Lesson 8
A Review of Data Structures:
Fully Developing the Container of Knowledge

Wholeness of the Lesson

An analysis of the average-case and worst-case running times of many familiar data structures (for instance, array lists, linked lists, stacks, queues, hashtables) highlights their strengths and potential weaknesses; clarifies which data structures should be used for different purposes; and points to aspects of their performance that could potentially be improved. Likewise, finer levels of intelligence are more expanded but at the same time more discriminating. For this reason, action that arises from a higher level of consciousness spontaneously computes the best path for success and fulfillment.

Abstract Data Types

1. Definition: A set of objects together with a set of operations

2. Example: the Integer data type consists of a range of integers together with the operations of addition, multiplication, and many others

The LIST ADT

1. The set of objects is any ordered list $A_1, A_2, \ldots, A_n$

2. Typical operations:

find(Object o)	returns position of first occurrence
findKth(int pos)	returns element based on index
insert(Object o, int pos)	inserts object into specified position
remove(Object o)	removes first occurrence of object
printList()	outputs all elements
makeEmpty()	empties the List

3. Other specialized operations are often considered, like *findMin*, *findMax*, *contains*.

4. Can be implemented in more than one way.

Array Implementation of LIST

1. In this implementation, an array is maintained internally in a List class and LIST operations are performed by internally carrying out array operations.

2. ***Advantage:*** Provides fastest (O(1)) implementation of *findKth*

3. ***Disadvantages:***

    - *insert* and *remove* require copying an average of half the array elements (running time: O(*n*))

    - periodic array resizing may be necessary

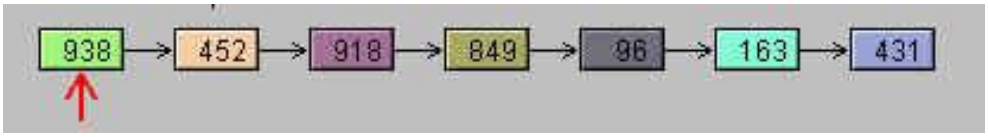
4. Other running times:

    - *makeEmpty*, *printList*, *find* are all O(*n*)

5. Examples in Java: `ArrayList` and `Vector` are array-based implementations of LIST. Both implement the `RandomAccess` interface

LinkedList Implementation of LIST

1. The Need: Improve performance of *insert, remove* and avoid the cost of resizing incurred by the array implementation.



2. A LinkedList consists of Nodes. In addition to storing an Object, a Node contains a link to the next Node (which may be Null).

3. ***Operations***

   - *find* requires traversing the Nodes via links, starting at the first Node.

   - *insert* requires traversing the Nodes to locate position and adjusting links

   - *remove* requires doing a *find*, and when the object is found, the *previous* object has to be located so that it can be linked to the *next* object

   - *using headers* - To make the remove operation uniform, headers are typically used, in conjunction with a doubly linked list; headers store no data; they are just placeholders.

Running Times for LinkedList

1. *find* and *findKth* are O(*n*)

2. *insert* and *remove* are O(*n*), but are faster than in the array implementation because re-copying elements has been avoided

3. *printList* and *makeEmpty* are O(*n*)

Circular Linked Lists

1. In a circular LinkedList, the last element has a link to the first (next) and first has a link to the last (previous)

2. Using a circular LinkedList cuts running time of *findKth* and *insert* operations in half because one can start searching for the specified index from either the front or back of the list; however, *find* and *remove* are no more efficient than before.

When to Use What

1. When the *findKth* operation is used more often than *remove* and *insert*, an array-based implementation is a good idea because it makes use of the random-access feature of the array.

2. If the main search operation requires iteration through the entire List, and insertions and/or deletions are frequent, a LinkedList is preferable since insertions/deletions are faster and the *find* operation is essentially the same for both implementations.

Application

- BucketSort and RadixSort

  Recall our use of lists in BucketSort and RadixSort. In both cases, the best choice of list is a circular linked list because we need to read values from the front of the list and add new values to the end of the list. In a circular linked list, both of these operations are O(1).

Main Point

Array Lists provide O(1) performance for lookup by index because of random access provided by the background array, but perform insertions and deletions in $\Theta(n)$ time, with extra overhead because of the need to break the underlying array into pieces. Linked lists improve the performance of insertion and deletion steps to O(1) though locating the insertion point still requires $\Theta(n)$ time. On the other hand, linked lists, lacking random access, perform reads of all kinds in $O(n)$ time.

