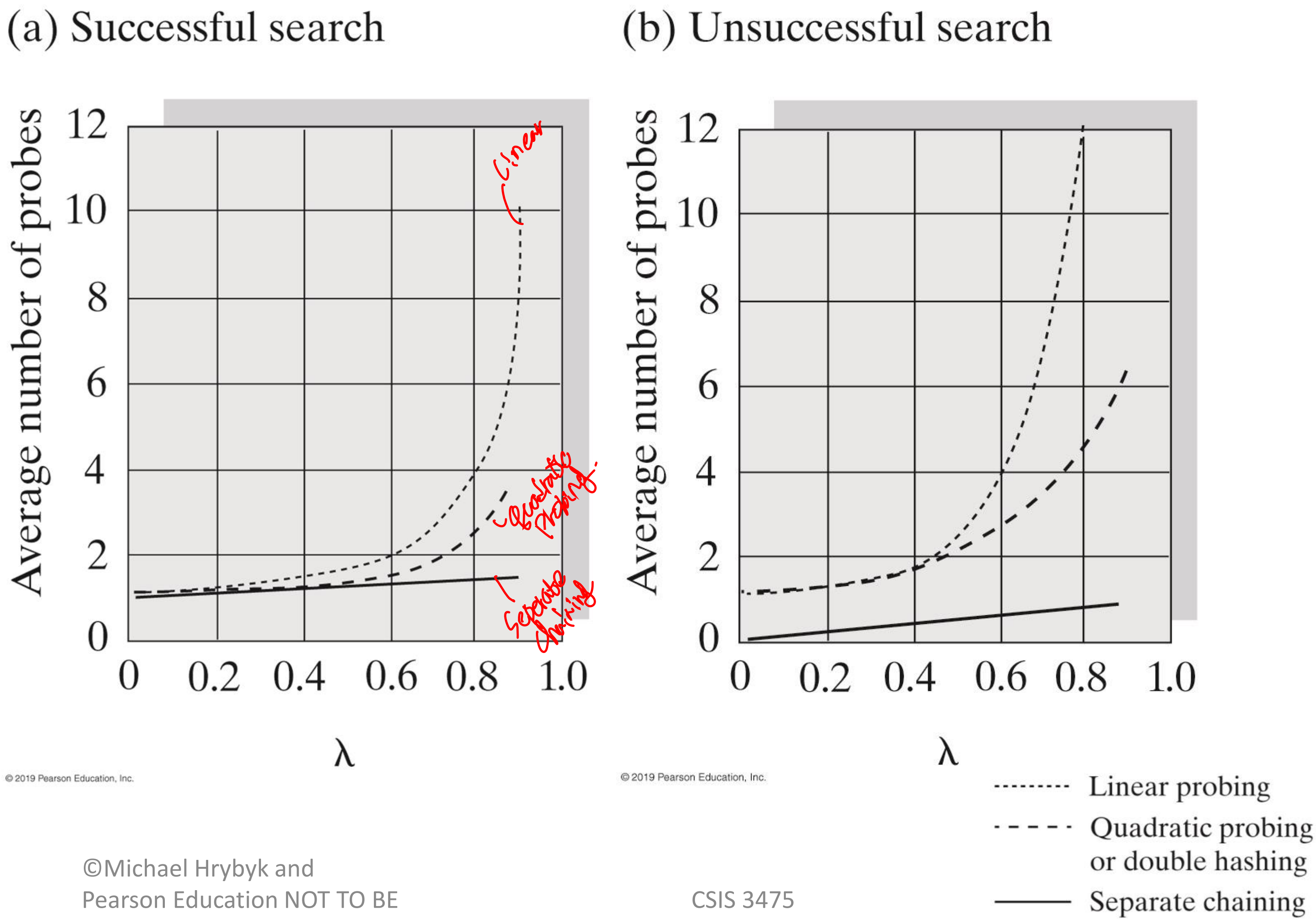
- To maintain efficiency, restrict the size of λ as follows:
    - $\lambda < 0.5$ for open addressing
    - $\lambda < 1.0$ for separate chaining
- Should the load factor exceed these bounds
    - Increase the size of the hash table

