CS 570: Data Structures

Sets and Maps

Instructor: Iraklis Tsekourakis

Email: itsekour@stevens.edu

CHAPTER 7 (PART 1)

Chapter Objectives

- To understand the Java Map and Set interfaces and how to use them
- To learn about hash coding and its use to facilitate efficient search and retrieval
- To study two forms of hash tables—open addressing and chaining—and to understand their relative benefits and performance trade-offs

Chapter Objectives (cont.)

- To learn how to implement both hash table forms
- To be introduced to the implementation of `Maps` and `Sets`
- To see how two earlier applications can be implemented more easily using `Map` objects for data storage

Week 12

- Reading Assignment: Koffman and Wolfgang, Sections 7.1-7.3

Introduction

- We learned about part of the Java Collection Framework in Chapter 2 (`ArrayList` and `LinkedList`)
- The classes that implement the `List` interface are all *indexed* collections
  - An index or subscript is associated with each element
  - The element's index often reflects the relative order of its insertion in the list
  - Searching for a particular value in a list is generally O(n)
  - An exception is a binary search of a sorted object, which is O(log n)

Introduction (cont.)

- In this chapter, we consider another part of the `Collection` hierarchy: the `Set` interface and the classes that implement it
- `Set` objects
  - are not indexed
  - do not reveal the order of insertion of items
  - enable efficient search and retrieval of information
  - allow removal of elements without moving other elements around

Introduction (cont.)

- Relative to a `Set`, `Map` objects provide efficient search and retrieval of entries that contain pairs of objects (a unique key and the information)
- Hash tables (implemented by a `Map` or `Set`) store objects at arbitrary locations and offer an average constant time for insertion, removal, and searching

Sets and the `Set` Interface

Section 7.1

The `Set` Abstraction

- A set is a collection that contains no duplicate elements and at most one `null` element
  - adding `"apples"` to the set `{"apples", "oranges", "pineapples"}` results in the same set (no change)
- Operations on sets include:
  - testing for membership
  - adding elements
  - removing elements
  - union $A \cup B$
  - intersection $A \cap B$
  - difference $A - B$
  - subset $A \subset B$

The Set Abstraction(cont.)

- The union of two sets A, B is a set whose elements belong either to A or B or to both A and B.

  Example: {1, 3, 5, 7} ∪ {2, 3, 4, 5} is {1, 2, 3, 4, 5, 7}
- The intersection of sets A, B is the set whose elements belong to both A and B.

  Example: {1, 3, 5, 7} ∩ {2, 3, 4, 5} is {3, 5}
- The difference of sets A, B is the set whose elements belong to A but not to B.

  Examples: {1, 3, 5, 7} – {2, 3, 4, 5} is {1, 7}; {2, 3, 4, 5} – {1, 3, 5, 7} is {2, 4}
- Set A is a subset of set B if every element of set A is also an element of set B.

  Example: {1, 3, 5, 7} ⊂ {1, 2, 3, 4, 5, 7} is true

The `Set` Interface and Methods

- Required methods: testing set membership, testing for an empty set, determining set size, and creating an iterator over the set
- Optional methods: adding an element and removing an element
- Constructors to enforce the “no duplicate members” criterion
  - The `add` method does not allow duplicate items to be inserted

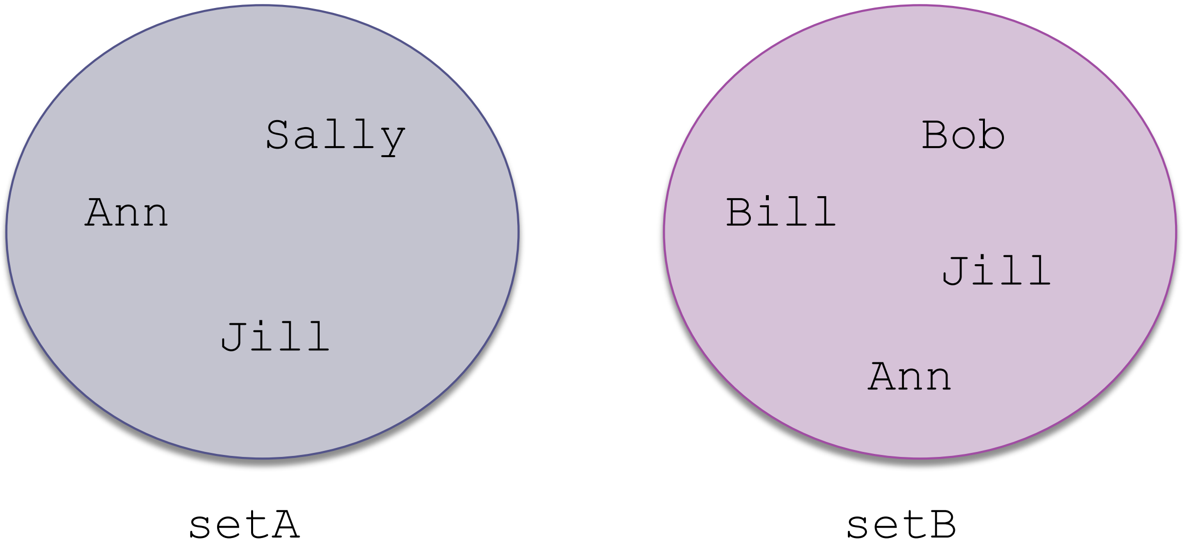
The `Set` Interface and Methods(cont.)

- Required method: `containsAll` tests the subset relationship
- Optional methods: `addAll`, `retainAll`, and `removeAll` perform union, intersection, and difference, respectively

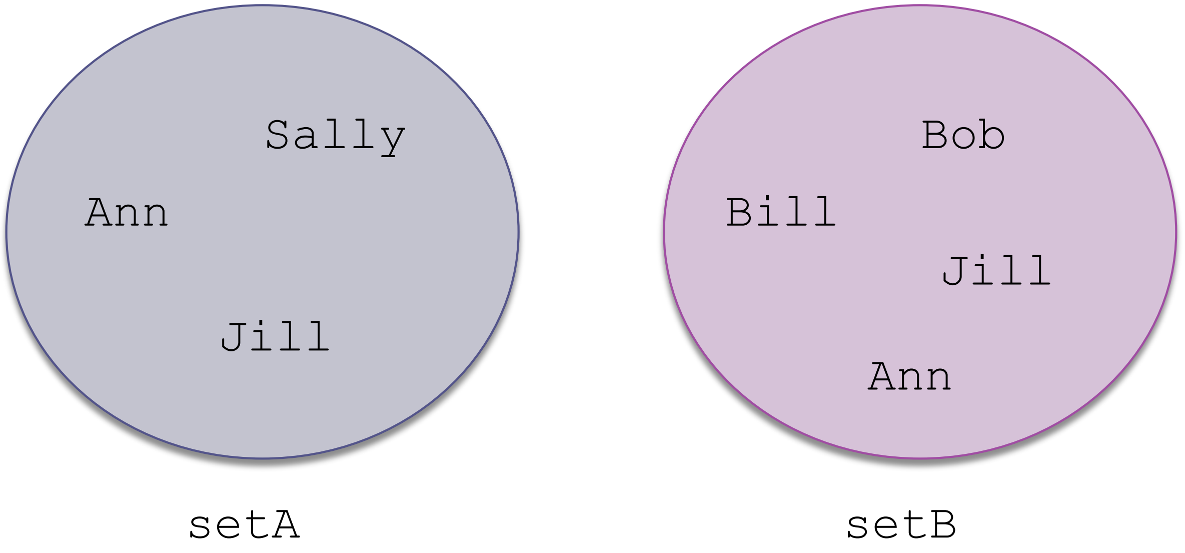
The Set Interface and Methods(cont.)

Method	Behavior
boolean add(E obj)	Adds item obj to this set if it is not already present (optional operation) and returns **true**. Returns false if obj is already in the set.
boolean addAll(Collection<E> coll)	Adds all of the elements in collection coll to this set if they're not already present (optional operation). Returns **true** if the set is changed. Implements *set union* if coll is a Set.
boolean contains(Object obj)	Returns **true** if this set contains an element that is equal to obj. Implements a test for *set membership*.
boolean containsAll(Collection<E> coll)	Returns **true** if this set contains all of the elements of collection coll. If coll is a set, returns **true** if this set is a subset of coll.
boolean isEmpty()	Returns **true** if this set contains no elements.
Iterator<E> iterator()	Returns an iterator over the elements in this set.
boolean remove(Object obj)	Removes the set element equal to obj if it is present (optional operation). Returns **true** if the object was removed.
boolean removeAll(Collection<E> coll)	Removes from this set all of its elements that are contained in collection coll (optional operation). Returns **true** if this set is changed. If coll is a set, performs the *set difference* operation.
boolean retainAll(Collection<E> coll)	Retains only the elements in this set that are contained in collection coll (optional operation). Returns **true** if this set is changed. If coll is a set, performs the *set intersection* operation.
int size()	Returns the number of elements in this set (its cardinality).