Science of Consciousness: Wholeness contains within it diverse – even contradictory – values; these opposite values that are integrated within wholeness make wholeness a field of all possibilities. Experience of this wholeness value brings into our lives the ability to handle and make good use of opposite values.

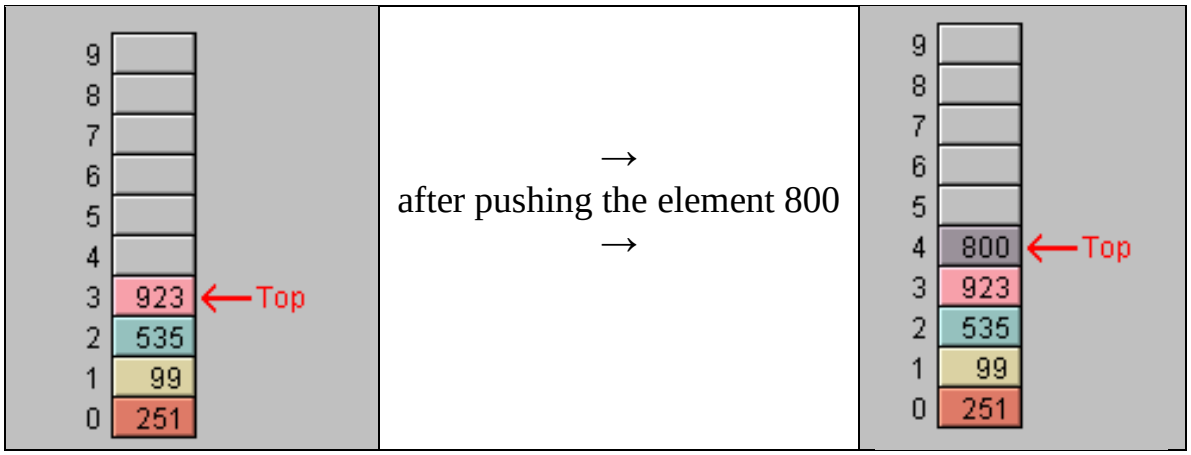
The STACK ADT

3. **Definition**: A STACK is a LIST in which insertions and deletions can occur relative to just one designated position (called the *top of the stack*).

4. **Operations**:

pop	remove top of the stack (usually also returns this object)
push	insert object as new top of stack
peek (or top)	view object at top of the stack without removing it

5. **Model:**



6. **Running Times**

   All operations, under a reasonable implementation, run in constant time.

7. **Implementation**

   Stacks can be implemented using arrays (rightmost element is top) or linked lists.

Implementation of STACK in Java Libraries

1. The standard Java distribution comes with a `Stack` class, which is a subclass of `Vector`.

2. `Vector` is an array-based implementation of LIST. Therefore, for implementations that require many more pushes than pops, a stack based on a Linked List should be used.

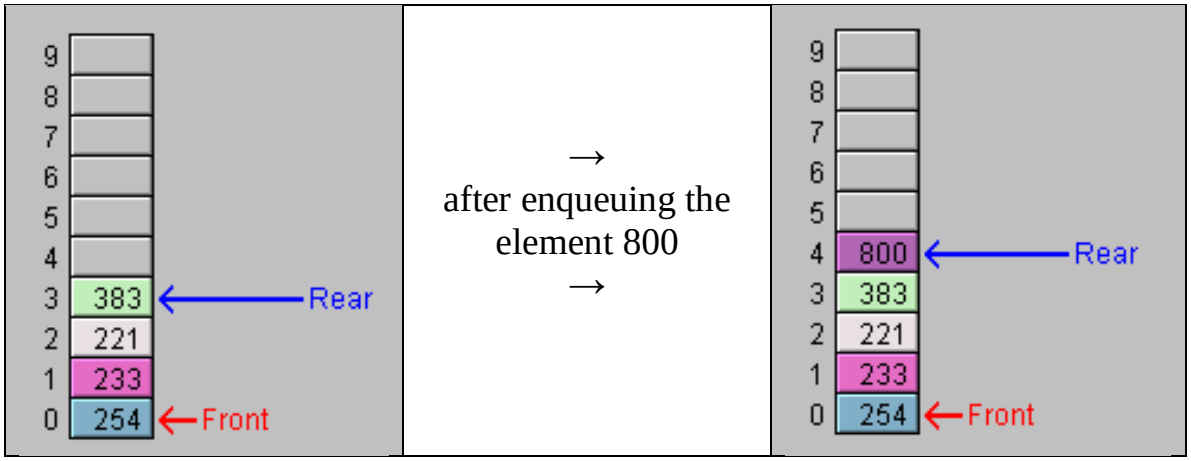
The QUEUE ADT

1. **Definition.** Like a STACK, a QUEUE is a specialized LIST in which insertions may occur only at a designated position (the *back*) and deletions may occur only at a designated position (the *front*).

