```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.Scanner;
import java.util.Hashtable;

public class Pretest {
    private static ArrayList<Customer> customerList = new ArrayList<>();
    private static ArrayList<Customer> originalCustomerList = new ArrayList<>(); // Copy of
customerList above.
    private static ArrayList<Stylist> stylistList = new ArrayList<>();
    private static ArrayList<Service> serviceList = new ArrayList<>();
    private static BST customerTree = new BST();

    public static void main(String[] args) {
        // initialise data of each list
        initialiseCustomerList();
        originalCustomerList.addAll(customerList); // Store the initialised data
        initialiseStylistList();
        initializeServiceList();

        for (Customer customer : customerList) {
            customerTree.insert(customer);
        }

        Scanner scanner = new Scanner(System.in);
        boolean running = true;
```

Menu

```
        /*
         * Below, this algorithm is for menu.
         * This menu is to make a user input to execute a specific task.
         * In order to avoid inputting wrong data such as a number which is 11 or more / character
data, this program can inform users of error information and provide a re-enter function.
         */
        while (running){
            // display the menu.
            System.out.println("Menu: ");
            System.out.println("1. Input(customer data)");
            System.out.println("2. Count(allocated customer for each stylist)");
            System.out.println("3. Sort(highest cost first)");
System.out.println("4. Sort(alphabetically all customers by their last name)");
System.out.println("5. Calculate(number of customers and total amount per service)");
System.out.println("6. Search(customer(s) who paid the highest service cost)");
System.out.println("7. Search(customer(s) who paid the lowest service cost)");
System.out.println("8. Search(customer(s) that each stylist has)");
System.out.println("9. Reset the order of the customer list");
System.out.println("10. Finish the application");
System.out.print("Enter the number (1-10): ");

try {
        int choice = scanner.nextInt();
        scanner.nextLine();

        switch (choice) {
                case 1:
                    takeUserInputOfCustomer(scanner);
                    break;
            /*case 1:
                displayCustomerList();
                break; */
            case 2:
                  countAllocatedCustomerList();
                  break;
            case 3:
                  quickSortCustomerList();
                  displayCustomerList();
                  break;
            case 4:
                  quickSortCustomerListByLastName();
                displayCustomerList();
                break;
            case 5:
                  calculateCostOfService();
                break;
            case 6:
                  displayHighestServiceCostCustomers();
                break;
            case 7:
                  displayLowestServiceCostCustomers();
                break;
            case 8:
                searchCustomersByStylist(scanner);
                break;
            case 9:
```

```
                            System.out.println("The order of the list has been reset.");
                            resetCustomerList();
                        displayCustomerList();
                        break;
                    case 10:
                        System.out.println("The application has finished.");
                        running = false;
                        break;
                    default:
                        System.out.println("The number is not valid. Please re-enter.");
                }
            } catch (java.util.InputMismatchException e) {
                System.out.println("Invalid input. Please enter a number between 1 and 10.");
                scanner.next(); // Consume the invalid input
             }
       }
      // End of menu.
    }
```

List

```
    /*
     * Make ArrayList for the Customer list.
     */
    private static void initialiseCustomerList(){
        customerList.add(new Customer(1, "James", "Roberts", "(49) 123-4567", 20.0, "Haircut",
"Georgia",1));
        customerList.add(new Customer(2, "Ali", "Akbar", "(49) 123-4576", 30.0, "Hair colouring",
"Richard",2));
        customerList.add(new Customer(3, "Maria", "Dawson", "(49) 123-4675", 35.0, "Styling",
"Georgia",1));
        customerList.add(new Customer(4, "Emmett", "Miller", "(49) 1235674", 50.0, "Perming",
"Georgia",2));
        customerList.add(new Customer(5, "Lily", "Taylor", "(49) 12-45673", 45.0, "Treatment",
"Richard",1));
        customerList.add(new Customer(6, "Peter", "Harley", "(49) 23-45671", 55.0, "Hair cut,
Styling", "Bill",3));
    }

    /*
     * Display the Customer list.
     */
    private static void displayCustomerList(){
        System.out.printf("%-10s %-15s %-15s %-15s %-15s %-20s %-15s%n",
                          "ID", "First Name", "Last Name", "Phone", "Service Cost(£)", "Services",
"Stylist");
        for (Customer customer : customerList) {
            System.out.printf("%-10d %-15s %-15s %-15s %-15.2f %-20s %-15s%n",
                              customer.getCustomerId(), customer.getFirstName(),
customer.getLastName(),
                              customer.getPhone(), customer.getServiceCost(), customer.getServices(),
customer.getStylistName());
        }
    }

    /*
     * Make ArrayList for the Stylist list.
     */
    private static void initialiseStylistList() {
        stylistList.add(new Stylist(1, "Georgia", "Senior Stylist"));
        stylistList.add(new Stylist(2, "Richard", "Stylist"));
        stylistList.add(new Stylist(3, "Bill", "Stylist"));
    }

    /*
     * Make ArrayList for the Service list.
     */
    private static void initializeServiceList() {
        serviceList.add(new Service(1, "Haircut", 20));
        serviceList.add(new Service(2, "Hair colouring", 30));
        serviceList.add(new Service(3, "Styling", 35));
        serviceList.add(new Service(4, "Perming", 50));
        serviceList.add(new Service(5, "Treatment", 45));
    }


    /*
     * Make service menu for user input.
     */
    private static final String[] AVAILABLE_SERVICES = {
            "Haircut",
            "Hair colouring",
            "Styling",
            "Perming",
            "Treatment"
        };
```

