CSE 2123
Collections: Sets and Iterators
(Hash functions and Trees)

Jeremy Morris
What is a Set?

- A Set is an *unordered sequence* of data with no *duplicates*
  - Not like a List where you know where in the collection you can find a data element
  - More like a “bag” of data elements, where you just know that the element is somewhere in the “bag”
  - Also not like a List – no duplicate values are allowed
  - The Set collection is an implementation of the mathematical idea of a Set
Collections - Set

- The Set collection is an implementation of the mathematical idea of a Set
  - \{ 10, 5, 8, 7, 3, 9 \}
  - Elements are not kept in any particular order
  - No duplicate elements
  - Same rules for the Set collection
Collections – Set

Sets rely on a few standard methods:

- Put an object into the Set:
  ```java
  boolean add(E value)
  ```

- Remove an object from the Set
  ```java
  boolean remove(E value)
  ```

- Check to see if the Set contains a value:
  ```java
  boolean contains(E value)
  ```
Collections – Set

- Set methods (cont):
  - Return an iterator over the Set:
    \[ \text{Iterator}\langle E \rangle \text{ iterator}() \]
  - Get the number of elements in the Set:
    \[ \text{int size}() \]
  - Check to see if the Set is empty
    \[ \text{boolean isEmpty}() \]
Like other collections, Java implements a number of different types of Sets

- Each useful under different circumstances
- We will talk about HashSets and TreeSets
- HashSets use a *hash function* to ensure uniqueness of elements
- TreeSets use a data structure known as a *binary tree* to ensure uniqueness of elements
Collections – Set (HashSets)

HashSet of names

- Elements are stored in no particular order
- No two names can be the same
- Hash function ensures uniqueness

HashSet<String>

“Marley, Bob”
“Cool, Joe”
“Brown, Charles”
Collections - Set

- Declaring a HashSet
  - A HashSet of Strings:
    
    ```java
    Set<String> mySet = new HashSet<String>();
    ```

  - A HashSet of Integers:
    
    ```java
    Set<Integer> mySet = new HashSet<Integer>();
    ```
Collections - Set

- Typical Set usage:
  - Fill the Set with values
  - Obtain an Iterator over the Set
  - Process values from the Set until the Iterator has seen all the values
Collections - Iterators

- An *iterator* is an object that we can use to examine elements of a collection
  - Anything that implements the Collection interface will provide an iterator
  - Use *iterators* instead of counters in a while loop
  - while loop:

    ```java
    while (counter < list.size() ) {
        ...
        counter = counter + 1;
    }
    ```

- **Iterator**

  ```java
  while (listIter.hasNext() ) {
      ...
  }
  ```
Collections - Iterators

- We obtain Iterator objects by calling the appropriate method on our Collection
  - Anything that implements the Collection interface must provide a method named `iterator` that we can call to get an appropriate Iterator

```java
List<Integer> intList = new ArrayList<Integer>();
Iterator<Integer> intIterator = intList.iterator();

Set<Double> dblSet = new HashSet<Double>();
Iterator<Double> dblIterator = dblSet.iterator();
```
Collections - Iterators

- Important methods
  - Check to see if there are more elements in the iterator
    ```java
    boolean hasNext()
    ```
  - Get the next object off from the iterator
    ```java
    E next()
    ```
  - Remove the last object seen by the iterator from the ArrayList
    ```java
    void remove()
    ```
Collections - Iterators

- Note that for a Set collection, we cannot use a typical while loop with a counter
  - This works for Lists, because elements in Lists are stored sequentially, so the index has a meaning
  - Elements in Sets are not stored in any particular order, so we cannot access them with an index
    - The only way to see everything in a Set is to use an Iterator
public static void main(String[] args) {
    Set<Integer> myInts = new HashSet<Integer>();

    for (int index=0; index<20; index++) {
        myInts.add(index);
        // create a set of numbers 0 - 19
    }

    Iterator<Integer> myIter = myInts.iterator();

    while (myIter.hasNext()) {
        System.out.println(myIter.next());
    }
}

public static void main(String[] args) {
    Set<String> mySet = new HashSet<String>();

    mySet.add("Marley, Bob");
    mySet.add("Cool, Joe");
    mySet.add("Brown, Charles");

    System.out.println(mySet);

    Iterator<String> myIter = mySet.iterator();
    while (myIter.hasNext()) {
        System.out.println(myIter.next());
    }
}
Recall: The HashSet class uses a hash function to implement a Set

- How does this work?
- Every object to be stored in a HashSet must have an associated hash function
  - The hash function is a function that maps each object to a numeric value
  - This numeric value provides an index for a “bucket” where elements with that numeric value are stored
HashSet of names

- Elements are stored in “buckets”
  - Each bucket has an associated index value
  - The element’s hash function tells us which bucket to use
  - When we need to find an element, just apply the hash function again
  - Result tells us which bucket to look in
What happens if two elements have the same hash function result?

