Lecture 5: Comparable, Comparator and the Stack ADT

Exercises

1. Design a class BrowserHistoryImpl that implements the following interface:

   ```java
   public interface BrowserHistory {
       // Adds an URL u after the current one. The new URL becomes both current
       // and the last URL.
       void addURL(String u);

       // Get the current URL. Returns the String "blank" if there are no URLs.
       String getCurrentURL();

       // Returns true if there is an URL before the current one
       boolean canGoBack();

       // Returns true if there is an URL after the current one
       boolean canGoForward();

       // Makes the current URL the previous one
       // Throws a NoSuchElementException if there is no URL before the current
       void goBack();

       // Makes the current URL the next one
       // Throws an NoSuchElementException if there is no URL after the current
       void goForward();
   }
   ```

2. This question deals with some Java we haven’t covered yet, but you may have seen before.

   (a) In Java, certain ArrayLists and certain kinds of arrays can be sorted using the
       standard library. Which kinds?

   (b) (★★) Last class we indirectly discussed a sorted dynamic array data structure that
       tries to optimize searching for elements by using binary search. To facilitate this,
       we must always keep the underlying array in sorted order. Is it possible make a
       generic SortedDynamicArray class?

3. (a) Write a function isBalanced that takes a String s composed of ’(‘ and ’)’ characters
       and returns a boolean saying if the parentheses are balanced. For example:

       ```
       (()) => true
       ()(()()) => true
       (() => false
       ()() => false
       )) => false
       ```
(b) (*) Write a new function isBalanced2 that assumes the String s can use parentheses, brackets or braces: '(', ')', '[', ']', '{', '}'. The new isBalanced2 should check if these are balanced and properly paired. For example:

<table>
<thead>
<tr>
<th></th>
<th>=&gt; true</th>
<th>=&gt; false</th>
</tr>
</thead>
<tbody>
<tr>
<td>[()]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[()]</td>
<td></td>
<td></td>
</tr>
<tr>
<td><a href="">{}</a></td>
<td>=&gt; true</td>
<td></td>
</tr>
<tr>
<td>(){}{}</td>
<td>=&gt; true</td>
<td></td>
</tr>
<tr>
<td>()(){}</td>
<td>=&gt; true</td>
<td></td>
</tr>
<tr>
<td>[][]{}</td>
<td>=&gt; false</td>
<td></td>
</tr>
<tr>
<td>[]</td>
<td>=&gt; false</td>
<td></td>
</tr>
</tbody>
</table>