The `Set` Interface and Methods(cont.)

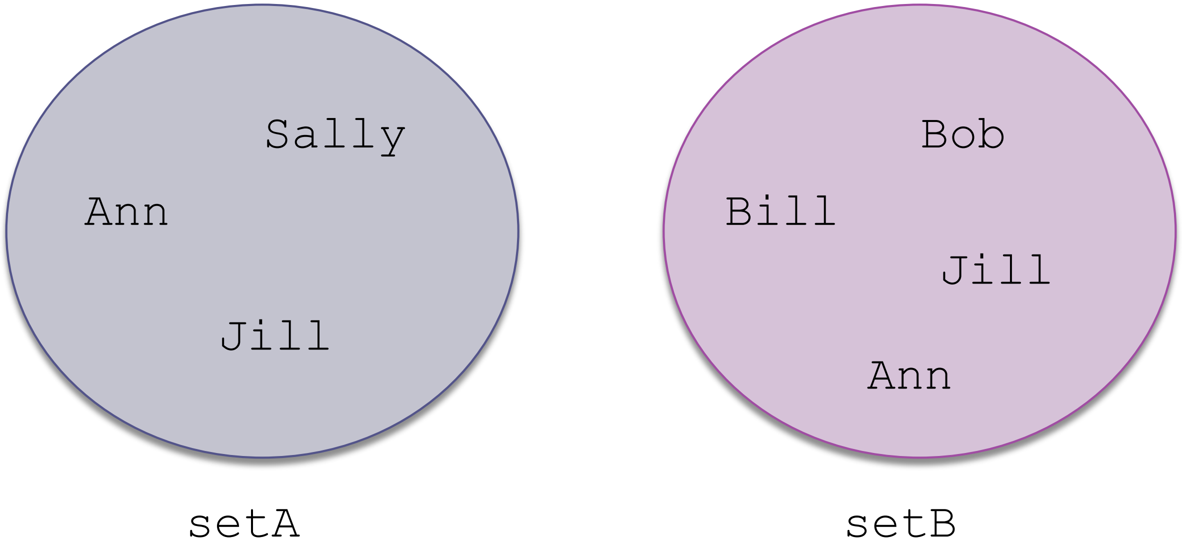


The `Set` Interface and Methods(cont.)



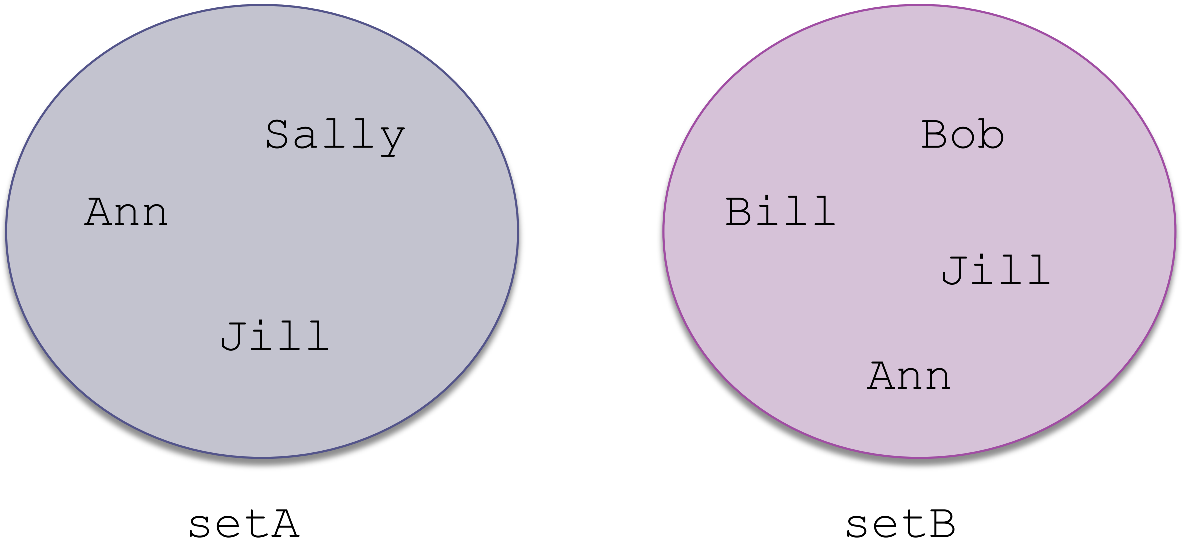
```
setA.addAll(setB);
```

The `Set` Interface and Methods(cont.)



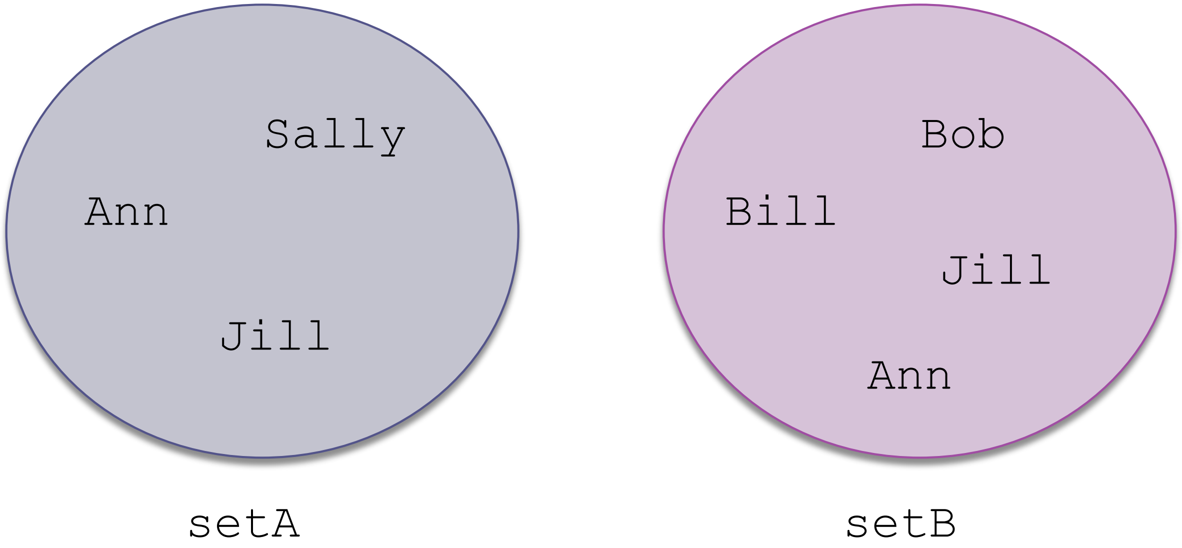
```
setA.addAll(setB);

System.out.println(setA);

Outputs:
[Bill, Jill, Ann, Sally, Bob]
```

The `Set` Interface and Methods(cont.)



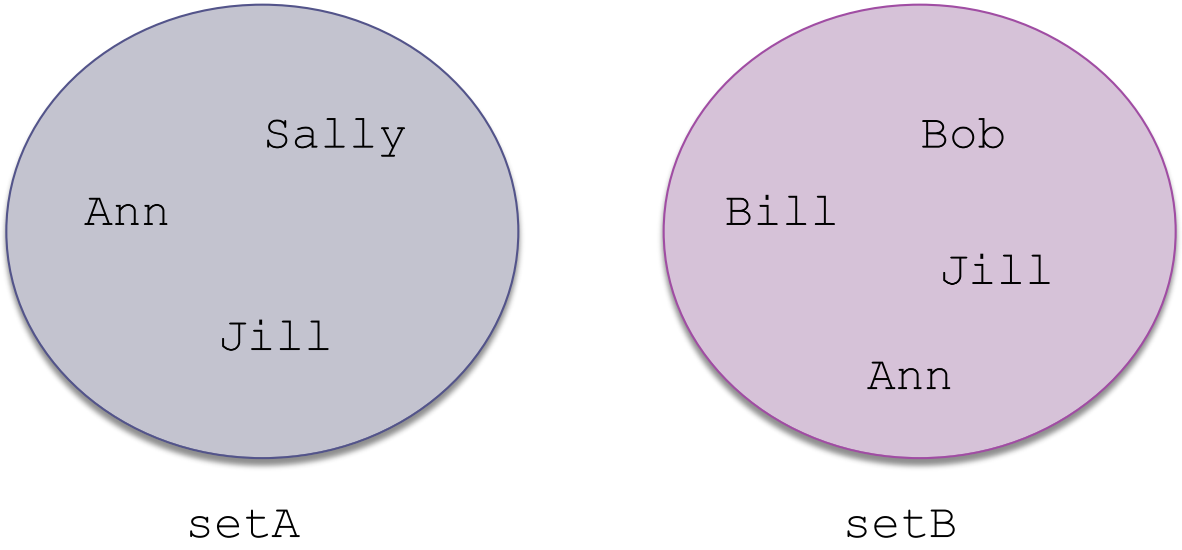
If a copy of original `setA` is in `setACopy`, then . . .