This is known as a *collision*

- In these cases, the HashSet first checks to see if they really are equal (using the objects equals() method).
- If they are not equal, then it must maintain both values at the same index.
What happens if two elements have the same hash function result?

- This is known as a *collision*
  - In these cases, the HashSet first checks to see if they really are equal (using the objects equals() method).
  - If they are not equal, then it must maintain both values at the same index
  - One method: use a LinkedList of values for the buckets
Hashing – User Defined Classes

- When we create our own classes, we need to provide a hash function for our objects
  - The hashCode() method is inherited from the Object class
    - Basic functionality – if two objects have different memory locations, hashCode returns different values
    - Two different “new” declarations, two different locations in memory
  - If we want different behavior, we need to override both the hashCode method and the equals method
How do we override the hashCode method?

- Look at what makes our objects equal
- Come up with a hashCode method based on that
- Student objects are equal if they have the same studentId

So we can use the hashCode of our id member variable as our hashCode for the Student class:

```java
@Override
public int hashCode() {
    return id.hashCode();
}
```
Hashing – User Defined Classes

- Sometimes we need to do more than just reuse an existing hash function
  - Need to come up with our own functions
  - As an example, the hash function for a String $s$ is:
    ```java
    int HASH_MULTIPLIER = 31;
    int h=0;
    for (int i=0; i<s.length(); i++) {
        h = HASH_MULTIPLIER*h + s.charAt(i);
    }
    ```
  - The hashcode value is a mathematical function of each character in the String $s$
  - Similar operations can be written for our own classes
Hashing – User Defined Classes

- `equals()` must also be properly implemented
  - Two objects that are equal according to `equals()` must return the same value from `hashCode()`
  - Two objects that are not equal according to `equals()` can have the same result from `hashCode()` but it should be avoided

- Strings, Integers, etc. all have `hashCode` and `equals` set up properly
  - We need to ensure that our own classes have it as well.
Collections – Set (TreeSets)

TreeSet<String>

- “Brown, Charles”
- “Cool, Joe”
- “Marley, Bob”

- TreeSet of names
  - Elements are stored in sorted order
  - No two names can be the same
  - Stored in a binary tree to ensure uniqueness
Declaring a TreeSet

- A TreeSet of Strings:
  ```java
  Set<String> mySet = new TreeSet<String>();
  ```

- A TreeSet of Integers:
  ```java
  Set<Integer> mySet = new TreeSet<Integer>();
  ```
Collections - TreeSet

- TreeSet use a *binary tree* to store elements
  - A binary tree is a data structure with many applications in Computer Science
  - Elements in a binary tree are stored in *nodes*
    - Each node holds an element of data and references to two child nodes – a “right” child and a “left” child
    - In a *binary search tree*, nodes are kept in sorted order by always putting elements less than the current node in the left child, and elements larger than the current node in the right child.
Collections – TreeSet (Binary Tree)

- An example:
  - Suppose we have elements: \{10, 3, 7, 19, 22, 1\} to store in a binary search tree
  - We add 10 first – it becomes the “head” of the tree
    - It’s children are currently empty (null references)
Collections – TreeSet (Binary Tree)

- \{10, 3, 7, 19, 22, 1\}
  - Next we add the 3
    - It’s less than 10, so it becomes the left-child of the head node

```
10
 ---->
 3
```
Collections – TreeSet (Binary Tree)

- \{10, 3, 7, 19, 22, 1\}
  - Next we add the 7
    - It’s less than 10, so we go down the left path
    - It’s more than 3, so it becomes the right child of 3
Collections – TreeSet (Binary Tree)

- `{10, 3, 7, 19, 22, 1}`
  - 19 is greater than 10
    - It becomes the right child of 10
Collections – TreeSet (Binary Tree)

- \{10, 3, 7, 19, 22, 1\}
  - We add 22
    - 22 is greater than 10, so we go down the right path
    - It is greater than 19, so it becomes the right child of 19
Collections – TreeSet (Binary Tree)

