Hyperiondev

TASK

Algorithms - Collecting Framework

Visit our website

Introduction

Welcome to the Algorithms - Collections Framework Task!

In this task, we will discuss the algorithms contained in the Java collections framework.

Get in touch

Connect for support

Remember that with our courses, you're not alone! You can contact your mentor to get support on any aspect of your course.

The best way to get help is to login to **www.hyperiondev.com/portal** to start a chat with your mentor. You can also schedule a call or get support via email.

Your mentor is happy to offer you support that is tailored to your individual career or education needs. Do not hesitate to ask a question or for additional support!

Java Collections Framework

The Java collections framework was introduced earlier on in this bootcamp. As a quick recap, a collections framework is a unified architecture for representing and manipulating collections. The following are contained in all collection frameworks:

- **Interfaces:** Interfaces are abstract data types that represent collections. They allow collections to be manipulated independently of the details of their implementation. In object-oriented languages, interfaces generally form a hierarchy.

- **Implementations:** Implementations are classes. These are the concrete implementations of the collection interfaces.

- **Algorithms:** Algorithms are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces. The algorithms are said to be polymorphic. This means that, the same method can be used on many different implementations of the appropriate collection interface.

In this task we will be discussing the polymorphic algorithms contained in the Java collections framework. All of these algorithms take the form of static methods whose first argument is the collection on which the operation is to be performed. Most of the algorithms provided by the Java platform operate on List instances however a few of them on arbitrary Collection instances.

In this task we will be discussing the following algorithms:

- Sorting
- Shuffling
- Data Manipulation
- Searching
- Composition
- Finding Extreme Values

Sorting

In the previous Java collections framework task, we introduced the Comparator interface. You can define a comparator to compare the elements of different

classes. To do this, you need to define a class that implements the java.util.Comparator<T> interface. The Comparator<T> interface has two methods, compare and equals.

- **public int compare(T element1, T element2)**
  Returns a negative value if element1 is less than element2, a positive value if element1 is greater than element2, and zero if they are equal.

- **public boolean equals(Object element)**
  Returns true if the specified object is also a comparator and imposes the same ordering as this comparator.

The Comparable interface however, is defined to compare objects of the same type. Many classes in the Java library implement Comparable to define a natural order for objects. The comparable interface defines the compareTo method for comparing objects. The interface is defined as follows:

```
package java.lang;

 public interface Comparable<E> {
     public int compareTo(E o);
}
```

The order of an object with the specified object o is determined by the compareTo method. It returns a negative integer if the object is less than o, zero if the object is equal to o, or a positive integer if this object is greater than o.

The Comparable interface is a generic interface where the generic type E is replaced by a concrete type when it is implemented.

The sorting algorithm reorders the elements of a List into ascending order. There are two static methods provided in the Java collections framework:

- sort(List list)
- sort(List list, Comparator c)

The sort(List list) method sorts the elements in a List in the order defined by the Comparable interface. This sort operation uses a merge sort algorithm that is slightly optimised. This algorithm is fast and stable. It runs in O(n logn) time.

The following program orders characters in a list alphabetically:

```
import java.util.*;


public class Sort {
  public static void main(String[] args) {

    List<String> myList = new ArrayList<>();
    myList.add("Z");
    myList.add("B");
    myList.add("R");
    myList.add("M");

    System.out.println("List: " + myList);

    // The sort method uses the Comparable interface in String class
    Collections.sort(myList);
    System.out.println("Sorted List: " + myList);

  }
}
```

The output of the above code is:

```
List: [Z, B, R, M]
Sorted List: [B, M, R, Z]
```

The sort(List list, Comparator c) method sorts the elements in a List with a Comparator object which is passed to the method. The type of the elements do not need implement the Comparable interface in this case. This is useful if we need to sort a list of custom objects where we can't modify the class, or if we don't want to rely on the natural ordering of the elements.