2. **Operations**:

dequeue	remove the element at the front and return it
enqueue	insert object at the back
peek	view object at front of queue without removing it

3. **Model:**



4. **Running Times**

   All operations, under a reasonable implementation, run in constant time.

5. **Implementation of Queues**

   Queues can be implemented using an array or linked list. To avoid wasted space, in the array implementation, link last array slot to first to provide a "circular array".

Java's Implementation

As of jdk1.5, Java provides a Queue interface that is implemented by LinkedList and several other collections classes.

Main Point

Stacks and queues achieve O(1) performance of their main operations, which involve either reading / removing the top element or inserting a new element either at the top or the end. Stacks and queues achieve their high level of efficiency by concentrating on a single point of input (top of stack or end of queue) and a single point of output (top of stack or front of queue).

Science of Consciousness: Stacks and queues make use of the principle from Maharishi Vedic Science that the dynamism of creation arises in the concentration of dyamic intelligence to a point value ("collapse of infinity to a point").

The HASHTABLE ADT

1. A **Hashtable** is a generalized array. The keys in an array (i.e. their indices) are always integers 0, 1, 2 . . ., but often there is a need for other keys – integers with other values (for instance, employee IDs) or keys of other types (for instance, strings as keys in a lookup table or vertices for a graph).

2. In a hashtable, each key is transformed to an index (called the *hash value* of the key) in an underlying array (called the *table* of the hashtable).

3. A request to read the value matched with a given key is done by transforming the key to its hash value *k* and examining the value stored at position *k.*

4. The worst case for a hashtable occurs when all keys are mapped to the same hash value – in that case, all values are stacked up (typically in a list) in a single slot. This results in O(*n*) performance for *find*, *insert,* and *remove* operations.

5. On average though, with an appropriate implementation, these operations run in O(1) time.

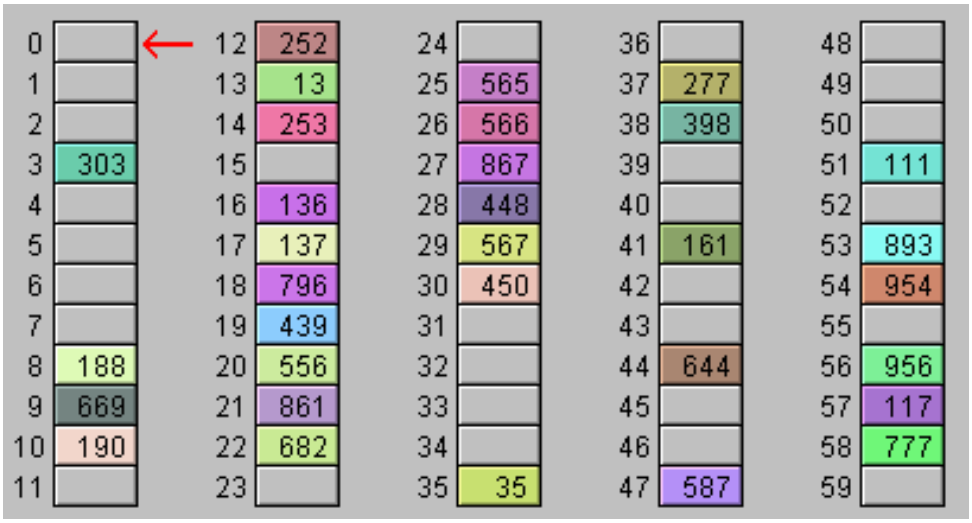
Hashing Strategy

1. In a Hashtable, values lying in a large range are compressed into a (typically) much smaller range consisting of array indices.

2. This transformation is accomplished in two phases:
   a. create a *hashcode* for each key object in the data to be stored
   b. create a *hash value* for each hashcode

3. Ultimately, key objects are transformed into array indices. Since a key object may be a string or other type of non-numeric object, a hashcode is created to provide an integer key. The "compression" of keys into a smaller range typically occurs when these hashcodes are then hashed to hash values (if possible, no compression occurs in creating hashcodes).