Case1 : Take user input of customer data with validation.

```
   /*
   * Case1 : Take user input of customer data with validation.
   * In order to avoid inputting wrong data such as customerID, stylistName,  this program can
inform users of error information and provide a re-enter function.
   * Encapsulation - To build a secure application [3]
   *      The private attributes in Customer class can be only accessed through using getter or
setter methods.
   *  Modular approach - To revise source code easily [3]
   *      takeUserInputOfCustomer method demonstrates a modular approach.
   *      By isolating this functionality, we can easily modify the source code in only the
takeUserInputOfCustomer method, contributing to without affecting other parts of the program.
   *  Code Reuse - To increase my productivity [3]
   *      The Customer class and its methods are reused throughout the application.
   *      For instance, the quickSortByLastName method in case 4 calls getLastName() in
Customer class, which the method in Customer class is reused.
   */
   private static void takeUserInputOfCustomer(Scanner scanner) {
      int customerId;
      while (true) {
         System.out.print("Enter Customer ID (7-1000): ");
         customerId = scanner.nextInt();
         scanner.nextLine(); // consume the newline
         if (customerId >= 7 && customerId <= 1000) {
            break;
         } else {
            System.out.println("Invalid ID. Please enter a number between 7 and 1000.");
         }
      }

      System.out.print("Enter First Name: ");
      String firstName = scanner.nextLine();
      System.out.print("Enter Last Name: ");
      String lastName = scanner.nextLine();
      System.out.print("Enter Phone: ");
      String phone = scanner.nextLine();

      double serviceCost;
      while (true) {
         System.out.print("Enter Service Cost (£20-300): ");
         serviceCost = scanner.nextDouble();
```

```
            scanner.nextLine(); // consume the newline
            if (serviceCost >= 20 && serviceCost <= 300) {
                break;
            } else {
                System.out.println("Invalid Service Cost. Please enter a number between £20 and
£300.");
            }
        }
        // Display available services
        System.out.println("Available Services:");
        for (String service : AVAILABLE_SERVICES) {
            System.out.println("- " + service);
        }
        System.out.print("Enter Services (comma-separated if multiple): ");
        String services = scanner.nextLine();

        String stylistName;
        int stylistId = 0;
        while (true) {
            System.out.print("Enter Stylist Name (Georgia, Richard, Bill): ");
            stylistName = scanner.nextLine();
            if (stylistName.equalsIgnoreCase("Georgia")) {
                stylistId = 1;
                break;
            } else if (stylistName.equalsIgnoreCase("Richard")) {
                stylistId = 2;
                break;
            } else if (stylistName.equalsIgnoreCase("Bill")) {
                stylistId = 3;
                break;
            } else {
                System.out.println("Invalid Stylist Name. Please enter Georgia, Richard, or Bill.");
            }
        }
        Customer newCustomer = new Customer(customerId, firstName, lastName, phone,
serviceCost, services, stylistName, stylistId);
        customerList.add(newCustomer);
        originalCustomerList.add(newCustomer); // Keep the original list updated
        System.out.println("Customer added successfully!");
    }
    // End of Case 1
```