The `Set` Interface and Methods(cont.)



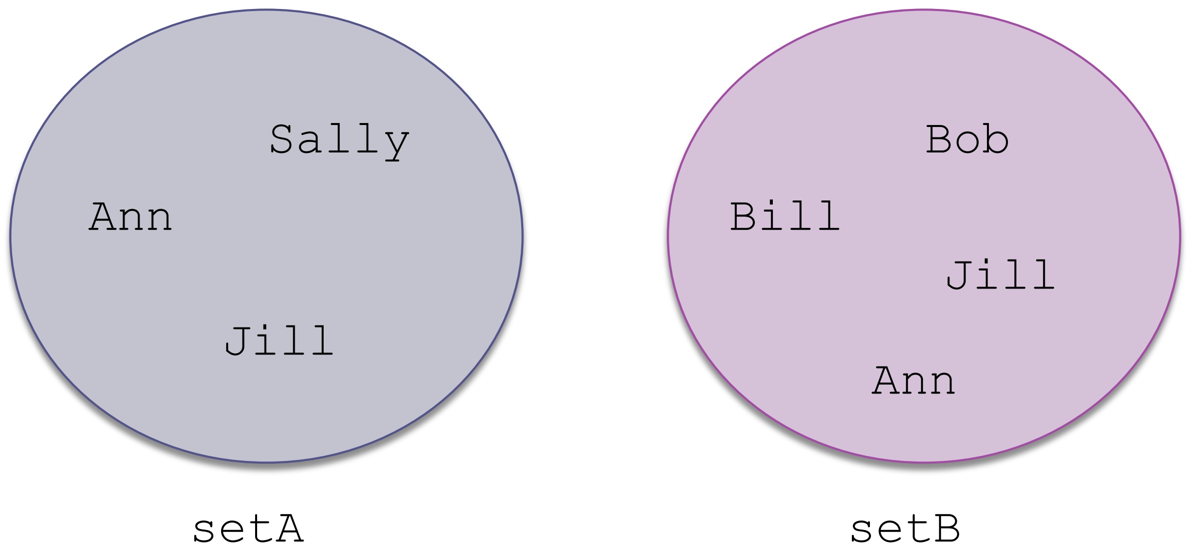
```
setACopy.retainAll(setB);
```

The `Set` Interface and Methods(cont.)



```
setACopy.retainAll(setB);

System.out.println(setACopy);

Outputs:
[Jill, Ann]
```

The Set Interface and Methods(cont.)



```
setACopy.removeAll(setB);

System.out.println(setACopy);

Outputs:
[Sally]
```

Comparison of Lists and Sets

- Collections implementing the `Set` interface must contain unique elements
- Unlike the `List.add` method, the `Set.add` method returns `false` if you attempt to insert a duplicate item
- Unlike a `List`, a `Set` does not have a `get` method—elements cannot be accessed by index

Comparison of Lists and Sets (cont.)

- You can iterate through all elements in a `Set` using an `Iterator` object, but the elements will be accessed in arbitrary order

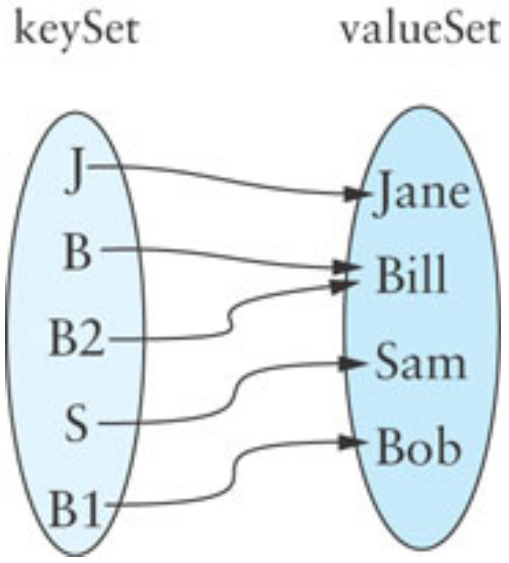
```
for (String nextItem : setA) {
  //Do something with nextItem
  …
}
```

Maps and the `Map` Interface

Section 7.2

Maps and the `Map` Interface

- The `Map` is related to the `Set`
- Mathematically, a `Map` is a set of ordered pairs whose elements are known as the key and the value
- Keys must be unique, but values need not be unique
- You can think of each key as a "mapping" to a particular value
- A map provides efficient storage and retrieval of information in a table
- A map can have *many-to-one* mapping: (`B, Bill`), (`B2, Bill`)



```
{(J, Jane), (B, Bill),
 (S, Sam), (B1, Bob),
 (B2,  Bill)}
```

Maps and the `Map` Interface(cont.)

- In an *onto* mapping, all the elements of valueSet have a corresponding member in keySet
- The `Map` interface should have methods of the form

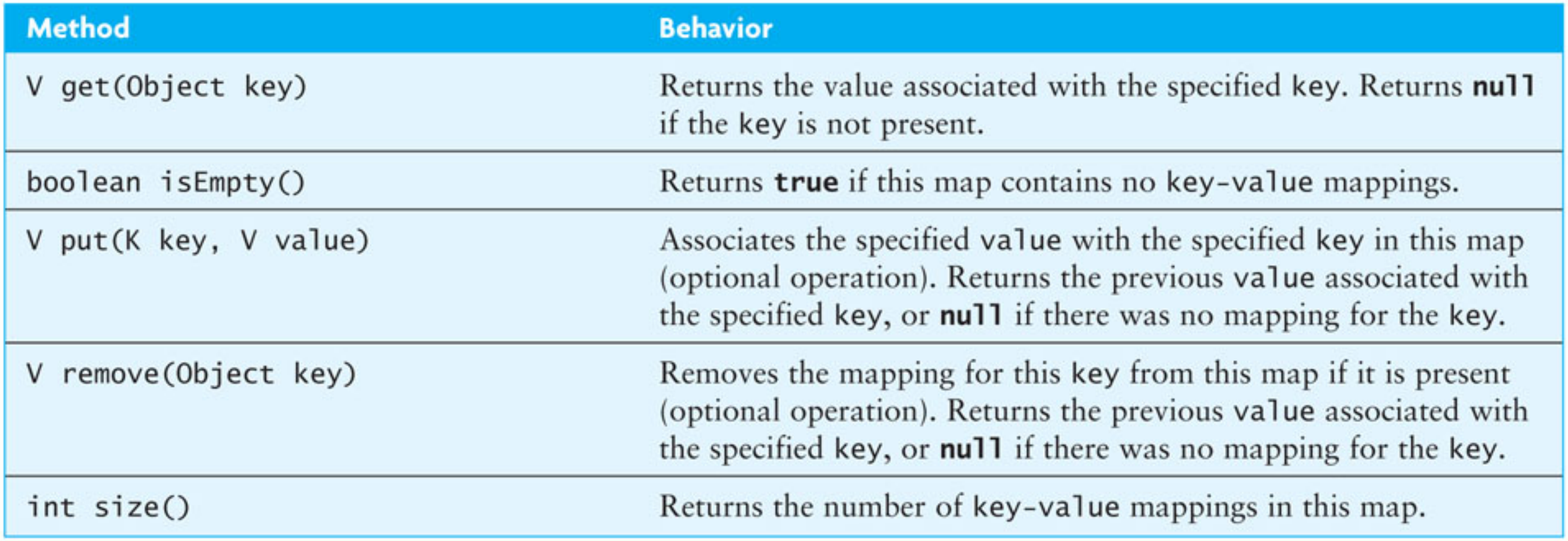
```
V.get (Object key)
V.put (K key, V value)
```

Maps and the `Map` Interface(cont.)