4. The model below shows how array indices (hash values) are matched up with keys (hash codes). (Here, the original keys already happen to be integers so that Step (a) described earlier can be skipped.)



Hashing Example

Example. To store records loaded into memory from a database, the following steps are typical:

a. Decide on an array size (which is also called the *table size*)

b. Decide on the field to be used as a key for hashing (like employee number)

c. Define hashcode function *hashCode* and hash function *h*. Typical hash function is h(x) = x % tableSize, but there are variations (as in Java)

d. For each record:
   i. extract the key object *key_ob*
   ii. apply *hashCode* to obtain an integer key (a hashcode)
   iii. apply *h* to obtain an array index
   iv. insert the record at position *h*(*code*(*key_ob*)) in the array

Finding records follows the same pattern: to find an object with key *key_ob*, the hashcode is computed and this code in turn is hashed to locate the array index. The record is then retrieved from the array using this index.

Obstacles to Creating an Efficient Hashtable

1. The worst case scenario for a hashtable occurs when all keys are hashed to the same value. This results in O(n) running time for all operations.

2. Given a hash function h, when two keys (hashcodes) k_1 and k_2 are hashed to the same value – that is, $h(k_1) = h(k_2)$ – we have a *collision.* Too many collisions lead to the worst case running time.

3. Collisions arise for two reasons:

    a. *A certain number of collisions are unavoidable.* Often, we have more keys than there are slots in the array, so a certain number of collisions are inevitable in this case. Even with fewer keys, collisions cannot usually be avoided entirely, in practice.

    b. *There may be hidden regularities in the keys.* When the hashcodes we create are assigned uniformly to the possible hashcode values, collisions can be minimized. But often there are hidden patterns in the data that can cause a particular hash function to produce far too many collisions at certain hash values.

Example Suppose keys (hashcodes) are numbers in the range 0..99, inclusive, and compressed into a table of size 9 (i.e. an array with indices 0..8) using the commonly used *division method*:

$$h(x) = x \% 9$$

If keys are uniformly distributed (all keys are equally likely), the average number of keys that will be mapped to any index i is $100/9 \approx 11$. In other words, on average, for each index i there will be 11 keys x for which $h(x) = i$.

Now consider a different set of 100 keys in which there is a hidden regularity: Suppose all keys chosen happen to be a multiple of 3 (for example, keys are in the range 0-299). Computing hash values mod 9 leads to only three possible values: 0, 3, 6. Therefore, each of the 100 keys is mapped to one of these three possible values. In this case, for every one of the reachable values, on average 33 of the keys will be mapped to that value.

This means that 3 times as many collisions occur using the same hashing strategy and same number of keys, all because of a (possibly unknown) regularity in the keys

Creating Good Hashcodes

The main idea is to “mix up” the assignment of data to numeric values as much as possible. If possible, map different data objects to different numbers, or come as close as possible to this.

In Java, like most enterprise-level languages, there is in each class a hashCode() function that you may override, which allows that type of object to occur as a key in a hashtable.

Joshua Bloch established a standard way to override hashCode() in a class, based on the hashCodes of the data fields inside the class.

Overriding `hashCode()` in Java

(From Bloch, *Effective Java, 2nd Ed.*)

You are trying to build a hashcode for a class, making use of the (hashcodes of) each of the instance variables. Suppose `f` is such an instance variable.

- If `f` is boolean, compute `(f ? 1 : 0)`
- If `f` is a byte, char, short, or int, compute `(int) f.`
- If `f` is a long, compute `(int) (f ^ (f >>> 32))`
- If `f` is a float, compute `Float.floatToIntBits(f)`
- If `f` is a double, compute `Double.doubleToLongBits(f)` which produces a long `f1`, then return `(int) (f1 ^ (f1 >>> 32))`
- If `f` is an object, compute `f.hashCode()`

Formula for creating your hashCode function

Step 1. Use the table above to produce a temporary hash of each variable in your class.
*Example***:** You have variables `u, v, w`. Produce (using the chart above) temporary hash vals `hash_u, hash_v, hash_w`.

Step 2. Combine these temporary hashes into a final hashCode that is to be returned
Example:

```
int result = 17;
result += 31 * result + hash_u;
result += 31 * result + hash_v;
result += 31 * result + hash_w;
return result;
```

Defining the Hash Function: Handling Collisions

1. Collisions are inevitable, how to handle them?

2. Two considerations:

    a. How does the strategy perform under the assumption that keys are evenly ("uniformly") distributed?

    b. What should be done when you suspect that keys may *not* be evenly distributed?