Case2: Count Allocated Customer for Each Stylist.

```
    /*
     * Case2: Count Allocated Customer for Each Stylist.
     * This algorithm uses a Hash Table.
     * The reason for using the Hash Table is because it provides efficient storage and retrieval
operations, with average time complexity of O(1) for both insertion and lookup.
     * This makes it suitable for counting the number of customers allocated to each stylist quickly
and efficiently.
     *
     * Encapsulation - To build a secure application [3]
     *     The private attributes in Stylist class can be only accessed through using getter or setter
methods.
     * Modular approach - To revise source code easily [3]
     *     countAllocatedCustomerList method demonstrates a modular approach.
     *     If you want to change the way  the values in the list are calculated, you can only modify
this method.
     * Code Reuse - To increase my productivity [3]
     *     The Customer/Stylist class and its methods are reused throughout the application.
     *     The methods such as getStylistID() and getStylistName() are re-used in the
countAllocatedCustomerList method.
     */
    private static void countAllocatedCustomerList() {
        Hashtable<Integer, Integer> stylistCustomerCount = new Hashtable<>();

        for (Customer customer : customerList) {
            int stylistId = customer.getStylistId();
            stylistCustomerCount.put(stylistId, stylistCustomerCount.getOrDefault(stylistId, 0) + 1);
        }

        System.out.printf("%-15s %-20s %-15s%n", "Stylist ID", "Stylist Name", "Number of
Customers");
        for (Stylist stylist : stylistList) {
            int count = stylistCustomerCount.getOrDefault(stylist.getStylistId(), 0);
            System.out.printf("%-15d %-20s %-15d%n", stylist.getStylistId(), stylist.getStylistName(),
count);
        }
    }
      // End of Case 2
```

Case3: Sort Highest Cost First

```
    /*
```

```
     * Case3: Sort Highest Cost First
     * This algorithm is implemented based on a quicksort algorithm.
     * Why I chose a quicksort for the sorting problem.
     * Note: This running time is basically O(n lg n ).
     * Note: The worst case is that the service cost has been already sorted, then running time
will be O(n^2)[1].
     * Note: However, the probability that the service cost on customerList has been already
sorted could be low.
     * Note: This is why I decide to utilise the quicksort.
     * Encapsulation - To build a secure application [3]
     *     The same as Case1
     *     The private attributes in Customer class can be only accessed through using getter or
setter methods.
     * Modular approach - To revise source code easily [3]
     *     The quickSortCustomerList method demonstrates a modular approach.
     *     If you want to change the sorting logic such as ascending order of the service cost, you
can modify only this method.
     * Code Reuse - To increase my productivity [3]
     *     The Customer class and its methods are reused throughout the application.
     *     For instance, the getServiceCost method is used in the quickSortCustomerList method
to compare service costs.
     */
    private static void quickSortCustomerList() {
        quickSort(0, customerList.size() - 1);
    }

    private static void quickSort(int p, int r) {
        if (p < r) {
            int q = partition(p, r);
            quickSort(p, q - 1);
            quickSort(q + 1, r);
        }
    }

    private static int partition(int p, int r) {
        double x = customerList.get(r).getServiceCost(); // pivot selection
        int i = p - 1;
        for (int j = p; j < r; j++) {
            if (customerList.get(j).getServiceCost() > x) { // If aim to implement ascending order, then
"<" should be used.
                i++;
                // exchange customerList[i] with customerList[j]
                Collections.swap(customerList, i, j);
            }
        }
        Collections.swap(customerList, i + 1, r);
        return i + 1;
    }
    // End of case 3
```

Case 4: Sort Alphabetically All Customers by Their Last Name

```
    /*
     * Case 4: Sort Alphabetically All Customers by Their Last Name
     * Below, the methods for quicksort of customer list by last name.
     * Why I choose a quicksort for the sorting problem.
     * Note: This algorithm is implemented based on a quicksort algorithm as the same as Case3.
     * Note: This means that this running time is basically O(n lg n ).
     * Note: The worst case is that the service cost has been already sorted, then runing time will
be O(n^2) [1].
     * Note: However, the probability that the last name on customerList has been already sorted
alphabetically could be low.
     * Note: This is why I decided to utilise the quicksort.
     * Encapsulation - To build a secure application [3]
     *     The same as Case1
     *     The private attributes in Customer class can be only accessed through using getter or
setter methods.
     * Modular approach - To revise source code easily [3]
     *     The quickSortCustomerListByLastName method demonstrates a modular approach.
     *     If you want to change the sorting logic such as ordering in their first name, you can
modify only this method.
     * Code Reuse - To increase my productivity [3]
     *     The Customer class and its methods are reused throughout the application.
     *     For instance, the getLastName method is used in the
quickSortCustomerListByLastName method to compare last names.
     */
    private static void quickSortCustomerListByLastName() {
        quickSortByLastName(0, customerList.size() - 1);
    }
    private static void quickSortByLastName(int p, int r) {
        if (p < r) {
            int q = partitionByLastName(p, r);
            quickSortByLastName(p, q - 1);
            quickSortByLastName(q + 1, r);
        }
    }
    private static int partitionByLastName(int p, int r) {
        String x = customerList.get(r).getLastName(); // pivot selection
        int i = p - 1;
        for (int j = p; j < r; j++) {
            if (customerList.get(j).getLastName().compareTo(x) < 0) { // "compareTo" is used when
we compare two character values. And "<" means that A is first, Z is last.
                i++;
                // Exchange customerList[i] with customerList[j]
                Collections.swap(customerList, i, j);
            }
        }
        Collections.swap(customerList, i + 1, r);
        return i + 1;
    }
     // End of Case 4
```