- When information about an item is stored in a table, the information should have a unique ID
- A unique ID may or may not be a number
- This unique ID is equivalent to a key

Type of item	Key	Value
University student	Student ID number	Student name, address, major, grade point average
Online store customer	E-mail address	Customer name, address, credit card information, shopping cart
Inventory item	Part ID	Description, quantity, manufacturer, cost, price

Map Interface



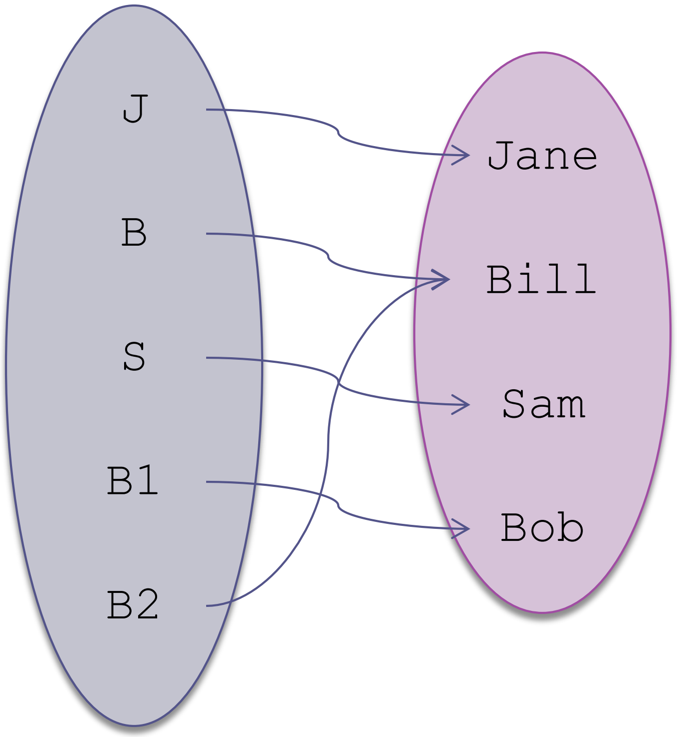
Method	Behavior
`V get(Object key)`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present.
`boolean isEmpty()`	Returns **`true`** if this map contains no `key-value` mappings.
`V put(K key, V value)`	Associates the specified `value` with the specified `key` in this map (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`V remove(Object key)`	Removes the mapping for this `key` from this map if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`int size()`	Returns the number of `key-value` mappings in this map.

Map Interface (cont.)

- The following statements build a Map object:

```
Map<String, String> aMap =
   new HashMap<String,
   String>();
```

```
aMap.put("J", "Jane");
aMap.put("B", "Bill");
aMap.put("S", "Sam");
aMap.put("B1", "Bob");
aMap.put("B2", "Bill");
```

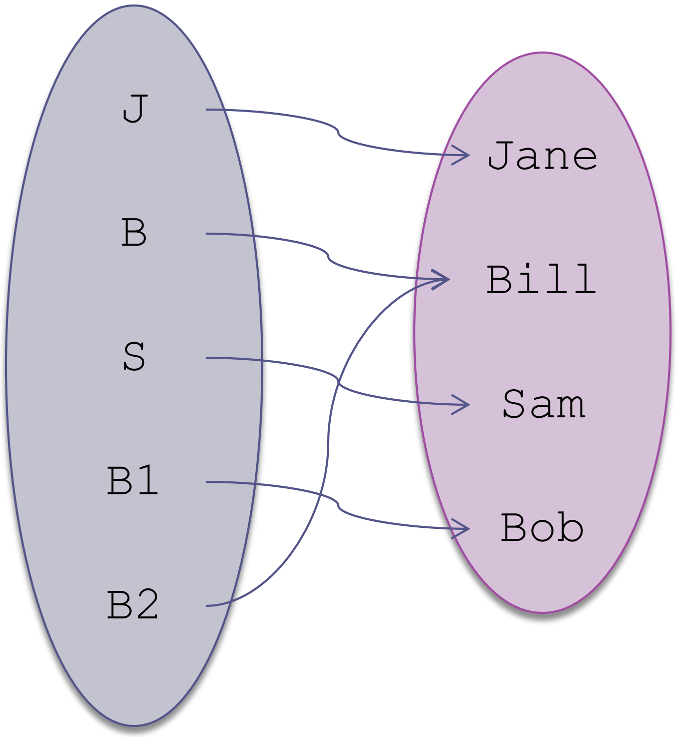


`Map` Interface (cont.)

`aMap.get("B1")`

returns:

`"Bob"`



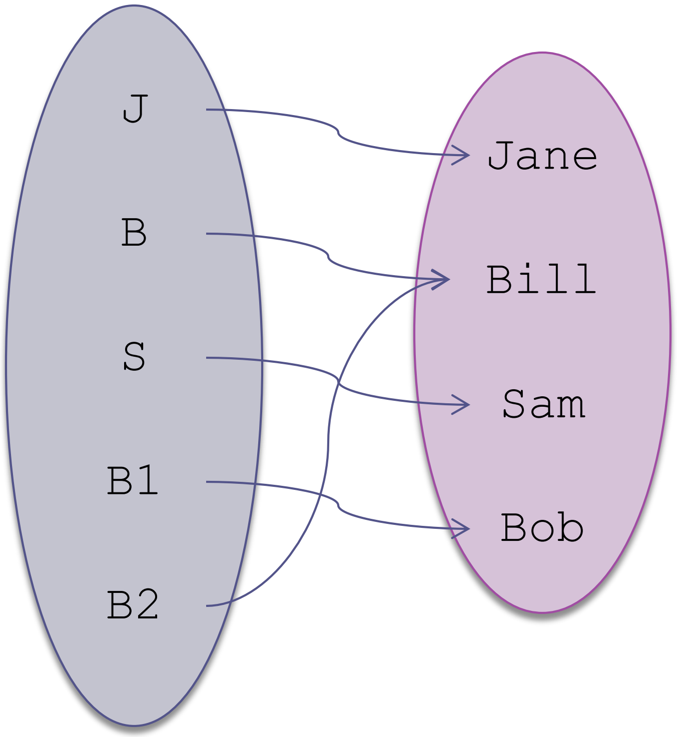
Map Interface (cont.)

`aMap.get("Bill")`

returns:

`null`

(`"Bill"` is a value, not a key)



Hash Tables

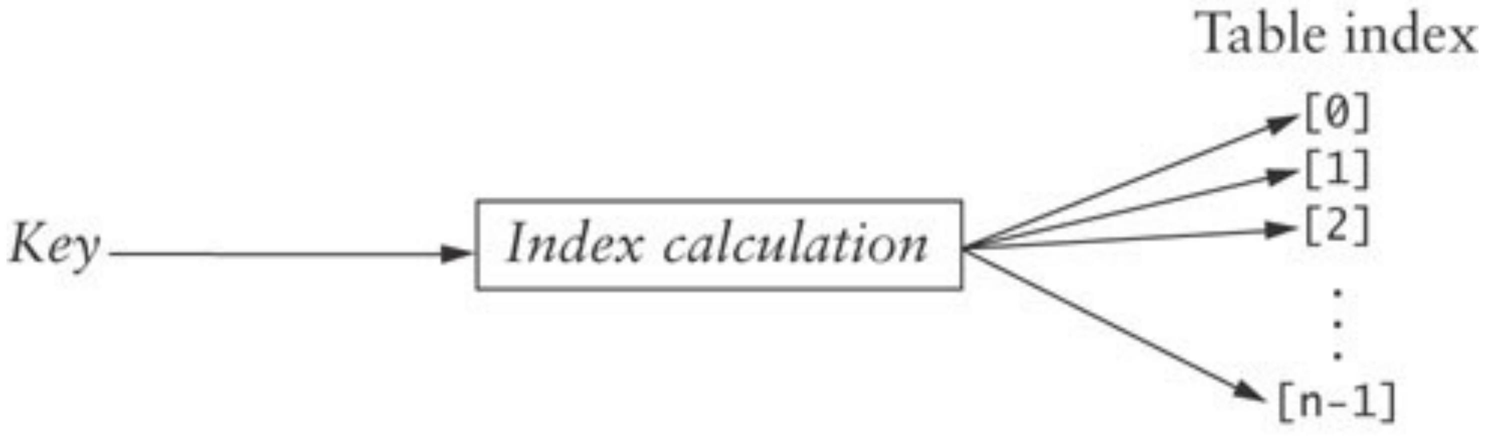
Section 7.3

Hash Tables

- The goal of hash table is to be able to access an entry based on its key value, not its location
- We want to be able to access an entry directly through its key value, rather than by having to determine its location first by searching for the key value in an array
- Using a hash table enables us to retrieve an entry in constant time (on *average*, O(1))

Hash Codes and Index Calculation

- The basis of hashing is to transform the item's key value into an integer value (its *hash code*) which is then transformed into a table index



Hash Codes and Index Calculation (cont.)

