## Department of Computer Science
### Undergraduate Events

#### Events this week

<table>
<thead>
<tr>
<th>Event</th>
<th>Date</th>
<th>Time</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resume &amp; Cover Letter Drop-In Session</td>
<td>Wed., Mar 3</td>
<td>12 – 3 pm (20 mins. sessions)</td>
<td>Rm 255, ICICS/CS</td>
</tr>
<tr>
<td>Find a Job Fast! Info Session</td>
<td>Thurs., Mar 4</td>
<td>12:30 – 1:45 pm</td>
<td>DMP 201</td>
</tr>
<tr>
<td>Townhall Meeting – 1st Year CS Students</td>
<td>Thurs., Mar 4</td>
<td>12:30 - 2 pm</td>
<td>DMP 310</td>
</tr>
<tr>
<td>Faculty Talk – Son Vuong</td>
<td>Thurs., Mar 4</td>
<td>12:30 – 1:45 pm</td>
<td>DMP 201</td>
</tr>
</tbody>
</table>

#### Events next week

<table>
<thead>
<tr>
<th>Event</th>
<th>Date</th>
<th>Time</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Townhall Meeting – Combined Majors/Honours, BA, B.Comm in CS</td>
<td>Thurs., Mar 11</td>
<td>12:30 – 2 pm</td>
<td>DMP 310</td>
</tr>
<tr>
<td>CS Distinguished Lecture Series – Featuring David Parkes</td>
<td>Thurs., Mar 11</td>
<td>3:30 – 4:50 pm</td>
<td>DMP 110</td>
</tr>
</tbody>
</table>

### Administrivia

- Lecture slides (day by day) are on the web:
- Assignment #3 is out
  - Due Thursday March 18, 10:00pm
- Midterm exam
  - Wednesday March 10, 6:00pm
  - Math 100 (BCS and 202 sections)
  - **You will be given helps (UML, APIs)**
- Final exam schedule is now posted
  - CPSC 211 - Apr 23, 7:00pm
  - No location yet announced
The **SortedSet Interface**

Allows the user to retrieve objects from the set in sorted order

To sort a collection, the objects within the collection must be comparable:

- the corresponding class must implement either the `Comparable` interface or the `Comparator` interface.

---

The **Comparable Interface**

The `Comparable` interface is declared as follows:

```java
public interface Comparable<T> {  
    int compareTo(T other)
}
```

the integer returned by `a.compareTo(b)` must adhere to the following convention:

- negative if `a < b`
- zero if `a.equals(b)`
- positive if `a > b`

`compareTo` defines the natural ordering for the class
Implementing `compareTo`

Rules to follow when you implement this method in a class `C`:

- `C` must implement `Comparable<C>`
- must be asymmetric
  - `a.compareTo(b)` and `b.compareTo(a)` must both equal 0 or have opposite signs
- must be transitive
  - if `a.compareTo(b) < 0` and `b.compareTo(c) < 0` then `a.compareTo(c) < 0`
- must be consistent with `equals()`
  - `a.equals(b)` is true iff `a.compareTo(b)` is 0 and `b.compareTo(a)` is 0

The Comparator Interface

Some classes may not have a single natural ordering; employees may be ordered by name or by salary or …

A comparator is an object that defines (encapsulates) one ordering for a class

A comparator has to implement:

```java
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

```java
class EmployeeComparator implements Comparator<Employee>
{
    @Override
    public int compare(Employee e1, Employee e2)
    {
        return e1.getSalary() - e2.getSalary();
    }
}
```
The Comparator Interface

The return value for this method is defined in the same way as for the compareTo method of the Comparable interface:

\[ \text{compare}(a,b) \text{ is like } a\.\text{compareTo}(b) \]

We may define many comparators for a class if we need to order objects of that type in different ways.

Example

Create a Comparator that compares Accounts by id numbers.

```java
public class AccountIdComparator implements Comparator<Account>
{
    public int compare(Account ac1, Account ac2)
    {
        return (ac1.getId() - ac2.getId());
    }
}
```
The SortedSet Interface

public interface SortedSet<E> extends Set<E>
{
    // Views on the sorted set
    SortedSet<E> subSet(E from, E to);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);

    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}

Like Set but keeps elements in ascending order according to
the natural order defined by the compareTo method of
Comparable, or
the compare method of a Comparator

Iterator will traverse elements in the defined order
Array produced by toArray methods is sorted

Additional operations:
first() and last() return min and max elements in set
comparator() returns the Comparator used to sort the
set, or null if the natural order is used
The TreeSet Class

The TreeSet class implements the SortedSet interface. It has the following constructors (among others):

public TreeSet()
// orders the elements according to their // natural order

public TreeSet(Comparator<? super E> c)
// orders the elements according to the // comparator c

Note the use of the bounded wildcard:
Comparator<? super E> c
This indicates that the Comparator must compare types that are supertypes of E (including E itself).

For example, if SavingsAccount is a subclass of Account and BalanceComparator implements the Comparator<Account> interface, then we can create the following TreeSet of SavingsAccount objects:

TreeSet<SavingsAccount> accts
    = new TreeSet<SavingsAccount>(
        new BalanceComparator() );
TreeSet - Time Complexity

The add, remove and contains methods all have a guaranteed \( O(\log N) \) time complexity.

So these operations on a TreeSet are less efficient than for a HashSet (assuming a good hashCode() implementation) but we have to remember that the TreeSet maintains the data in sorted order.

Exercises

Write a generic public static method that removes duplicates from a collection. Note that the Collection interface defines the following method:

```java
public static <E> void removeAll(Collection<? extends E> c);
```

More Exercises:

2nd Ed: P19.12, P21.1, P21.11 (but use HashSet<Integer> rather than their IntSet class)

3rd Ed: P14.12, P16.1, P16.12 (but use HashSet<Integer> rather than their IntSet class)

4th Ed: P14.12, P16.1, P16.3 (but use HashSet<Integer> rather than their IntSet class)
Collection Review

**Collection<E>**
- • Items have positions
- • Allows duplicates
- • Additional ops
- • add(i, E)
- • remove(i)
- • listIterator()
- • etc

**List<E>**
- • Items are in an array
- • add() takes nearly O(1)
- • add(i, E) takes O(n)
- • get(i) takes O(1)
- • remove(i) takes O(n)
- • add/remove using an iterator takes O(n)

**ArrayList<E>**
- • Items are in an array
- • add() takes nearly O(1)
- • add(i, E) takes O(n)
- • get(i) takes O(1)
- • remove(i) takes O(n)
- • add/remove using an iterator takes O(n)

**LinkedList<E>**
- • Items are NOT in array
- • add() takes O(1)
- • add(i, E) takes O(n)
- • get(i) takes O(n)
- • remove(i) takes O(n)
- • add/remove using an iterator takes O(n)

**Set<E>**
- • Items are NOT in array
- • add() takes O(1)
- • remove() takes O(1)
- • contains() takes O(1)
- • iterator returns items in NO order

**HashSet<E>**
- • Items are NOT in array
- • add() takes O(1)
- • remove() takes O(1)
- • contains() takes O(1)
- • iterator returns items in NO order

**TreeSet<E>**
- • Items ARE IN ORDER
- • add() takes O(log n)
- • remove() takes O(log n)
- • contains() takes O(log n)
- • iterator returns items IN ORDER

**SortedSet<E>**
- • Items DON’T have positions
- • NO duplicates

add(E)
remove(E)
iterator() etc