Case 5：Calculate Number of Customers and Total Amount Per Service

```
    /*
     * Case 5：Calculate Number of Customers and Total Amount Per Service
     * This algorithm is to calculate the total cost per service based on the number of customers.
     * This algorithm is implemented based on a Hash Table.
     * The reason of using a Hash Table is because it provides efficient storage and retrieval
operations, with average time complexity of O(1) for both insertion and lookup.
     * This makes it suitable for aggregating the total cost and counting the number of customers
for each service quickly and efficiently.
     *
     * Encapsulation - To build a secure application [3]
     *     The private attributes in Customer and Service class can be only accessed through
using getter or setter methods.
     * Modular approach - To revise source code easily [3]
     *     The calculateCostOfService method demonstrates a modular approach.
     * Code Reuse - To increase my productivity [3]
     *     The Service and Customer classes and their methods are reused throughout the
application.
     *     For instance, the getServiceName and getPrice methods in the Service class, and
getServices method in the Customer class are used in the calculateCostOfService method.
     */
    private static void calculateCostOfService() {
        Hashtable<String, Integer> serviceCustomerCount = new Hashtable<>();
        Hashtable<String, Double> serviceTotalCost = new Hashtable<>();
        for (Customer customer : customerList) {
            String[] services = customer.getServices().split(", ");

            for (String serviceName : services) {
                serviceCustomerCount.put(serviceName,
serviceCustomerCount.getOrDefault(serviceName, 0) + 1);
                for (Service service : serviceList) {
                    if (service.getServiceName().equals(serviceName)) {
                        serviceTotalCost.put(serviceName, serviceTotalCost.getOrDefault(serviceName,
0.0) + service.getPrice());
                        break;
                    }
                }
            }
        }

        System.out.printf("%-20s %-20s %-20s%n", "Service", "Number of Customers", "Total
Cost(£)");
        for (Service service : serviceList) {
            String serviceName = service.getServiceName();
            int count = serviceCustomerCount.getOrDefault(serviceName, 0);
            double totalCost = serviceTotalCost.getOrDefault(serviceName, 0.0);
            System.out.printf("%-20s %-20d %-20.2f%n", serviceName, count, totalCost);
        }
    }
    // End of Case 5
```

Case 6: Search Customer(s) Who Paid the Highest Service Cost

```
    /*
     * Case 6: Search Customer(s) Who Paid the Highest Service Cost
     * This algorithm is to find maximum data of service cost.
     * This algorithm is implemented based on Binary Search Tree (BST).
     * The reason for this is because its running time is O(h)( i.e., O(lg h)) [2].
     * Therefore, I have judged BST is a much better choice compared to other data structures.
     *
     * Encapsulation - To build a secure application [3]
     *     The same as Case1
     *     The private attributes in Customer class can be only accessed through using getter or
setter methods.
     * Modular approach - To revise source code easily [3]
     *     The displayHighestServiceCostCustomers method demonstrates a modular approach.
     *     If the logic for determining the highest service cost or how to display the customers
needs to be changed, modifications can be made directly within this method without affecting
other parts of the program. [3]
     * Code Reuse - To increase my productivity
     *     The Customer class and its methods are reused throughout the application.
     *     For instance, the getServiceCost method in the Customer class is called in the
displayHighestServiceCostCustomers method to determine the highest service cost.
     */
    private static void displayHighestServiceCostCustomers() {
       if (customerList.isEmpty()) {
          System.out.println("No customers available.");
          return;
       }

       double maxCost = customerTree.treeMaximum(customerTree.root).getServiceCost();

       System.out.printf("%-10s %-15s %-15s %-15s %-15s %-20s %-15s%n",
                     "ID", "First Name", "Last Name", "Phone", "Service Cost(£)", "Services",
"Stylist");
       for (Customer customer : customerList) {
          if (customer.getServiceCost() == maxCost) {
             System.out.printf("%-10d %-15s %-15s %-15s %-15.2f %-20s %-15s%n",
                           customer.getCustomerId(), customer.getFirstName(),
customer.getLastName(),
                           customer.getPhone(), customer.getServiceCost(), customer.getServices(),
customer.getStylistName());
          }
       }
    }
    // End of Case 6
```