- Consider the Huffman code problem from chapter 6
- If a text contains only ASCII values, which are the first 128 Unicode values we could use a table of size 128 and let its Unicode value be its location in the table

```
    int index = asciiChar;
```

. . .	. . .
65	A, 8
66	B, 2
67	C, 3
68	D, 4
69	E, 12
70	F, 2
71	G, 2
72	H, 6
73	I, 7
74	J, 1
75	K, 2
. . .	. . .

Hash Codes and Index Calculation (cont.)

- However, what if all 65,536 Unicode characters were allowed?
- If you assume that on average 100 characters were used, you could use a table of 200 characters and compute the index by:

```
int index = unicode % 200;
```

Hash Codes and Index Calculation (cont.)

- If a text contains this snippet:

```
. . . mañana (tomorrow), I'll finish my program. . .
```

- Given the following Unicode values:

Hexadecimal	Decimal	Name	Character
0x0029	41	right parenthesis	)
0x00F1	241	lower case n with tilde	ñ

- The indices for letters 'ñ' and ')' are both 41

  `41 % 200 = 41` and `241 % 200 = 41`

- This is called a *collision;* we will discuss how to deal with collisions shortly

Methods for Generating Hash Codes

- In most applications, a key will consist of strings of letters or digits (such as a social security number, an email address, or a partial ID) rather than a single character
- The number of possible key values is much larger than the table size
- Generating good hash codes typically is an experimental process
- The goal is a *random (uniform) distribution of values*
- Simple algorithms sometimes generate lots of collisions

Hash Function Examples

- Ideal goal: scramble the keys uniformly to produce a table index
  - Efficiently computable
  - Each table index equally likely for each key
- Phone numbers
  - Bad: first three digits
  - Better: last three digits
- Social Security numbers
  - Bad: first three digits
  - Better: last three digits

Java `HashCode` Method

- For strings, simply summing the `int` values of all characters returns the same hash code for `"sign"` and `"sing"`
- The Java API algorithm accounts for position of the characters as well
- `String.hashCode()` returns the integer calculated by the formula:

  $$s_0 \times 31^{(n-1)} + s_1 \times 31^{(n-2)} + \ldots + s_{n-1}$$

  where s_i is the *i*th character of the string, and *n* is the length of the string
- "Cat" has a hash code of:

  $$\text{'C'} \times 31^2 + \text{'a'} \times 31 + \text{'t'} = 67{,}510$$

- 31 is a prime number, and prime numbers generate relatively few collisions

Java `HashCode` Method (cont.)

- Because there are too many possible strings, the integer value returned by `String.hashCode` cannot be unique
- However, because the `String.hashCode` method distributes the hash code values fairly evenly throughout the range, the probability of two strings having the same hash code is low
- The probability of a collision with

  ```
          s.hashCode() % table.length
  ```

  is proportional to how full the table is

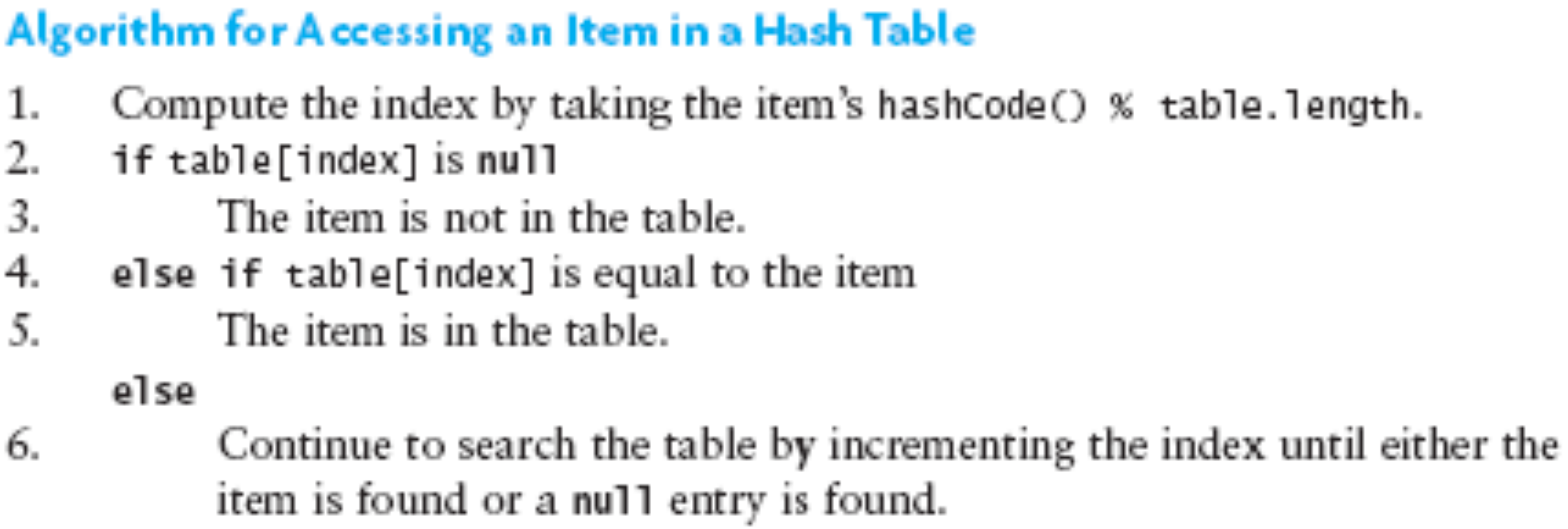
Methods for Generating Hash Codes (cont.)

- A good hash function should be relatively simple and efficient to compute
- It doesn't make sense to use an O(*n*) hash function to avoid doing an O(*n*) search

Open Addressing

- We now consider two ways to organize hash tables:
  - open addressing
  - chaining
- In open addressing, *linear probing* can be used to access an item in a hash table
  - If the index calculated for an item's key is occupied by an item with that key, we have found the item
  - If that element contains an item with a different key, increment the index by one
  - Keep incrementing until you find the key or a `null` entry (assuming the table is not full)

Open Addressing (cont.)



Algorithm for Accessing an Item in a Hash Table

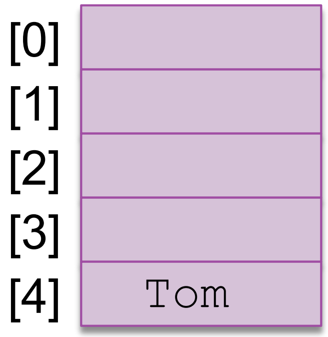
```
1.  Compute the index by taking the item's hashCode() % table.length.
2.  if table[index] is null
3.          The item is not in the table.
4.  else if table[index] is equal to the item
5.          The item is in the table.
    else
6.          Continue to search the table by incrementing the index until either the
            item is found or a null entry is found.
```

Table Wraparound and Search Termination

- As you increment the table index, your table should wrap around as in a circular array
- This enables you to search the part of the table before the hash code value in addition to the part of the table after the hash code value
- But it could lead to an infinite loop
- How do you know when to stop searching if the table is full and you have not found the correct value?
  - Stop when the index value for the next probe is the same as the hash code value for the object
  - Ensure that the table is never full by increasing its size after an insertion when its load factor exceeds a specified threshold

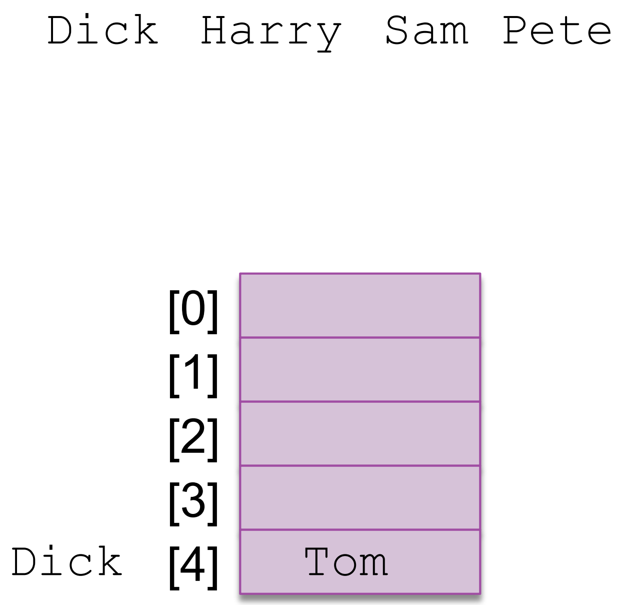
Hash Code Insertion Example

`Tom  Dick  Harry  Sam  Pete`



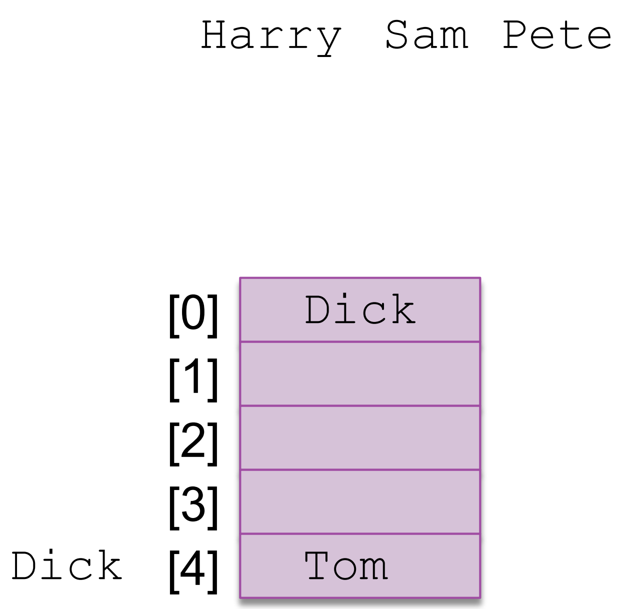
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