    Our analysis will begin with the assumption that keys are evenly distributed.

3. The most commonly used method for defining a hash function is the ***Division Method.*** This method defines the hash function *h* by

$$h(k) = k \% m$$

    where *m* is the table size and *k* is an integer key (i.e. a hashcode). Java's approach is different (and cannot be modified – only hashCode can be overridden).

    This approach distributes the range of hashCodes as evenly as possible. Java's approach, however, attempts to handle subtler regularities in the hashCodes as well (addressing (b) above), so they take a different approach.

Handling Collisions: Open Addressing

1. In Open Addressing, when $h(k)$ is an array slot that is already occupied, a *probe* is performed to find an available slot. We always assume that the Division Method is used in defining h.

2. ***Rehashing.*** When there are no available slots, this is called *data overflow*. When this happens, the hashtable must be *rehashed* – this means a larger array must be created; new hash values must be assigned to the keys; and the new array must be populated.

3. ***Linear Probes.*** In a linear probe, the following sequence of slots is examined until a free slot is found, if $h(k)$ happens to be occupied:

   $$h(k),\ h(k)+1,\ h(k)+2,\ \ldots$$

   The sequence wraps around after it reaches the end of the array. (Demo)

Handling Collisions: Open Addressing, continued

4. ***Quadratic Probes.*** The following probe sequence is used for a quadratic probe:

$$h(k),\ h(k)+1^2,\ h(k)+2^2,\ h(k)+3^2,\ \ldots$$

5. ***Double Hashing.*** In double hashing, a second hash function h' is defined, and the following probe sequence is used:

$$h(k),\ h(k)+h'(k),\ h(k)+2*h'(k),\ h(k)+3*h'(k),\ \ldots$$

Problems with Open Addressing

- ***Deletions Using Open Addressing.*** Since probe sequences are used for both insertions and searches, objects cannot be removed from the table. Instead, a marker is used to indicate that the object has been "logically deleted".
- ***Sensitive to Load Factor.*** Load factor is α = numElements/tableSize. In open addressing, when load factor reaches (approximately) 80%, performance degrades and a rehash is needed.

Handling Collisions: Separate Chaining

1. With separate chaining, the underlying array is an array of linked lists rather than hashtable entries. Insertion into the hashtable follows these steps:

    a. hash the integer key *k* to a value *j* in the range [0, *m* - 1], where *m* is the table size (as usual)

    b. insert the object with key *k* at the front of the linked list that lives in slot *j*

2. ***Advantage: Load Factor No Longer an Issue.*** With separate chaining, load factor can exceed 1 without a penalty in performance. The average search time is proportional to the average length these linked list. Assuming keys are all equally likely to occur, this average is equal to the load factor:

    $$\alpha = \text{number of elements / table size}$$

3. ***Advantage: Deletions Are Real.*** Values are really deleted in a remove operation, making more space available.

4. ***Disadvantage.*** Separate chaining requires more overhead than open addressing.

Handling Collisions: Separate Chaining, continued

Efficiency:

a. For unsuccessful searches and insertions, expected number of slots checked:

$$\alpha$$

b. For successful searches, the expected number of slots checked:

$$\alpha/2$$

c. This is a table of some values:

value of load factor α	expected num collisions in successful search	expected num collisions in unsuccessful search
.50	1.3	1.5
.75	1.4	1.8
1.0	1.5	2.0
2.0	2.0	3.0
5.0	3.5	6.0

Optional: **Universal Hashing**

1. Our efficiency analysis for open address hashtables as well as for separate chaining hashtables is based on the assumption that keys are evenly distributed. Often this is a reasonable assumption.

2. **Universal Hashing** is a technique, which is used here in conjunction with Separate Chaining, that ensures expected O(α) performance (where α denotes the load factor) of *find, insert, remove* in a hashtable even when the keys may have otherwise deadly regularities.

3. ***Objective:*** Devise a scheme for producing hash values that guarantees expected performance of basic operations in O(α) time. Though we have already established this performance for separate chaining hashtables, the assumption before was that keys were evenly distributed. The goal here is to eliminate this assumption.