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
    - rehashing

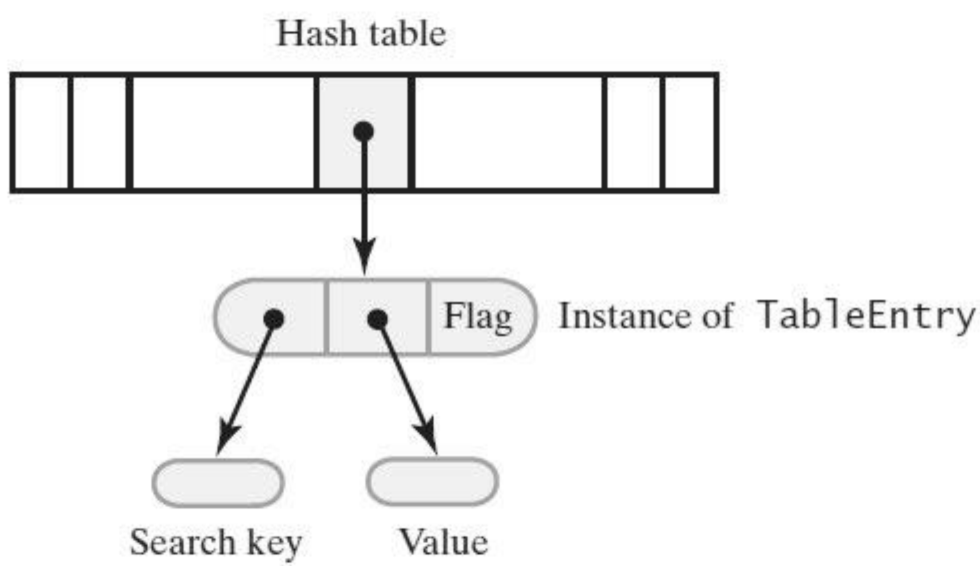
Comparing Schemes for Collision Resolution

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



Dictionary Implementation That Uses Hashing

- 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
- Note use of Comparable key, and Entry class from DictionaryPackage
- Assume keys are unique but unsorted

```java
public class CompletedHashedDictionary<K extends Comparable<? super 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 = 1000000;

             // The hash table:
             private Entry<K, V>[] hashTable;

             // Max size of hash table
             private static final int MAX_SIZE = 2 * MAX_CAPACITY;
             // Fraction of hash table that can be filled
             private static final double MAX_LOAD_FACTOR = 0.5;

             // Occupies locations in the hash table in the


             // available state (locations whose entries were removed
             private final Entry<K, V> AVAILABLE = new Entry<>(null, null);
```

Use a key/value Entry class

- Simple generic class to keep keys and values
- Implements Comparable compareTo() and equals() for use with Iterators
    - Only compares keys, NOT values
- Reused from DictionaryPackage

```java
public class Entry<K extends Comparable<? super K>, V> implements Comparable<Entry<K,V>> {
		private K key;
		private V value;

		public Entry(K searchKey, V dataValue) {
				key = searchKey;
				value = dataValue;
		}

		public K getKey() {
				return key;
		}

		public V getValue() {
				return value;
		}

		public void setValue(V dataValue) {
				value = dataValue;
		}
```

Entry class compareTo()

- Only compares keys, not values

```
    public int compareTo(Entry<K, V> obj) {
        // if only looking for the key, a new entry
        //  with a value of null will have to be used

        // if this is the same object, then we are equal
        if (this == obj)
            return 0;

        // if the object we are comparing is null,
        // then we are higher
        if(obj == null)
            return 1;

        Entry<K, V> other = (Entry<K, V>) obj;

        // if we are null, we are lower
        // if both are null, then return equals

        if (key == null) {
            // null is always lower
            if (other.key != null)
                return -1;
            else return 0;
        }

        // this is a repeat of above for safety

        if(other.key == null)
            return 1;

        // done accounting for nulls, simply return compareTo()

        return key.compareTo(other.key);
    }
```

Entry class equals() and toString()

```java
public boolean equals(Object obj) {

    // if only looking for the key, a new entry
    //  with a value of null will have to be used

    // this code is autogenerated by eclipse
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;

    // simply compare the keys for equality
    //  we don't really care about the value
    //  as keys must be unique
    @SuppressWarnings("unchecked")
    Entry<K, V> other = (Entry<K, V>) obj;
    if (key == null) {
        if (other.key != null)
            return false;
    } else if (!key.equals(other.key))
        return false;

    return true;
}

@Override
public String toString() {
    return "[Key: " + key + ", Value: " + value + "]";
}
```

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 CompletedHashedDictionary() {
     this(DEFAULT_CAPACITY); // Call next constructor
}

public CompletedHashedDictionary(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;

}
```

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);
//      hashIndex = quadraticProbe(hashIndex, key);

        return hashIndex;
    }
    /**
     * 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 % hashTable.length;
        if (index < 0)
            index = index + hashTable.length;

        return index;
    }
```

Important piece.