Case 7: Search Customer(s) Who Paid the Lowest Service Cost

```
   /*
    * Case 7: Search Customer(s) Who Paid the Lowest Service Cost
    * This algorithm is to find minimum data of service cost.
    * This algorithm is implemented based on Binary Search Tree (BST) as same as the Case6.
    * The reason for this is because its running time is O(h)( i.e., O(lg h)) [2].
    * Therefore, I have judged BST is a much better choice compared to other data structures.
    *
    * This is almost same as Case6.
    */
    private static void displayLowestServiceCostCustomers() {
        if (customerList.isEmpty()) {
            System.out.println("No customers available.");
            return;
        }

        double minCost = customerTree.treeMinimum(customerTree.root).getServiceCost();

        System.out.printf("%-10s %-15s %-15s %-15s %-15s %-20s %-15s%n",
                "ID", "First Name", "Last Name", "Phone", "Service Cost(£)", "Services",
"Stylist");
        for (Customer customer : customerList) {
            if (customer.getServiceCost() == minCost) {
                System.out.printf("%-10d %-15s %-15s %-15s %-15.2f %-20s %-15s%n",
                        customer.getCustomerId(), customer.getFirstName(),
customer.getLastName(),
                        customer.getPhone(), customer.getServiceCost(), customer.getServices(),
customer.getStylistName());
            }
        }
    }

    // End of Case 7
```

Case 8: Search Customer(s) That Each Stylist Has

```
    /*
     * Case 8: Search Customer(s) That Each Stylist Has
     * The algorithm is to input a stylist name and to display the customerList through inputting a
stylist.
     * In order to avoid inputting wrong information such as invalid stylistName, this program can
inform users of error information and provide a re-enter function.
     * Encapsulation - To build a secure application [3]
     *     The same as Case1
     *     The private attributes in Customer class can be only accessed through using getter or
setter methods.
     * Modular approach - To revise source code easily [3]
     *     The searchCustomersByStylist method exemplifies the modular approach.
     *     If you want to change how customers search for a specific stylist or how the results are
displayed, you only need to modify this method.
     * Code Reuse - To increase my productivity [3]
     *     The Customer and Stylist classes and their methods are reused throughout the
application. For instance, in the searchCustomersByStylist method, the getStylistName and
getStylistId methods from the Customer
     */
    private static void searchCustomersByStylist(Scanner scanner) {
            String stylistName;
        while (true) {
            System.out.print("Enter the stylist's name (Georgia, Richard, Bill): ");
            stylistName = scanner.nextLine();
            if (stylistName.equalsIgnoreCase("Georgia") ||
                stylistName.equalsIgnoreCase("Richard") ||
                stylistName.equalsIgnoreCase("Bill")) {
                break;
            } else {
                System.out.println("Invalid stylist name. Please enter Georgia, Richard, or Bill.");
            }
        }

        boolean found = false;
        System.out.printf("%-10s %-15s %-15s %-15s %-15s %-20s %-15s%n",
                          "ID", "First Name", "Last Name", "Phone", "Service Cost(£)", "Services",
"Stylist");
        for (Customer customer : customerList) {
            if (customer.getStylistName().equalsIgnoreCase(stylistName)) {
                System.out.printf("%-10d %-15s %-15s %-15s %-15.2f %-20s %-15s%n",
                                  customer.getCustomerId(), customer.getFirstName(),
customer.getLastName(),
                                  customer.getPhone(), customer.getServiceCost(), customer.getServices(),
customer.getStylistName());
                found = true;
            }
        }

        if (!found) {
            System.out.println("No customers found for stylist, or this stylist does not exist: " +
stylistName);
        }
    }
    // End of Case 8
```