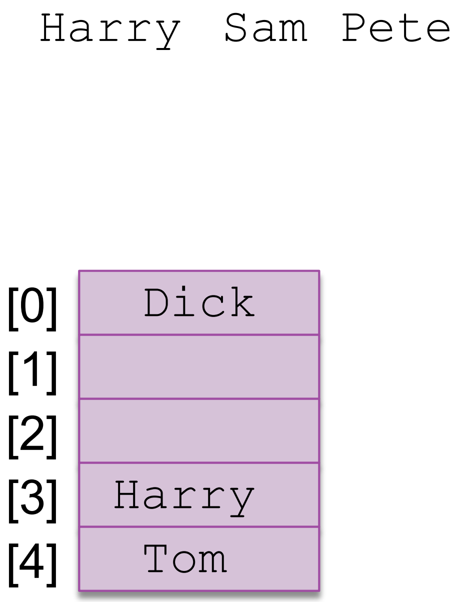
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



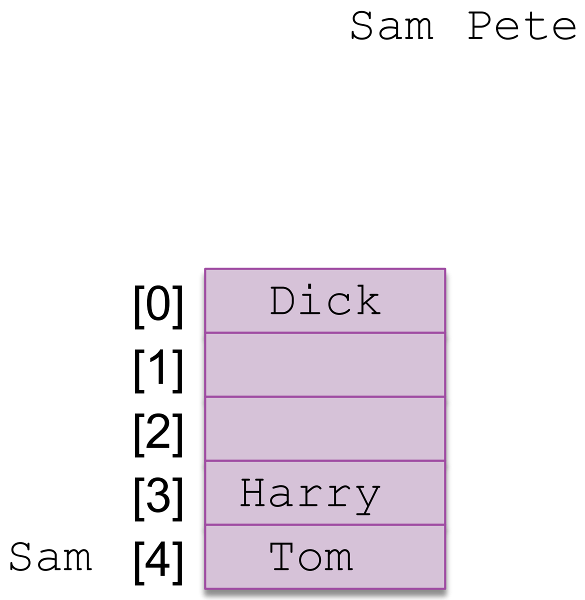
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



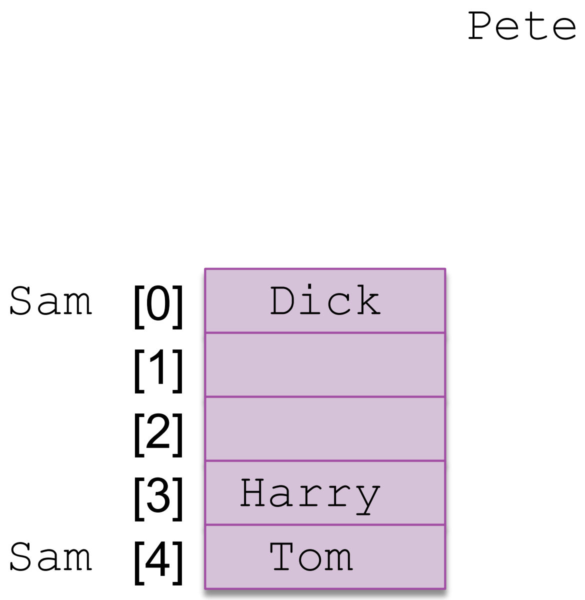
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



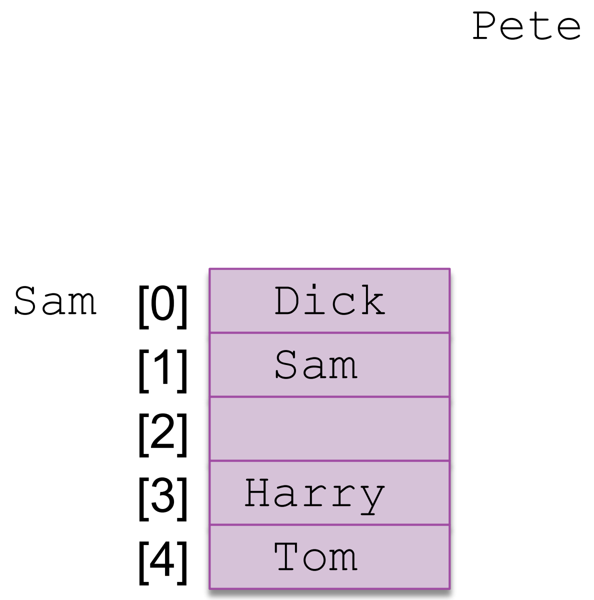
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



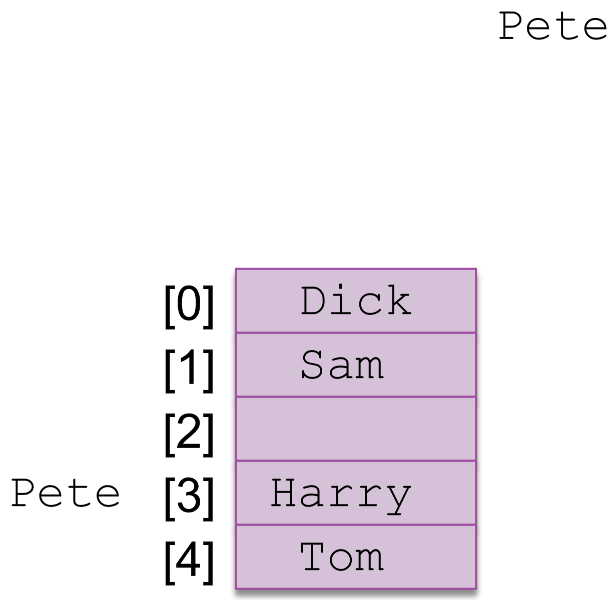
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



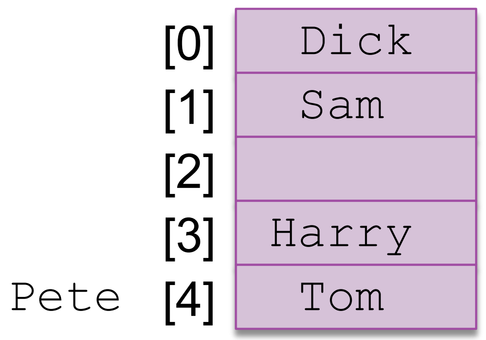
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



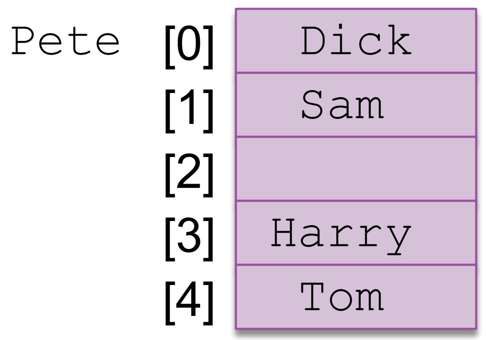
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



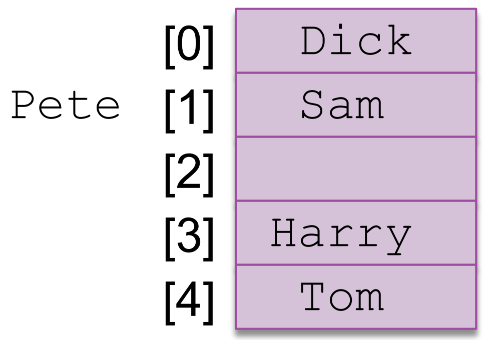
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



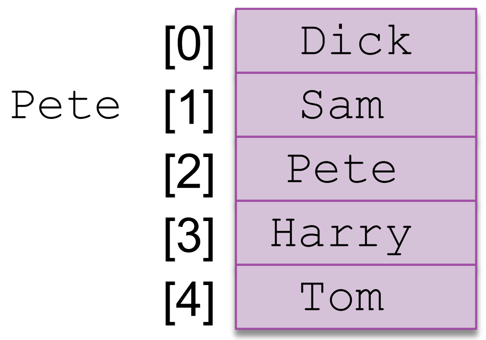
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Hash Code Insertion Example (cont.)