Linear Probe implementation

```
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
}
```

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
```

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

```java
public V add(K key, V value) {
    if ((key == null) || (value == null))
        return null;

    V oldValue; // Value to return

    // get the next available hash index for the key
    int index = getHashIndex(key);

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

    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);
    }

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

    return oldValue;

}
```

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.

```java
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) // hashTable[index] is not null and does not equal AVAILABLE
{
        removedValue = hashTable[index].getValue()
        hashTable[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;
}
```

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;

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


        // 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

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 separate chaining
  - When limit exceeded, size of table increased by rehashing
- Possible to avoid rehashing by setting number of buckets initially larger
- HashMapDictionary class uses an internal HashMap
  - implements all DictionaryInterface methods by calling HashMap methods.

Java Class Library: The Class `HashSet`

- Implements the interface **`java.util.Set`**
  - **`Set`** uses unique keys
- **`HashSet`** uses an instance of the class **`HashMap`**
  - HashMap uses key/value pairs, whereas HashSet uses a single entry.

Separate Chaining implementation

- See HashedChainedDictionary class
- Requires unique keys.
- Uses an array of linked lists.
  - The objects in the array must implement ListWithIteratorInterface
  - LLinkedWithIterator objects, but could have used any object
    - AListWithIterator can be easily substituted
  - Must import ListWithIteratorsPackage
- Each list consists of Entry objects with a key and value.

HashedChainedDictionary

```
package HashingPackage;

import java.util.Iterator;
import java.util.NoSuchElementException;
import DictionaryPackage.*;
import ListWithIteratorsPackage.*;

/**
 * Implements a hash table ADT by using an array of lists;
 *
 * The lists must implement ListWithIteratorInterface.
 *
 * Items in each list consist of Entry objects.
 *
 * In general, this is a cleaner implementation than that in the textbook, and
 * uses both the dictionary and list ADT implementations created previously.
 *
 * LListWithIterator is used here, but AListWithIterator could also be used.
 */
public class CompletedHashedChainedDictionary<K extends Comparable<? super K>, V>
              implements DictionaryInterface<K, V> {

              private static final int DEFAULT_CAPACITY = 3;
              private static final int MAX_CAPACITY = 1000000;

              private int numberOfEntries;

              // keep a count of hash table buckets used.
              // the count is used to increase the table size if necessary

              private int occupiedBuckets;

              // The hash table is an array of lists.
              // The list must implement Iterator.

              private ListWithIteratorInterface<Entry<K, V>>[] hashTable;

              // Max size of hash table
              private static final int MAX_SIZE = 2 * MAX_CAPACITY;

              // Fraction of hash table that can be filled
              private static final double MAX_LOAD_FACTOR = .9;
```

HashedChainedDictionary constructor

- Creates the hash table

```
public CompletedHashedChainedDictionary() {
    this(DEFAULT_CAPACITY); // Call next constructor
}

public CompletedHashedChainedDictionary(int initialCapacity) {

    // create a new hash table

    createHashTable(initialCapacity);
}

/**
 * Throws an exception if the hash table becomes too large.
 *
 * @param size
 */
private void checkSize(int size) {
    if (size > MAX_SIZE)
        throw new IllegalStateException("Dictionary has become too large.");
}
```

HashedChainedDictionary createHashTable()

- Also used to resize hash table
- Creates new array, then addes old entries to the new array using add() method.

```
private void createHashTable(int newSize) {

    // save the old table location

    ListWithIteratorInterface<Entry<K, V>>[] oldTable = hashTable;

    int oldSize = 0;

    // if the hash table already exists, save its size

    if (hashTable != null)
        oldSize = hashTable.length;

    // Initial size of hash table if it is prime;
    // otherwise increase it until it is prime size
    newSize = getNextPrime(newSize);

    checkSize(newSize); // Check that the prime size is not too large

    // create the new table

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

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

    hashTable = tempTable;

    // reset the counters
    occupiedBuckets = 0;
    numberOfEntries = 0;

