Introduction

- Java Collection Framework (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 into 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 $O(\log n)$
Next, 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

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
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: \(\{1, 3, 5, 7\} \cup \{2, 3, 4, 5\}\) 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: \(\{1, 3, 5, 7\} \cap \{2, 3, 4, 5\}\) 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: \(\{1, 3, 5, 7\} \subset \{1, 2, 3, 4, 5, 7\}\) is true
The Set Interface and Methods (cont.)

Using sets in Java

```java
import java.util.Set;

Set<String> setA = new HashSet<String>();
Set<String> setB = new TreeSet<String>();
```

HashSet is implemented using Hash Table (coming next)

TreeSet is implemented using a special kind of Binary Search Tree – Red-Black Trees.
setA.addAll(setB);  // Set Union

System.out.println(setA);

Outputs:
[Bill, Jill, Ann, Sally, Bob]
If a copy of original setA is in setACopy, then . . .

setACopy.retainAll(setB); // Set Intersection
System.out.println(setACopy);

Outputs:
[Jill, Ann]
The *Set* Interface and Methods (cont.)

```java
setACopy.removeAll(setB); // Set Difference
System.out.println(setACopy);
```

Outputs:
[Sally]

---

Comparison of Lists and Sets

- Collections implementing the *Set* interface may contain only 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.

```java
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)

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.

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

Map Hierarchy

![Map Hierarchy Diagram](image)
### Map Interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>V get(Object key)</td>
<td>Returns the value associated with the specified key. Returns \texttt{null} if the key is not present.</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>Returns \texttt{true} if this map contains no key-value mappings.</td>
</tr>
<tr>
<td>V put(K key, V value)</td>
<td>Associates the specified value with the specified key in this map (optional operation). Returns the previous value associated with the specified key, or \texttt{null} if there was no mapping for the key.</td>
</tr>
<tr>
<td>V remove(Object key)</td>
<td>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 \texttt{null} if there was no mapping for the key.</td>
</tr>
<tr>
<td>int size()</td>
<td>Returns the number of key-value mappings in this map.</td>
</tr>
</tbody>
</table>

### Map Interface (cont.)

- The following statements build a Map object:

```java
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 Diagram](image)
Map Interface (cont.)

aMap.get("B1")

returns:

"Bob"

Map Interface (cont.)

aMap.get("Bill")

returns:

null

("Bill" is a value, not a key)
Map Interface (cont.)

Map<String, String> places = new HashMap<String, Place>();

places.put("Bryn MawrPA", new Place("Bryn Mawr", "PA", "19010"));

Places.get("Bryn MawrPA");
returns

<Bryn Mawr, PA, 19010>
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

- 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.

![Diagram of Hash Code Calculation](image)

- \( f(\text{"Bryn Mawr PA"}) = 2 \)

Hash Codes and Index Calculation

- \( F(<\text{key}>) \) returns an index in range \([0..n-1]\)

- Goal: Similar keys map to different locations in an array.

  \[ f(\text{"Richards"}) = 53 \]
  \[ f(\text{"Richardson"}) = 417 \]

- When two or more keys map to the same location, it is called a collision.

  E.g. \( f(\text{"Richards"}) = 53 \)
  \[ f(\text{"Deepak"}) = 53 \]
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. E.g. 10-letter strings have $26^{10}$ keys!
- Generating good hash codes typically is an experimental process.
- The goal is a random distribution of values.
- Simple algorithms sometimes generate lots of collisions.

Java `hashCode` Method

- Example hash function: sum the `int` values of all
  \[ f_1 = (\text{code}(S_0) + \text{code}(S_1) + \ldots + \text{code}(S_{N-1})) \mod \text{tableSize} \]
  But, this returns the same hash code for "sign" and "sing".

- Example hash function: use position + code
  \[ f_2 = [1 \cdot \text{code}(S_0) + 2 \cdot \text{code}(S_1) + \ldots + N \cdot \text{code}(S_{N-1})] \mod \text{tableSize} \]

- Java uses the following:
  \[ f_3 = S_0 \times 31^{(n-1)} + S_1 \times 31^{(n-2)} + \ldots + s_{N-1} \]
  `<string>`.hashCode() method in Java...
Java `hashCode` Method

- 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:
  \[ 'C' \times 31^2 + 'a' \times 31 + 't' = 67,510 \]
- 31 is a prime number, and prime numbers generate relatively few collisions.

Java `hashCode` Method

<table>
<thead>
<tr>
<th>String</th>
<th><code>hashCode()</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>“Tom”</td>
<td>84274</td>
</tr>
<tr>
<td>“Dick”</td>
<td>2129869</td>
</tr>
<tr>
<td>“Harry”</td>
<td>69496448</td>
</tr>
<tr>
<td>“Sam”</td>
<td>82879</td>
</tr>
<tr>
<td>“Pete”</td>
<td>2484038</td>
</tr>
</tbody>
</table>
Java `hashCode` Method (cont.)

- Because there are too many possible strings, the integer value returned by `hashCode()` can't be unique.
- However, because the `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.
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

1. If the index calculated for an item's key is occupied by an item with that key, we have found the item.
2. If that element contains an item with a different key, increment the index by one.
3. 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.
   6. 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 (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hashCode()</th>
<th>hashCode()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

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.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hashCode()</th>
<th>hashCode()%11</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>5</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>10</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>5</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>7</td>
</tr>
</tbody>
</table>

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

Only one collision occurred.

![Diagram showing hash code insertion example.]
Traversing a Hash Table

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

![Diagram of a hash table with names: Dick, Sam, Pete, Harry, Tom.]

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 reduce search efficiency which is partially mitigated if they are marked as available.
- You cannot simply replace a deleted item with a new item until you verify that the new item is not in the table.
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 + ...)

```java
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),...
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:

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

---

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), ...
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

\[
\text{load factor} = \frac{\text{#filled cells}}{\text{table size}}
\]

- Lower load factor -> better performance
- Higher load factor -> worse performance

- 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 $L$

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

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

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

where $L$ is the average number of items in a list (the number of items divided by the table size)
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

<table>
<thead>
<tr>
<th></th>
<th>Hash Table</th>
<th>Unsorted Array</th>
<th>Binary Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash table</td>
<td>$O(1)$ expected; worst case $O(n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unsorted array</td>
<td>$O(n)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search tree</td>
<td>$O(\log n)$; worst case $O(n)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>