We will now use a comparator to compare two employees based on their ages:

Suppose we have the following custom type called Employee:

```
public class Employee {
  private String name;
  private int id = 0;
  private int age;

  public Employee(String emp_name, int emp_age, int emp_id) {
    name = emp_name;
    age = emp_age;
    id = emp_id;
  }
```

```
  public int getAge() {
    return age;
  }

  public int getId() {
    return id;
   }

  public String getName() {
    return name;
  }

  public void setFirstName(String fn) {
    name = fn;
  }

  public String toString() {
    return String.format("(%s, %d, %d)", name, age, id);
  }

}
```

The following is an example of a comparator that compares two employees based on their ages:

```
class EmployeeAgeComparator implements Comparator<Employee> {

    @Override
    public int compare(Employee employee1, Employee employee2) {
        return employee1.getAge() - employee2.getAge();
    }
}
```

You can then sort a list of employees using the following code:

```
   public static void main(String[] args) {

     List<Employee> listEmp = new ArrayList<Employee>();

     listEmp.add(new Employee("Tom", 25, 100000));
     listEmp.add(new Employee("Jane", 23, 100001));
     listEmp.add(new Employee("Alex", 36, 100005));
     listEmp.add(new Employee("Sally", 32, 100003));

     System.out.println("Before sorting: " + listEmp);

     Collections.sort(listEmp, new EmployeeAgeComparator());

     System.out.println("After sorting: " + listEmp);
```

```
   }
```

The output of the above code is:

```
Before sorting: [(Tom, 25, 100000), (Jane, 23, 100001), (Alex, 36, 100005),
(Sally, 32, 100003)]
After sorting: [(Jane, 23, 100001), (Tom, 25, 100000), (Sally, 32, 100003),
(Alex, 36, 100005)]
```

Shuffling

The shuffle algorithm does the complete opposite of what the sort algorithm does. While the sort algorithm puts the elements of a List in some sort of order, the shuffle algorithm destroys all traces order that may be present in the List. In other words, the shuffle algorithm shuffling gives us a random permutation of the elements in a List. This is useful if you would like to create a program the implements a game of chance such as a card game. It can also be useful for generating test cases.

Like the sort algorithm, there are two shuffle methods provided in the Java collections framework:

- shuffle(List list)
- shuffle(List list, Random rnd)

The shuffle(List list) method takes in a List object as a argument. And uses a default source of randomness. The shuffle(List list, Random rnd) method however requires a **Random** object to use as a source of randomness.

The following code is used to shuffle a number of words in a List:

```
import java.util.*;

public class Shuffle {
   public static void main(String[] args) {

    List<String> list = new ArrayList<>();
    list.add("Java");
    list.add("Programming");
```

```
        list.add("Is");
        list.add("Fun");

        System.out.println("Original List: " + list);

        Collections.sort(list);
        System.out.println("Sorted List: " + list);

        Collections.shuffle(list);
        System.out.println("Shuffled List: " + list);

        Collections.shuffle(list);
        System.out.println("Shuffled List: " + list);
    }
}
```

A sample output of the above code is:

```
Original List: [Java, Programming, Is, Fun]
Sorted List: [Fun, Is, Java, Programming]
Shuffled List: [Java, Programming, Fun, Is]
Shuffled List: [Fun, Is, Programming, Java]
```

If you run this code yourself your shuffled list might look different as the shuffle method gives a random permutation of the List. In fact if this program is run again two different shuffled lists should be produced.

Routine Data Manipulation

The Java collections framework provides five straightforward algorithms for routine data manipulation. These algorithms are:

- reverse(List list) - reverses the order of elements in the given List
- fill(List list, Object o) - fills every element in a List with a specified element.
- copy(List destination, List source) - copies the elements of a source List into a destination List. This overwrites the contents of the destination List. The destination List should be as long as the source list, otherwise if it is longer, the remaining elements in the destination List are unaffected.
- swap(List list, int i, int j) - swaps the elements at given positions in a List.
- addAll(Collection c, Object o1, Object o2, ...) - adds all given elements to a Collection.

The following program uses the reverse method to reverse elements in a List.

```
import java.util.*;

public class Reverse {
   public static void main(String[] args) {

    List<String> list = new ArrayList<>();
    list.add("Java");
    list.add("Programming");
    list.add("Is");
    list.add("Fun");

    System.out.println("Original List: " + list);

    Collections.reverse(list);
    System.out.println("Reversed List: " + list);

   }
}
```

The output of the above code is:

```
Original List: [Java, Programming, Is, Fun]
Reversed List: [Fun, Is, Programming, Java]
```

Searching

The Java collections framework contains two static binarySearch() methods. The binarySearch algorithm searches for a given element in a sorted list. As the name of these methods implies, they use the binary search algorithm to perform the search. The binarySearch() methods are:

- int binarySearch(List list, T key)
- int binarySearch(List list, T key, Comparator c)

The binarySearch(List list, T key) method takes a List and an element "key". This element is also known as the search key. The binarySearch(List list, T key) method assumes that the List is sorted in ascending order according to the natural ordering of its elements.