4. ***Definition.*** A finite set $\boldsymbol{H}$ of hash functions $[0, M\text{-}1] \rightarrow [0, m\text{-}1]$ (assuming all integer keys lie in the range $[0, M\text{-}1]$ and table slots are in the range $0..m\text{-}1$) is a *universal family* if for each pair of distinct keys i, j, the number of hash functions h in $\boldsymbol{H}$ for which $h(i) = h(j)$ is at most

$$|\boldsymbol{H}| / m.$$

   Note: This means that the probability that a hash function h, when selected from $\boldsymbol{H}$ at random, happens to cause the collision $h(i) = h(j)$ is $1/m$, where m is the table size.

5. ***Main Result about Universal Families.***

   Theorem. Suppose ***H*** is a universal family of hash functions $[0, M\text{-}1] \rightarrow [0, m\text{-}1]$. Let j be an integer in $[0, M\text{-}1]$ and let S be a set of n integers in the same range. Suppose h is chosen uniformly at random from ***H.*** Then the expected number of collisions between j and the integers in S is at most n/m.

6. ***Construction of a Universal Family.*** Let p be a prime greater than the largest key. For each a, b, with $0 < a < p$ and $0 \leq b < p$, define

$$h_{a,b}(k) = ((ak+b) \bmod p) \bmod m$$

   Then the family $\boldsymbol{H} = \{ h_{a,b} \mid 0 < a < p \text{ and } 0 \leq b < p \}$ is a universal family.

When to Use What

1. ***When to Use Open Addressing***
   When all of the following are true, open addressing, using double hashing preferably, is a good choice:

   a. Delete operations are not frequent (otherwise the table will become filled with delete markers and this will affect performance)

   b. An upper bound on the expected number of elements is known (otherwise, rehashing could become too frequent)

   c. There is a great deal of data (the small overhead for open addressing allows more table elements to be fit into the same amount of memory)

2. ***When to Use Separate Chaining***
   Separate Chaining is the natural first choice for most purposes. It handles deletes properly and handles unexpected growth of the load factor in a robust manner. Separate Chaining is used by Sun in Java in the HashMap and Hashtable classes.

3. ***When More Is Needed***
   Even separate chaining isn't good enough if there is a possibility of hidden regularities in the keys that can skew the behavior of the hash function. To solve this problem, don't hesitate to use universal hashing in conjunction with separate chaining.

Optional: Implementation in Java Libraries

1. In the Java Collections API, the classes `HashMap` and `Hashtable` are provided. The main differences between them are:

   a. `HashMap` accepts `null` values while `Hashtable` does not
   b. Methods in `Hashtable` are synchronized, but not in `HashMap`

   We focus on `HashMap`, but the same remarks apply to `Hashtable`.

2. `HashMap` uses separate chaining to resolve collisions. The underlying array is named `Entry[]`, and `Entry` is an inner class of `HashMap`. An `Entry` is a list node. New objects are added to the table by inserting a new `Entry` as the first node.

3. The table size in `HashMap` is always a power of 2. If the `initialCapacity` (i.e. the table size) is set by the user, the `HashMap` routines expand this value to the nearest power of 2.

4. For purposes of producing a hashcode, `HashMap` uses a method `hash` that obtains the hashcode value for the given `Object`, and then transforms it further to avoid regularities that might arise in a table having size a power of 2.

5. By default, `HashMap` sets 0.75 as a load factor threshhold; unless this default is overridden by the user, when load factor exceeds this threshhold value, the table is rehashed. If initial table size is not specified (the `initialCapacity` variable), the default value of 16 is used.

6. Usual way of defining the hash function is not used. Instead:

   ```
   h(k) = hashCode(k) & tablesize - 1
   ```

   to compress hashcodes to table indices, and this value is computed in the method `indexFor`. (Verify that for any positive a and b, a `&` $b \leq \min(a,b)$.)

The SET ADT

Characteristic. A set is a collection which does not attempt to maintain an ordering of elements and which does not allow duplicate values.

Main Operations

void add(ob)
Object remove(ob)
boolean find(ob)

Implementation Use a hashtable (`Hashmap h = new Hashmap()`) as a background data structure (in the Java libraries, `HashSet` is implemented this way). Main operations run in O(1) on average.

```
void add(ob) {
   h.put(ob, ob);
}

Object remove(ob) {
   return h.remove(ob);
}

boolean find(ob) {
   return h.containsKey(ob);
                              }
```

MAIN POINT

Hashtables are a generalization of the concept of an array. They support (nearly) random access of table elements by looking up with a (possibly) non-integer key, and therefore their main operations have an average-case running time of O(1). Hashtables illustrate the principle of *Do less and accomplish more* by providing extremely fast implementation of the main List operations.