**Solutions**

1. Below is our implementation:

```java
import java.util.ArrayList;
import java.util.NoSuchElementException;

public class BrowserHistoryImpl implements BrowserHistory {
    private ArrayList<String> data;
    private int current;
    public BrowserHistoryImpl()
    {
        data = new ArrayList<String>();
        current = -1;
    }
    public void addURL(String url)
    {
        while (data.size() > current+1)
            data.remove(data.size()-1);
        data.add(url);
        ++current;
    }
    public String getCurrentURL()
    {
        return current == -1 ? "blank" : data.get(current);
    }
    public boolean canGoBack() { return current > 0; }
    public boolean canGoForward()
    {
        return current+1 < data.size();
    }
}
```

```
public void goBack()
{
    if (canGoBack()) current--; 
    else throw new NoSuchElementException();
}

public void goForward()
{
    if (canGoForward()) current++; 
    else throw new NoSuchElementException();
}

Here we use the ArrayList class that comes with Java, and we have an integer current that points at the current URL.

2. (a) To be able to sort a list of objects, you need to know how to compare each pair of objects. There are two ways to indicate this in Java: the interfaces Comparable<T> and Comparator<T>. You can make your class implement Comparable when you want your objects to have a default ordering amongst themselves. For example, you may want to order students by how many credits they have. Comparator (in java.util package) is used when you want to order a given group of objects, but they either aren’t comparable, or aren’t comparable in the way you want. By creating a separate class that implements Comparator, you are writing an external comparison function that can be used by thing like sorting methods to order your objects. We will see both of these at work below:

    Student.java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.Comparator;

public class Student implements Comparable<Student>
{
    private String name;
    private String id;
    private int credits;
    public Student(String n, String i)
    {
        name = n;
        id = i;
    }
    public void setCredits(int c) { credits = c; }
    public String getID() { return id; }
}
@Override
public int compareTo(Student o)
{
    if (credits < o.credits) return -1;
    if (credits > o.credits) return 1;
    return 0;
    //return Integer.compare(credits,o.credits);
}
@Override
public boolean equals(Object o)
{
    if (o instanceof Student)
    {
        Student s = (Student)o;
        return credits == s.credits;
    }
    return false;
}
public String toString() { return name; }
public static void main(String[] args)
{
    Student a = new Student("A","A");
    Student b = new Student("B","B");
    Student c = new Student("C","C");
    a.setCredits(30);
    b.setCredits(20);
    c.setCredits(20);
    System.out.println("Using Credits");
    System.out.println(a.compareTo(b));
    System.out.println(b.compareTo(a));
    Student[] arr = {a,b,c};
    ArrayList<Student> al =
        new ArrayList<>(Arrays.asList(a,b,c));
    Arrays.sort(arr);
    System.out.println(Arrays.toString(arr));
    Collections.sort(al);
    System.out.println(al);
    System.out.println(Collections.min(al));
    System.out.println("Now by ID");
    Comparator<Student> co =
        new StudentIDComparator();
    Arrays.sort(arr,co);
    Collections.sort(al,co);
    System.out.println(Arrays.toString(arr));
System.out.println(al);
System.out.println(Collections.min(al,co));
}
}

class StudentIDComparator implements Comparator<Student>
{
    public int compare(Student o1, Student o2) {
        return o1.getID().compareTo(o2.getID());
    }
}

A few things to point out above:

i. Student implements the generic type Comparable<Student>. Since interface Comparable<T> has the abstract method int compareTo(T o) we must override it. We have done this above and have compared the credits of each Student. As described in the javadocs for Comparable, we must return a negative number if this object is less than o, a positive number if this object is greater than o, and 0 if they are considered equal by the ordering. Note that Integer provides a helper method we could have used that does the same thing.

ii. We have included the @Override annotations in the code. This doesn’t effect the program at all but has two nice side-effects. Firstly, it shows that reader than we intend to override a method. Secondly, it tells the compiler that we intend to override a method, so if we have a typo and aren’t actually overriding, the compiler will complain. It is a good habit to always include the @Override on every method that is overriding an ancestor’s method.

iii. We overrided the equals method in Object. It is often a good idea to do this so that users can check if two objects are considered equal (since == only checks reference equality). It is an especially good idea to override equals when you have implemented Comparable since users will expected a.compareTo(b)==0 to be the same as a.equals(b). Our equals method above does this. Note that equals takes an Object as argument. The first thing we do is test if the Object is actually a Student. More precisely,

        a instanceof TypeName

        is true if and only if a is not null, and the dynamic type (type of the object referred to by a) is a descendent of TypeName.

iv. In main we show that arrays and ArrayLists of type Student are now sortable. The last thing we want to do is sort students by their ids. To do this we create a StudentIDComparator.

(b) Here we are going to go a bit deeper into generics. Even though we won’t need
this much in class, it will help clear up things you may see in the JavaDocs. If we simply do something like

```java
public class SortedArrayList<T> {
    //stuff
}
```

we cannot be sure that objects of type T are Comparable. To enforce this requirement we can do the following:

```java
public class SortedArrayList<T extends Comparable<T>> {
    //stuff
}
```

This is called a bounded type parameter, since we are restricting our parameters T to extend T extends Comparable<T>. This is a step in the right direction, but the above will not work for the following situation.

**BoundedGenerics.java**

```java
class BParent implements Comparable<BParent> {
    private int val;
    public BParent(int v) { val = v; }
    @Override
    public int compareTo(BParent p) { return Integer.compare(val, p.val); }
}
class BChild extends BParent {
    public BChild(int v) { super(v); }
}

class SortedArrayList<T extends Comparable<T>> {}
class BetterSortedArrayList<T extends Comparable<? super T>> {}
import java.util.ArrayList;

public class Balanced {
    public static boolean isBalanced(String s) {
        int numOpen = 0;
        for (int i = 0; i < s.length(); ++i) {
            char c = s.charAt(i);
            if (c == '(') numOpen++;
            else numOpen--;
            if (numOpen < 0) return false;
        }
        return numOpen == 0;
    }

    public static boolean isBalanced2(String s) {
        ArrayList<Character> al = new ArrayList<Character>();
        String open = "\{\[\), openAt(\}])";
        String close = "\{\[\), closeAt(\}])";
        for (int i = 0; i < s.length(); ++i) {
            char c = s.charAt(i);
            if (open.indexOf(c) != -1) al.add(c);
            else {
                if (al.isEmpty()) return false;
                char d = al.remove(al.size() - 1);
                int j = close.indexOf(c);
                if (d != open.charAt(j)) return false;
            }
        }
        return al.size() == 0;
    }

    public static void main(String[] args) {
        String[] barr = {"(())","()(()())","(()","())()",")("};
        String[] barr2 = {"[\[]","[(\])","[({})[\[]","[()(){}{}]","()[]{}"};
        System.out.println("isBalanced:");
        for (int i = 0; i < barr.length; ++i)
            System.out.println(isBalanced(barr[i]));
    }
}
System.out.println("isBalanced2:");
for (int i = 0; i < barr2.length; ++i)
    System.out.println(isBalanced2(barr2[i]));
}

(a) For isBalanced we simply count how many open parentheses haven’t been closed yet. If this number ever becomes negative, or doesn’t end at zero we return false.