- \{10, 3, 7, 19, 22, 1\}
  - Finally we add 1
    - 1 is less than 10, so we go down the left path
    - It is less than 3, so it becomes the left child of 3
In the end, all of our elements are stored in the tree in sorted order
- Suppose we want to know if 6 is in our tree
- Follow the left branch to 3
- … then the right branch to 7
- … and we see that 7 has no children – so 6 cannot be in the tree
- Searching a binary tree is fast, if the tree is balanced
Collections – TreeSet (Binary Tree)

- What do we mean by *balanced*?
  - Roughly, this means that each side of the tree contains the same number of nodes
  - Another way to think about it – keeping the *height* of the tree small
  - Unbalanced trees can cause problems
Unbalanced trees

- Suppose we inserted elements in a different order
  - \{1,3,7,10,19,22\}
- What would our tree look like then?
Unbalanced trees

- Suppose we inserted elements in a different order
- \{1, 3, 7, 10, 19, 22\}
- What would our tree look like then?
Collections – TreeSet (Binary Tree)

- Unbalanced trees
  - Suppose we inserted elements in a different order
  - \{1,3,7,10,19,22\}
  - What would our tree look like then?
- Unbalanced trees
  - Suppose we inserted elements in a different order
  - \{1,3,7,10,19,22\}
  - What would our tree look like then?
Unbalanced trees

- Suppose we inserted elements in a different order
  - \{1,3,7,10,19,22\}
- What would our tree look like then?

```plaintext
1
  3
    7
      10
        19
```
Unbalanced trees

- Suppose we inserted elements in a different order
- \{1,3,7,10,19,22\}
- What would our tree look like then?
Unbalanced trees

- Suppose we inserted elements in a different order
- \{1,3,7,10,19,22\}
- What would our tree look like then?
- One giant chain of nodes
Unbalanced trees

- Suppose we inserted elements in a different order
- \{1,3,7,10,19,22\}
- What would our tree look like then?
Why is an unbalanced tree a problem?

- Consider the two trees below
- How many steps does it take to find the number 22 in each of them? Which one is faster?
Because balanced trees are so important for efficiency, TreeSet is actually built on a self-balancing binary search tree

- A red-black binary tree structure
- Finding elements in the tree the same as a binary search tree
- Adding elements to the tree slightly more complex
  - Must ensure that the tree remains balanced
  - Worth it, because algorithm gives us balanced trees regardless of the order we insert elements in
Your Turn - Trees

- Provide the binary search tree that would result if the following integers were added to the tree in the given order:
  
  [ 8, 3, 23, 15, 1, 5, 42, 64, 36 ]

- Provide the binary search tree that would result if the following characters were added in the given order:
  
  ['r', 'q', 'z', 'a', 'c', 'l', 'x', 'w', 'b']
public static void main(String[] args) {
    Set<Integer> myInts = new TreeSet<Integer>();

    for (int index=0; index<20; index++) {
        myInts.add(index);
        // create a set of numbers 0 - 19
    }

    Iterator<Integer> myIter = myInts.iterator();

    while (myIter.hasNext()) {
        System.out.println(myIter.next());
    }
}
public static void main(String[] args) {
    Set<String> mySet = new TreeSet<String>();

    mySet.add("Marley, Bob");
    mySet.add("Cool, Joe");
    mySet.add("Brown, Charles");

    System.out.println(mySet);

    Iterator<String> myIter = mySet.iterator();
    while (myIter.hasNext()) {
        System.out.println(myIter.next());
    }
}
Binary trees depend on the ordering of elements to be added to them

- So if we want to use a TreeSet, the elements we add to it must have an ordering

- How do we ensure this?
  - Any object we put in a TreeSet must have some way to compare it with other objects in the Set
  - So for any class that we want to use in a TreeSet, we should implement the Comparable interface
    - Strings, Characters, Integers, etc. all have this built in
    - We need to ensure our own classes have this functionality if we need it
HashSet vs. TreeSet

If we need a Set, what kind of Set should we use?

- If we need the Set to iterate in sorted order, we should use a TreeSet
- Otherwise, we should use a HashSet
  - HashSet allows us to add and remove elements from the Set in constant time – the time it takes to add and remove elements is independent of the number of elements in the Set
  - TreeSet uses log(n) time to add and remove elements from the Set. Fast, but dependent on the number of elements in our Collection