The binarySearch(List list, T key, Comparator c) method takes a Comparator object as well as a List and search key. It assumes that the List is sorted into ascending

order according to the given Comparator. Therefore, before calling binarySearch, the sort algorithm can be used to sort the List.

Both of the binarySearch() methods, returns the index of the element if it is found in the List. If the element is not found in the List a negative value is returned. This value is equal to (-(insertion_index) -1). Math.abs((-(insertion_index) -1)) is the index where we can insert the given key and still keep the list in order.

The following program demonstrates how the binarySearch() method is used:

```
import java.util.*;

public class Reverse {
   public static void main(String[] args) {

      List<String> list = new ArrayList<>();
      list.add("Java");
      list.add("Programming");
      list.add("Is");
      list.add("Fun");

      Collections.sort(list);
      System.out.println("Sorted List: " + list);

      int index = Collections.binarySearch(list, "Java");
      System.out.println("Java in List  is at " + index);

      index = Collections.binarySearch(list, "Python");
      System.out.println("Python in List is  at " + index);

   }
}
```

The output of the above code is:

```
Sorted List: [Fun, Is, Java, Programming]
Java in List  is at 2
Python in List is  at -5
```

Since “Python” is not in the List, the the binary search returns -5. This means that if you insert “Python” into the List, it will be inserted at index 4. This is calculated using (-(insertion_index) - 1) which is (-(-5) -1).

Composition

The collections framework contain two algorithms to test some aspect of the composition of one or more Collections:

- int frequency(Collection c, Object o)
- boolean disjoint(Collection c1, Collection c2)

The frequency(Collection c, Object o) method returns the number of time a specific object occurs in the given collection.

The disjoint(Collection c1, Collection c2) method determines whether the two specified Collections contain no elements in common.

Finding Extreme Values

The collections framework contain two algorithms, min and max, that allow you to find the minimum or maximum element in a Collection. Each of these algorithms come in two forms:

- max(Collection c) - Takes in a Collection and returns the maximum element according to the elements' natural ordering.
- max(Collection c, Comparator comp) - Takes in a Comparator and a Collection and returns the maximum element according to the specified Comparator.
- min(Collection c) - Takes in a Collection and returns the minimum element according to the elements' natural ordering.
- min(Collection c, Comparator comp) - Takes in a Comparator and a Collection and returns the minimum element according to the specified Comparator.

Compulsory Task

Follow these steps:

- Design a class called Course. The class should contain:
  - The data fields courseName (String), numberOfStudents (int) and courseLecturer (String).
  - A constructor that constructs a Course object with the specified courseName, numberOfStudents and courseLecturer.
  - The relevant get and set methods for the data fields.
  - A toString() method that formats that returns a string that represents a course object in the following format: (courseName, courseLecturer, numberOfStudents)

- Create a new ArrayList called courses1, add 5 courses to it and print it out.

- Sort the List according the numberOfStudents and print it out.

- Swap the element at position 1 of the List with the element at position 2 and print it out.

- Create a new ArrayList called courses2.

- Using the addAll method add 5 courses to the courses2 List and print it out.

- Copy all of the courses from courses1 into courses2.

- Add the following two elements to courses2:
  - (Java 101, Dr. P Green, 55)
  - (Advanced Programming, Prof. M Milton, 93)

- Sort the courses in courses2 alphabetically according to the course name and print it out.

- Search for the course “Java 101” in courses2 and print out the index of the course in the List.

- Use the disjoint function to determine whether courses1 and courses2 have any elements in common and print out the result.

- In courses2, find the course with the most students and the course with the least students and print each out.

Rate us

Share your thoughts

Hyperion strives to provide internationally-excellent course content that helps you achieve your learning outcomes.

Think the content of this task, or this course as a whole, can be improved or think we've done a good job?

Click here to share your thoughts anonymously.