(b) Here we keep adding the opening parenthesis, bracket, or brace to the back of our ArrayList. Whenever we find a closing character we attempt to pair it with the last unpaired opening character, which we find at the back of our ArrayList. If no such character exists, or it is the wrong type of character then we return false. When we are done, we return true if and only if the ArrayList is empty.

ADT: Stack

When writing isBalanced2 above we used an ArrayList but didn’t really need all of its functionality. We could have gotten away with only having the following operations:

Stack.java

```java
public interface Stack<T>
{
    //Adds an element to the Stack
    public void push(T t);
    //Removes the most recently pushed element that wasn’t
    //already popped
    //Throws an NoSuchElementException if the stack is empty
    public T pop();
    //Determines if the stack is empty
    public boolean isEmpty();
}
```

As we have alluded to in the name above, the above operations constitute a Stack ADT. An ADT (Abstract Data Type) is an abstract definition of what behaviors a data structure should support, without stating how the data structure should be implemented. Here we have used a Java interface to describe our ADT, but we could have simply used English text. Some ADTs also include runtime requirements of the behaviors in big-Oh or big-Theta format. Another example of an ADT is a List:

List.java

```java
public interface List<T>
{
    //Remove element in position i and shift later elements to fill gap
```
The dynamic arrays we created last time were data structures that implemented the behavior specified in the List ADT. Note that a List can be used to implement a Stack. In other words, any problem we can solve with a Stack can be solved with a List. This begs the question why we need Stacks at all if we can just use Lists. This also touches on the question of why ADTs are useful. Here are some important points to consider.

1. Communication: If you tell someone your algorithm/code uses a Stack, then you are conveying some information. The person hearing this then realizes that you only need pushing and popping behavior, and thus better understands your algorithm.

2. Optimization: In some cases, a data structure with a simpler interface may allow for a simpler or more performant implementation.

3. More generally, in this class we will often look at a particular ADT and then consider different data structures that implement the ADT. We can then see what the tradeoffs are when using one data structure over another for implementing a given ADT.

Stack Exercises

1. Create a class that implements the `Stack<T>` interface using an ArrayList.

2. (*) Consider the `IntMaxStack` ADT with the following interface:

   ```java
   public interface IntMaxStack
   {
       // Adds an element to the Stack
       void push(int t);
       // Removes the most recently pushed element that wasn't already popped
       // Throws an NoSuchElementException if the stack is empty
       int pop();
       // Gives the maximum value of all values currently on the stack.
       // Throws an NoSuchElementException if the stack is empty
   }
   ```
int getMax();
    // Returns if empty
    boolean isEmpty();
}

(a) Write a class that implements this interface. Give the runtime for each method on your implementation. [It is possible to get Θ(1) for getMax without sacrificing much performance in the other methods.]

(b) How would you make this interface and implementation generic?

Stack Solutions

1. Our solution follows:

    ArrayListStack.java

    import java.util.ArrayList;
    import java.util.NoSuchElementException;

    public class ArrayListStack<T> implements Stack<T> {
        private ArrayList<T> data;

        public ArrayListStack() { data = new ArrayList<T>(); }

        @Override
        public void push(T t) { data.add(t); }

        @Override
        public T pop() {
            if (data.isEmpty()) throw new NoSuchElementException();
            int n = data.size() - 1;
            T t = data.get(n);
            data.remove(n);
            return t;
        }

        @Override
        public boolean isEmpty() { return data.isEmpty(); }
    }

2. (a) Here is one implementation:

    IntMaxStackImpl.java

    import java.util.ArrayList;
    import java.util.NoSuchElementException;

    public class IntMaxStackImpl {
        private ArrayList<Integer> data;

        public IntMaxStackImpl() { data = new ArrayList<Integer>(); }

        @Override
        public void push(int t) { data.add(t); }

        @Override
        public int pop() {
            if (data.isEmpty()) throw new NoSuchElementException();
            int n = data.size() - 1;
            int t = data.get(n);
            data.remove(n);
            return t;
        }

        @Override
        public int getMax() {
            if (data.isEmpty()) throw new NoSuchElementException();
            return data.get(data.size() - 1);
        }