    // Copy the old table to the new one.
    // Rehash dictionary entries from old array to the new and bigger array;
    // skip both null locations and buckets with no entries
    // note use of add() to do this which rehashes keys

    for (int index = 0; index < oldSize; index++) {
        if (oldTable[index] != null) {
            for (Entry<K, V> entry : oldTable[index])
                add(entry.getKey(), entry.getValue());
        }
    }
}
```

add() algorithm

- hash the key, and see if a bucket in the table exists for it.
    - if not, create the list and add the key/value entry to it
- if the bucket exists, find the key in the existing list
    - if it does not exist, add the key/value entry to the existing list
    - if it does exist, simply replace the value associated with the key

HashedChainedDictionary add()

```java
public V add(K key, V value) {
   if ((key == null) || (value == null))
      return null;

   V oldValue; // Value to return

   // dummy Entry for key, where value is null
   // used for findEntry()

   Entry<K, V> newEntry = new Entry<>(key, value);

   // hash the key
   int index = getHashIndex(key);

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

   // save the list - this is just shorthand, makes
things easier to read

   ListWithIteratorInterface<Entry<K, V>> list =
hashTable[index];

   if (list == null) {
      // List doesn't exist, so create new one, and add
entry
      list = new CompletedLListWithIterator<>();
      hashTable[index] = list;
      numberOfEntries++;
      occupiedBuckets++;
      list.add(newEntry);
      oldValue = null;

   } else {
      // a list exists in the bucket, so use it

      // if the entry is not in the list, add it
      // otherwise just reset the value

      Entry<K, V> existingEntry = getEntry(list, key);
```

Not in to the list -> add it.

```java
      if (existingEntry == null) {
         numberOfEntries++;
         list.add(newEntry);
         oldValue = null;
      } else {
         oldValue = existingEntry.getValue();
         existingEntry.setValue(value);
      }
   }

   // Ensure that hash table is large enough for the
   // next time we add
   // If full, double the size

   if (isHashTableTooFull()) {
      createHashTable(2 * hashTable.length);
   }

   return oldValue;
}
```

HashedChainedDictionary getEntry()

- Look for an entry in a list. The list will be a bucket in the hash table array.
- Use iterator to traverse the list

```
private Entry<K, V> getEntry(ListWithIteratorInterface<Entry<K, V>> list, K key) {
    if (list == null || key == null)
        return null;

    for (Entry<K, V> item : list) {
        if (key.equals(item.getKey()))
            return item;
    }
    // not found
    return null;

}
```

remove() algorithm

- hash the key, and see if a hash table bucket exists for the index
- if so, find the key in the list associated with the bucket
- if found, remove it and adjust counters
- Implementation
  - Use List methods findEntry() and remove().

HashedChainedDictionary remove()

```
public V remove(K key) {
    if (key == null)
        return null;

    V oldValue = null;

    // get the hash index for the key
    int index = getHashIndex(key);

    // is the bucket allocated?

    if (hashTable[index] != null) {

        // look for the key in the existing list
        // if found, remove it

        // this is a dummy entry with only the key
        Entry<K, V> entry = new Entry<>(key, null);

        // find the key
        int position = hashTable[index].findEntry(entry);

        if (position >= 0) {
            // found it, now remove it
            entry = hashTable[index].remove(position);
            oldValue = entry.getValue();
            numberOfEntries--;

            // if the list is empty, decrease the count of
            // available buckets
            if (hashTable[index].isEmpty())
                occupiedBuckets--;
        }
    }
    return oldValue;
}
```

HashedChainedDictionary getValue()

- Get an Entry object matching the key, and return thevalue

```
public V getValue(K key) {
    if (key == null)
        return null;

    // hash the key and get its entry from the associated list

    int index = getHashIndex(key);

    Entry<K, V> entry = getEntry(hashTable[index], key);

    // if it exists, get the value
    if (entry != null) {
        return entry.getValue();
    } else
        return null;
}
```

HashedChainedDictionary hashing

- No linear or quadratic probing
- getHashIndex(), setHashIndex(), isHashTableTooFull() , getNextPrime() and isPrime() are the same as in HashedDictionary
- Use the object’s hashCode() method to generate the hash, and then compress to table size using % operator

HashedChainedDictionary Iterators

- Same Iterators as HashedDictionary
- Additional StatsIterator to provide information about the hash table
- Implementation design
  - Need to iterate through all Entry objects in the hash table.
  - next() only needs to return either a key or value
  - Use common EntryIterator which KeyIterator and ValueIterator will use

```
public Iterator<K> getKeyIterator() {
    return new KeyIterator();
}

public Iterator<V> getValueIterator() {
    return new ValueIterator();
}

public Iterator<String> getStatsIterator() {
    return new StatsIterator();
}
```

Common EntryIterator

