Week 7. Collection Framework Java Generics, Stack, PriorityQueue &Comparator

1. Given a string s, regroup the characters of s so that any two adjacent characters are not the same. Return any possible rearrangement of s or return "" if not possible. **Details**:
   1. Input: A string s consisting of lowercase English letters. The length of s is between 1 and 500 characters.
   2. Output: A string where no two adjacent characters are identical. If such a rearrangement is not possible, return an empty string.

# Example:

1. Input:s="aab"

Output:"aba"(oranyothervalidrearrangementlike"baa")

1. Input:s="aaab"

Output: "" (since it's not possible to rearrange the characters without having at least two adjacent as)

# Constraints:

1. Thestringlengthisbetween1and500characters.
2. ThestringconsistsonlyoflowercaseEnglishletters.

# Approach:

* **Character Frequency withLinkedList:** Use a**LinkedList** tostore characters and their frequencies, ensuring you can maintain and update frequencies as needed.
* **PriorityQueue (Min-Heap):** Use the **PriorityQueue** with a **custom comparator** to manage the characters based on their frequencies.
* **Stack for Previous Characters:** Use a **Stack** to track previously placed characters to avoid placing the same character consecutively.

**Class: RearrangeString**

import java.util.\*;

public class RearrangeString {

public String rearrangeString(String s) {

int[] charCount = new int[26];

for (char c : s.toCharArray()) {

charCount[c - 'a']++;

}

PriorityQueue<Character> maxHeap = new PriorityQueue<>((a, b) -> charCount[b - 'a'] - charCount[a - 'a']);

for (char c = 'a'; c <= 'z'; c++) {

if (charCount[c - 'a'] > 0) {

maxHeap.offer(c);

}

}

StringBuilder result = new StringBuilder();

char prevChar = '#';

int prevCount = 0;

while (!maxHeap.isEmpty()) {

char currentChar = maxHeap.poll();

result.append(currentChar);

charCount[currentChar - 'a']--;

if (prevCount > 0) {

maxHeap.offer(prevChar);

}

prevChar = currentChar;

prevCount = charCount[currentChar - 'a'];

}

return result.length() == s.length() ? result.toString() : "";

}

public static void main(String[] args) {

RearrangeString rs = new RearrangeString();

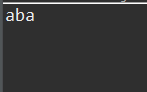
System.out.println(rs.rearrangeString("aab"));

System.out.println(rs.rearrangeString("aaab"));

}

}

**OUTPUT**



# AdvancedTypeBoundswithGenerics

**Objective:** Create a generic class that uses advanced type bounds and wildcards and understand how the generic methods work with type bounds and wildcards.

# Details:

**ClassDefinition:**

* 1. DefineTwithmultiplebounds:itmustimplementbothComparable<T>.

# Methods:

1. **processList(List<? extends T> list)**: This method should process a list of elements that are of type T or its subtypes. It should iterate through the list and print each element.
2. **addToList(List<? super T> list, T element)**: This method should add an element of type T to a list that can hold T or any supertype of T.

# UseCase:

1. Create a Product class that implements Comparable<Product> and Serializable.
2. UseAdvancedGenericwithProductto:
   * ProcessalistofProductobjects.
   * AddanewProducttoanotherlistandprocessit.

**Class: Main**

# import java.io.Serializable;

# import java.util.ArrayList;

# import java.util.List;

# class Product implements Comparable<Product>, Serializable {

# private String name;

# public Product(String name) {

# this.name = name;

# }

# @Override

# public int compareTo(Product other) {

# return this.name.compareTo(other.name);

# }

# @Override

# public String toString() {

# return name;

# }

# }

# class AdvancedGeneric<T extends Comparable<T>> {

# public void processList(List<? extends T> list) {

# for (T element : list) {

# System.out.println(element);

# }

# }

# public void addToList(List<? super T> list, T element) {

# list.add(element);

# }

# }

# public class Main {

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

# AdvancedGeneric<Product> advancedGeneric = new AdvancedGeneric<>();

# List<Product> productList = new ArrayList<>();

# productList.add(new Product("Product A"));

# productList.add(new Product("Product B"));

# advancedGeneric.processList(productList);

# List<Product> anotherList = new ArrayList<>();

# advancedGeneric.addToList(anotherList, new Product("Product C"));

# advancedGeneric.processList(anotherList);

# }

# }

# OUTPUT

# 

1. AclassEmployeethathasthefollowingrequirements:

**Objective**: To Understand the functionality of a custom Comparator for sorting Employee objects by multiple criteria and to ensure correct application of the complex sorting logic.

# ClassDefinition:

* 1. TheEmployeeclasshasthefollowingattributes:
     + name:String,
     + age:Integer,
     + Salary:Double.
  2. Implement the Comparable<Employee> interface in the Employee class to provide a natural ordering based on salary.

# CustomComparatorImplementation:

1. Implement a custom Comparator<Employee> that provides the following functionalities:
   * **PrimarySorting:**Sortemployeesbyageinascendingorder.
   * **Secondary Sorting:** If two employees have the same age, sort them by salary in descending order.
   * **Tertiary Sorting:** If two employees have the same age and salary, sort them by name in alphabetical order.

**Class: Main**

import java.util.\*;

class Employee implements Comparable<Employee> {

String name;

Integer age;

Double salary;

public Employee(String name, Integer age, Double salary) {

this.name = name;

this.age = age;

this.salary = salary;

}

@Override

public int compareTo(Employee other) {

return this.salary.compareTo(other.salary);

}

@Override

public String toString() {

return "Employee{name='" + name + "', age=" + age + ", salary=" + salary + "}";

}

}

class EmployeeComparator implements Comparator<Employee> {

@Override

public int compare(Employee e1, Employee e2) {

int ageComparison = e1.age.compareTo(e2.age);

if (ageComparison != 0) {

return ageComparison;

}

int salaryComparison = e2.salary.compareTo(e1.salary);

if (salaryComparison != 0) {

return salaryComparison;

}

return e1.name.compareTo(e2.name);

}

}

public class Main {

public static void main(String[] args) {

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

employees.add(new Employee("Alice", 30, 70000.0));

employees.add(new Employee("Bob", 25, 80000.0));

employees.add(new Employee("Charlie", 30, 60000.0));

employees.add(new Employee("David", 25, 70000.0));

employees.add(new Employee("Eve", 30, 70000.0));

Collections.sort(employees, new EmployeeComparator());

for (Employee employee : employees) {

System.out.println(employee);

}

}

}

**OUTPUT**