        @Override
        public boolean isEmpty() { return data.isEmpty(); }
    }
public class IntMaxStackImpl implements IntMaxStack {
    private ArrayList<Integer> data;
    public IntMaxStackImpl() { data = new ArrayList<Integer>(); }
    @Override
    public void push(int t) { data.add(t); }
    @Override
    public int pop() {
        if (data.isEmpty()) throw new NoSuchElementException();
        int n = data.size() - 1;
        int t = data.get(n);
        data.remove(n);
        return t;
    }
    @Override
    public int getMax() {
        if (data.isEmpty()) throw new NoSuchElementException();
        int max = data.get(0);
        for (int i = 1; i < data.size(); ++i) max = Math.max(max, data.get(i));
        return max;
    }
    @Override
    public boolean isEmpty() {
        return data.isEmpty();
    }
}

Let $n$ denote the current size of the stack. Then the worst-case runtime for pop is $\Theta(1)$, for getMax is $\Theta(n)$ and for push is $\Theta(n)$ but has $\Theta(1)$ amortized time. Here is a different implementation that uses roughly double the space, but has improved runtime for getMax:

IntMaxStackImpl2.java

import java.util.ArrayList;
import java.util.NoSuchElementException;

public class IntMaxStackImpl2 implements IntMaxStack {
    private ArrayList<Integer> data;
    private ArrayList<Integer> maxes;
    public IntMaxStackImpl2() {
        data = new ArrayList<Integer>();
        maxes = new ArrayList<Integer>();
    }

    public void push(int t) { data.add(t); maxes.add(t); }
    public int pop() {
        int t = data.get(data.size() - 1);
        data.remove(data.size() - 1);
        return t;
    }
    public int getMax() {
        int max = data.get(0);
        for (int i = 1; i < data.size(); ++i) max = Math.max(max, data.get(i));
        return max;
    }
    public boolean isEmpty() {
        return data.isEmpty();
    }
}
@Override
public void push(int t)
{
    data.add(t);
    int max = t;
    if (!maxes.isEmpty())
        max = Math.max(max, maxes.get(maxes.size() - 1));
    maxes.add(max);
}

@Override
public int pop()
{
    if (data.isEmpty())
        throw new NoSuchElementException();
    int n = data.size() - 1;
    int t = data.get(n);
    data.remove(n);
    maxes.remove(n);
    return t;
}

@Override
public int getMax()
{
    if (data.isEmpty())
        throw new NoSuchElementException();
    return maxes.get(data.size() - 1);
}

@Override
public boolean isEmpty() {
    return data.isEmpty();
}

Here we maintain two stacks: a data stack and a max stack. Each element of the max stack stores the max of all the data before it. For example, suppose the stack is (bottom-to-top)

1, 5, 2, 3, 2, 6, 4, 9.

Then maxes will have (bottom-to-top)

1, 5, 5, 5, 5, 6, 6, 9.

On each push and pop we can effectively update both stacks. The push and pop operations have the same runtimes as above, but getMax is now $\Theta(1)$.

(b) Below we have the interface and implementation (we modified IntMaxStackImpl2 above).
MaxStack.java

public interface MaxStack<T>
{
    // Adds an element to the Stack
    void push(T t);
    // Removes the most recently pushed element that wasn't already popped
    // Throws an NoSuchElementException if the stack is empty
    T pop();
    // Gives the maximum value of all values currently on the stack
    // Throws an NoSuchElementException if the stack is empty
    T getMax();
}

MaxStackImpl2.java

import java.util.ArrayList;
import java.util.Comparator;
import java.util.NoSuchElementException;

public class MaxStackImpl2<T> implements MaxStack<T>
{
    private ArrayList<T> data;
    private ArrayList<T> maxes;
    private Comparator<? super T> c;

    public MaxStackImpl2() {
        this(null); // Calls other constructor
    }

    public MaxStackImpl2(Comparator<? super T> co)
    {
        data = new ArrayList<T>();
        maxes = new ArrayList<T>();
        c = co;
    }

    @SuppressWarnings("unchecked")
    private int compare(T a, T b)
    {
        if (c == null)
        {
            Comparable<? super T> ca = (Comparable<? super T>)a;
            return ca.compareTo(b);
        }
        return c.compare(a, b);
    }

    @Override
    public void push(T t)
    {
        // Code for push method
    }
}
data.add(t);
T max = t;
if (!maxes.isEmpty())
{
    T prevMax = maxes.get(maxes.size() - 1);
    if (compare(max, prevMax) < 0) max = prevMax;
}
maxes.add(max);
}@Override
public T pop()
{
    if (data.isEmpty()) throw new NoSuchElementException();
    int n = data.size() - 1;
    T t = data.get(n);
    data.remove(n);
    maxes.remove(n);
    return t;
}@Override
public T getMax()
{
    if (data.isEmpty()) throw new NoSuchElementException();
    return maxes.get(data.size() - 1);
}

Here we properly allow for a Comparator or a Comparable. If no Comparator is provided and no Comparable is provided then a ClassCastException will occur in our private compare helper method.