```
/**
 * Common iterator returning all the Entry objects in a hash table.
 *
 * Entry objects have a key and value, so iterating over all Entry objects
 * will also give all keys and values
 *
 * @author mhrybyk
 *
 */
private class EntryIterator implements Iterator<Entry<K, V>> {
   private int currentIndex; // Current position in hash table

   // this is the iterator for a hash table's bucket which contains a list
   // of entries

   private Iterator<Entry<K, V>> listIterator;

   private EntryIterator() {
      currentIndex = 0;
      listIterator = null;
   }
```

EntryIterator hasNext()

- Need to iterate over the array **and** each list in each array bucket.

```
public boolean hasNext() {

    // get the next bucket that has a non-empty list

    while (currentIndex < hashTable.length) {
        // see if the list at the bucket has entries
        if (hashTable[currentIndex] != null && hashTable[currentIndex].size() > 0) {
            break;
        }
        currentIndex++;
    }

    // if none found and we are at the end, the index will be the table size

    if (currentIndex < hashTable.length) {
        // if the list exists but there is no listIterator yet, return true
        // next() will allocate the listIterator.
        // otherwise look to see if the listIterator has another item in the list
        if (listIterator == null)
            return true;
        else
            return listIterator.hasNext();
    }

    // if the index is the size of the table, we are at the end

    return false;

}
```

EntryIterator next()

```
public Entry<K, V> next() {
    Entry<K, V> result = null;

    if (hasNext()) {

        // if this is a new bucket with a list, the listIterator will be null
        // so get its listIterator
        if (listIterator == null) {
            listIterator = hashTable[currentIndex].getIterator();
        }

        // get the next entry in the list.
        if (listIterator.hasNext()) {
            result = listIterator.next();

            // are we at the end of the list? If so, bump the hashtable index
            // and reset the listIterator
            if (!listIterator.hasNext()) {
                currentIndex++;
                listIterator = null;
            }

        }
    } else
        throw new NoSuchElementException();

    return result;
}
```

KeyIterator

- Using an EntryIterator, just get the next Entry if it exists.
- Then extract the key from the Entry object.
- No duplicated code (unlike the textbook).

```
private class KeyIterator implements Iterator<K> {

    private EntryIterator iterator;

    private KeyIterator() {
        iterator = new EntryIterator();
    }

    public boolean hasNext() {
        return iterator.hasNext();
    }

    public K next() {
        K result = null;

        if (hasNext()) {
            result = iterator.next().getKey();
        }
        return result;
    }
}
```

ValueIterator

```java
private class ValueIterator implements Iterator<V> {

      private EntryIterator iterator;

      private ValueIterator() {
          iterator = new EntryIterator();
      }

      public boolean hasNext() {
          return iterator.hasNext();
      }

      public V next() {
          V result = null;

          if (hasNext()) {
              result = iterator.next().getValue();
          }
          return result;
      }

   }
```

Testing - HashedDictionaryTestDriver

- testDictionary() is similar to that in DictionaryDriverPackage
  - tests basic dictionary methods
- testHashTable() tests hash table functions, and displays stats
- Uncomment the class that needs to be test
- HashedChainedDictionary, HashedDictionary, or HashedMapDictionary

```
    public static void main(String[] args) {
        // uncomment out Completed to test implementation

        DictionaryInterface<String, String> telephoneDirectory = new CompletedHashedChainedDictionary<>();
//      DictionaryInterface<String, String> telephoneDirectory = new CompletedHashMapDictionary<>();
//      DictionaryInterface<String, String> telephoneDirectory = new CompletedHashedDictionary<>();
        testDictionary(telephoneDirectory);

        // can't use the plain Directory interface for hashed chain, as it needs a unique
        // display method to show hash table stats

        // edit the method declaration as well to change

        // For chained, output is for load factor of .5 and initial size of 2

        CompletedHashedChainedDictionary<Name, String> nameList = new CompletedHashedChainedDictionary<>();
//      DictionaryInterface<Name, String> nameList = new CompletedHashMapDictionary<>();
//      DictionaryInterface<Name, String> nameList = new CompletedHashedDictionary<>();
        testHashTable(nameList);
        System.out.println("\n\nDone.");
    }

    public static void testDictionary(DictionaryInterface<String, String> telephoneDirectory) {

    }

    public static void testHashTable(CompletedHashedChainedDictionary<Name, String> nameList) {

    }
```