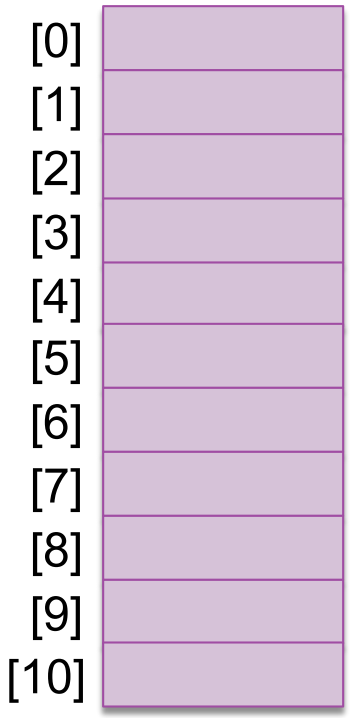
Name	hashCode()	hashCode()%5
"Tom"	84274	4
"Dick"	2129869	4
"Harry"	69496448	3
"Sam"	82879	4
"Pete"	2484038	3

Retrieval of "Tom" or "Harry" takes one step, O(1)

Because of collisions, retrieval of the others requires a linear search

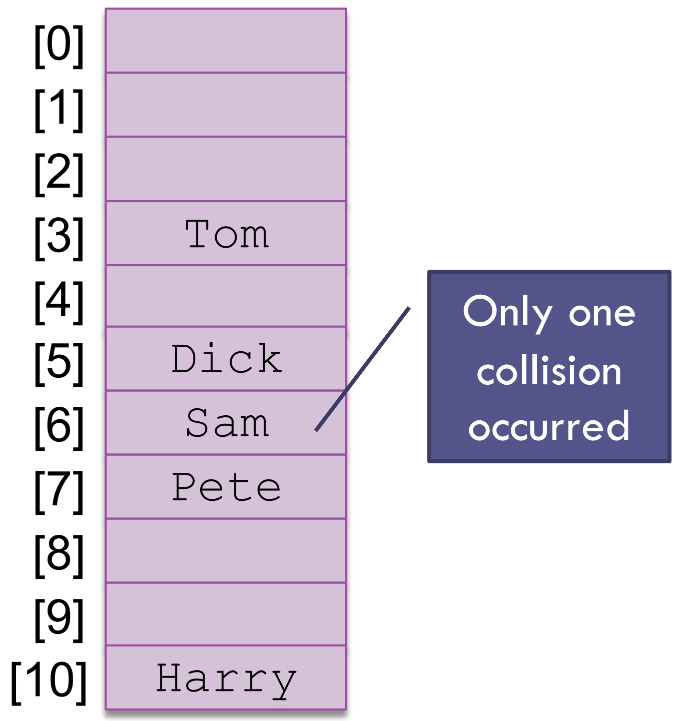
Hash Code Insertion Example (cont.)

Name	hashCode()	hashCode()%11
"Tom"	84274	3
"Dick"	2129869	5
"Harry"	69496448	10
"Sam"	82879	5
"Pete"	2484038	7



Hash Code Insertion Example (cont.)

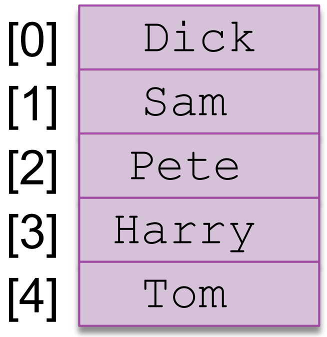
Name	hashCode()	hashCode()%11
"Tom"	84274	3
"Dick"	2129869	5
"Harry"	69496448	10
"Sam"	82879	5
"Pete"	2484038	7



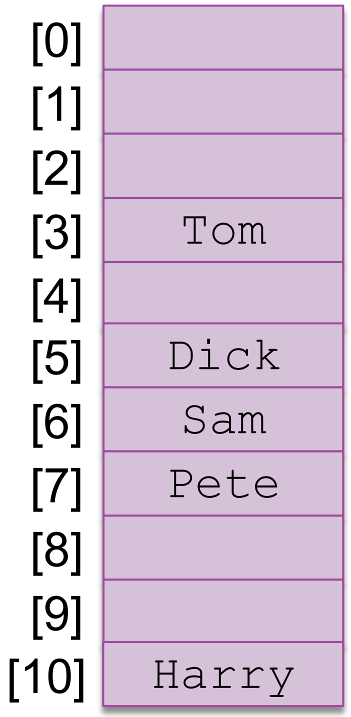
The best way to reduce the possibility of collision (and reduce linear search retrieval time because of collisions) is to increase the table size

Traversing a Hash Table

- You cannot traverse a hash table in a meaningful way since the sequence of stored values is arbitrary



`Dick, Sam, Pete, Harry, Tom`



`Tom, Dick, Sam, Pete, Harry`

Deleting an Item Using Open Addressing

- When an item is deleted, you cannot simply set its table entry to null
- If we search for an item that may have collided with the deleted item, we may conclude incorrectly that it is not in the table.
- Instead, store a dummy value or mark the location as available, but previously occupied
- Deleted items waste storage space and reduce search efficiency unless they are marked as available

Reducing Collisions by Expanding the Table Size

- Use a prime number for the size of the table to reduce collisions
- A fuller table results in more collisions, so, when a hash table becomes sufficiently full, a larger table should be allocated and the entries reinserted
- You must reinsert (*rehash*) values into the new table; do not copy values as some search chains which were wrapped may break
- Deleted items are not reinserted, which saves space and reduces the length of some search chains

Reducing Collisions Using Quadratic Probing

- Linear probing tends to form clusters of keys in the hash table, causing longer search chains
- *Quadratic probing* can reduce the effect of clustering
  - Increments form a quadratic series ($1 + 2^2 + 3^2 + \ldots$)

```
probeNum++;
index = (startIndex + probeNum * probeNum) %
table.length
```

- If an item has a hash code of 5, successive values of index will be 6 (5+1), 9 (5+4), 14 (5+9), . . .

Problems with Quadratic Probing

- The disadvantage of quadratic probing is that the next index calculation is time-consuming, involving multiplication, addition, and modulo division
- A more efficient way to calculate the next index is:

```
k += 2;
index = (index + k) % table.length;
```

Problems with Quadratic Probing (cont.)

- Examples:
  - If the initial value of k is -1, successive values of k will be 1, 3, 5, …
  - If the initial value of index is 5, successive value of index will be 6 (= 5 + 1), 9 (= 5 + 1 + 3), 14 (= 5 + 1 + 3 + 5), …
- The proof of the equality of these two calculation methods is based on the mathematical series:

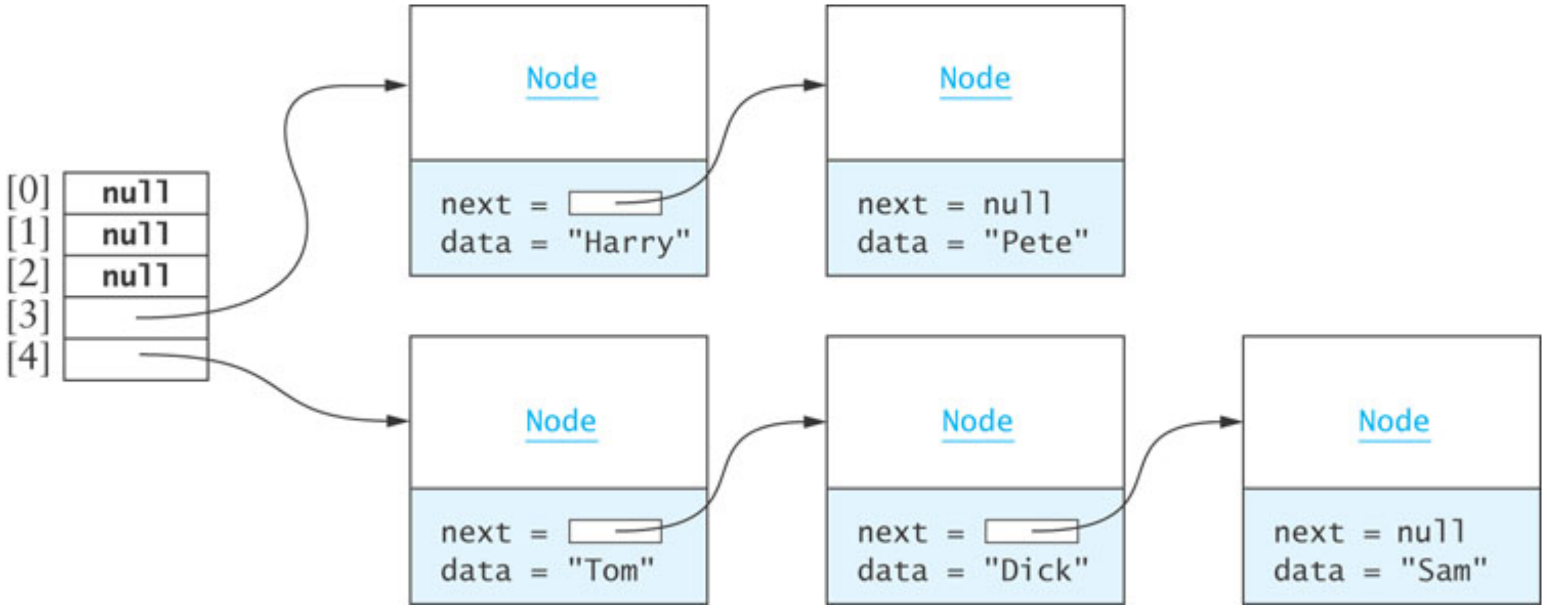
$$n^2 = 1 + 3 + 5 + \ldots + 2n - 1$$

Problems with Quadratic Probing (cont.)

- A more serious problem is that not all table elements are examined when looking for an insertion index; this may mean that
  - an item can't be inserted even when the table is not full
  - the program will get stuck in an infinite loop searching for an empty slot
- If the table size is a prime number and it is never more than half full, this won't happen
- However, requiring a half empty table wastes a lot of memory

Chaining

- *Chaining* is an alternative to open addressing
- Each table element references a linked list that contains all of the items that hash to the same table index
  - The linked list often is called a *bucket*
  - The approach sometimes is called *bucket hashing*



Chaining (cont.)

- Advantages relative to open addressing:
  - Only items that have the same value for their hash codes are examined when looking for an object
  - You can store more elements in the table than the number of table slots (indices)
  - Once you determine an item is not present, you can insert it at the beginning or end of the list
  - To remove an item, you simply delete it; you do not need to replace it with a dummy item or mark it as deleted

Performance of Hash Tables

- *Load factor* is the number of filled cells divided by the table size
- Load factor has the greatest effect on hash table performance
- The lower the load factor, the better the performance as there is a smaller chance of collision when a table is sparsely populated
- If there are no collisions, performance for search and retrieval is O(1) regardless of table size

Performance of Open Addressing versus Chaining

- Donald E. Knuth derived the following formula for the expected number of comparisons, *c*, required for finding an item that is in a hash table using open addressing with linear probing and a load factor *L*

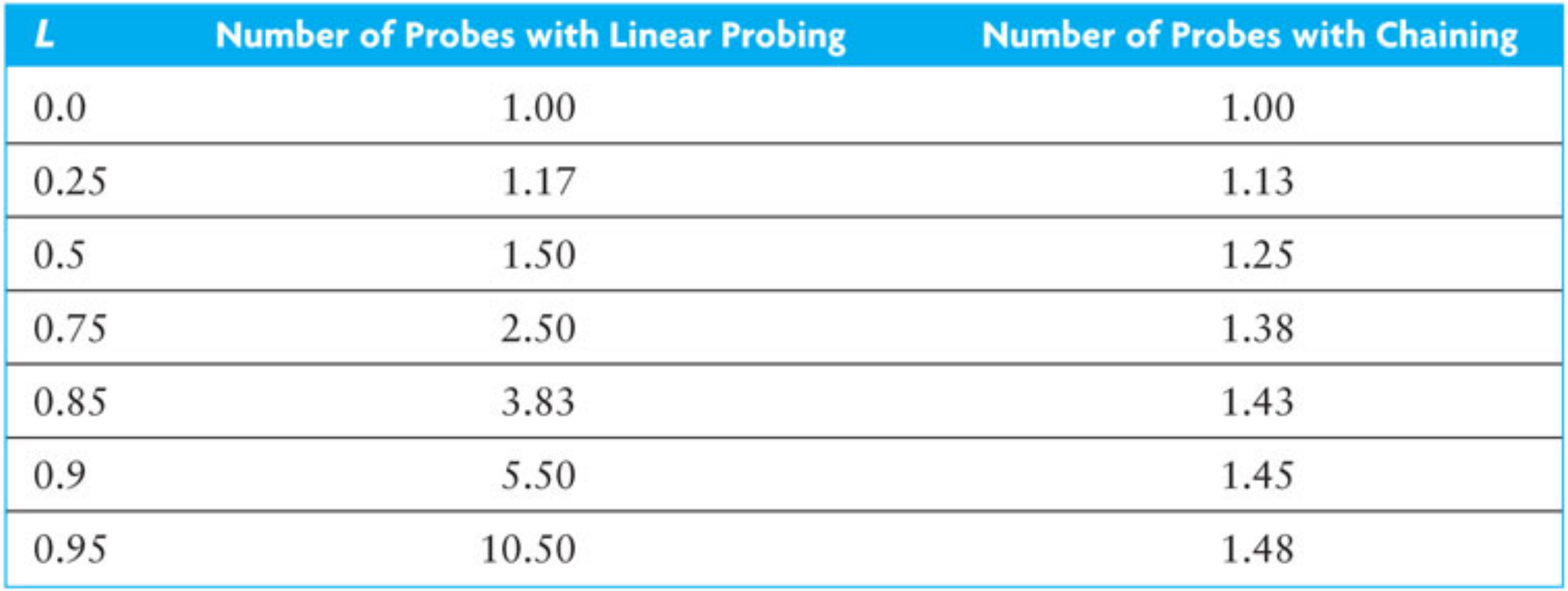
$$c = \frac{1}{2}\,(1 + \frac{1}{1 - L})$$

Performance of Open Addressing versus Chaining (cont.)

- Using chaining, if an item is in the table, on average we must examine the table element corresponding to the item's hash code and then half of the items in each list
- The average number of items in a list is *L*, the number of items divided by the table size

$$c = 1 + \frac{L}{2}$$

Performance of Open Addressing versus Chaining (cont.)



L	Number of Probes with Linear Probing	Number of Probes with Chaining
0.0	1.00	1.00
0.25	1.17	1.13
0.5	1.50	1.25
0.75	2.50	1.38
0.85	3.83	1.43
0.9	5.50	1.45
0.95	10.50	1.48

Performance of Hash Tables versus Sorted Array and Binary Search Tree

- The number of comparisons required for a binary search of a sorted array is O(log *n*)
  - A sorted array of size 128 requires up to 7 probes (2^7 is 128) which is more than for a hash table of any size that is 90% full
  - A binary search tree performs similarly
- Insertion or removal

hash table	O(1) expected; worst case O(*n*)
sorted array	O(*n*)
binary search tree	O(log *n*); worst case O(*n*)

Storage Requirements for Hash Tables, Sorted Arrays, and Trees

- The performance of hashing is superior to that of binary search of an array or a binary search tree, particularly if the load factor is less than 0.75
- However, the lower the load factor, the more empty storage cells
  - there are no empty cells in a sorted array
- A binary search tree requires three references per node (item, left subtree, right subtree), so more storage is required for a binary search tree than for a hash table with load factor 0.75

Storage Requirements for Open Addressing and Chaining

- For open addressing, the number of references to items (key-value pairs) is *n* (the size of the table)
- For chaining , the average number of nodes in a list is *L* (the load factor) and *n* is the number of table elements
  - Using the Java API `LinkedList`, there will be three references in each node (item, next, previous)
  - Using our own single linked list, we can reduce the references to two by eliminating the previous-element reference
  - Therefore, storage for $n + 2L$ references is needed

Storage Requirements for Open Addressing and Chaining (cont.)

- Example:
  - Assume open addressing, 60,000 items in the hash table, and a load factor of 0.75
  - This requires a table of size 80,000 and results in an expected number of comparisons of 2.5
  - Calculating the table size *n* to get similar performance using chaining

    $2.5 = 1 + L/2$

    $5.0 = 2 + L$

    $3.0 = 60{,}000/n$

    $n = 20{,}000$

Storage Requirements for Open Addressing and Chaining (cont.)

- A hash table of size 20,000 provides storage space for 20,000 references to lists
- There are 60,000 nodes in the table (one for each item)
- This requires storage for 140,000 references (2 x 60,000 + 20,000), which is 175% of the storage needed for open addressing