Case 9 Reset the order of the customer list.

```
    /*
```

```
     * Case 9 Reset the order of the customer list.
     */
    private static void resetCustomerList() {
        customerList.clear();
        customerList.addAll(originalCustomerList); // read initialised data
    }
    // End of Case 9


}

/*
 * The class below is for Customer.
 * This class includes four types of methods, such as constructor, getter,
 * setter, and override method.
 */
class Customer {
            private int customerId;
            private String firstName;
            private String lastName;
            private String phone;
            private double serviceCost;
            private String services;
            private String stylistName;
            private int stylistId;


            // Constructor
            public Customer(int customerId, String firstName, String lastName, String phone, double
serviceCost, String services, String stylistName, int stylistId ){
                        this.customerId = customerId;
                        this.firstName = firstName;
                        this.lastName = lastName;
                        this.phone = phone;
                        this.serviceCost = serviceCost;
                        this.services = services;
                        this.stylistName = stylistName;
                        this.stylistId = stylistId;
            }

            /*
     * Start of the getter methods in Customer class.
     */
```

```
        public int getCustomerId(){
                return customerId;
        }

        public String getFirstName(){
                return firstName;
        }

        public String getLastName(){
                return lastName;
        }

        public String getPhone(){
                return phone;
        }

        public double getServiceCost(){
                return serviceCost;
        }

        public String getServices(){
                return services;
        }

        public String getStylistName(){
                return stylistName;
        }

        public int getStylistId(){
                return stylistId;
        }

        /*
     * End of the getter methods in Customer class.
     */

    /*
     * Start of the setter methods in Customer class.
     * Note: stylistId should be set in Stylist Class, not here(Customer class).
     */
    public void setCustomerId(int customerId) {
        this.customerId = customerId;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }

    public void setServiceCost(double serviceCost) {
        this.serviceCost = serviceCost;
    }

    public void setService(String services) {
        this.services = services;
    }

    public void setstylistName(String stylistName) {
        this.stylistName = stylistName;
    }
    /*
     * End of the setter methods in Customer class.
     */

    @Override
    public String toString() {
        return "Customer [ID=" + customerId + ", Name=" + firstName + " " + lastName + ",
Phone=" + phone + ", Service Cost(£)=" + serviceCost + ", Services=" + services + ", Stylist=" +
stylistName + "]";
    }
}


/*
 * The below class is for Stylist.
 * This class includes three types of methods, such as constructor, getters,
 * and setters.
 */

class Stylist {
    private int stylistId;
private String stylistName;
private String title;

public Stylist(int stylistId, String stylistName, String title) {
    this.stylistId = stylistId;
    this.stylistName = stylistName;
    this.title = title;
}

/*
 * Start of the getter methods in Customer class.
 */

public int getStylistId() {
    return stylistId;
}

public String getStylistName() {
    return stylistName;
}

public String getTitle() {
    return title;
}

/*
 * End of the getter methods in Customer class.
 */


/*
 * Start of the setter methods in Customer class.
 */
public void setStylistId(int stylistId) {
    this.stylistId = stylistId;
}

public void setstylistName(String stylistName) {
    this.stylistName = stylistName;
}

public void setTitle(String title) {
   this.title = title;
}
```

```
    /*
     * End of the setter methods in Customer class.
     */
}

// End of the Stylist class

/*
 * Below, Service class is implemented.
 * The class has three types of methods including constructor, getters and setters.
 */

class Service {
    private int serviceId;
    private String serviceName;
    private double price;

    public Service(int serviceId, String serviceName, double price) {
        this.serviceId = serviceId;
        this.serviceName = serviceName;
        this.price = price;
    }

    public int getServiceId() {
            return serviceId;
    }

    public String getServiceName() {
            return serviceName;
    }

    public double getPrice() {
            return price;
    }

    public void setServiceId(int serviceId) {
            this.serviceId = serviceId;
    }

    public void setServiceName(String serviceName) {
            this.serviceName = serviceName;
    }
    public void setPrice(double price) {
            this.price = price;
    }
}

// End of the Service class.

/*
 * The below class is a node class for Binary Search Tree.
 */
class TreeNode {
    Customer data;
    TreeNode left;
    TreeNode right;

    TreeNode(Customer data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

/*
 * The below class is a Binary Search Tree class.
 */
class BST {
    TreeNode root;

    BST() {
        this.root = null;
    }

    void insert(Customer data) {
        root = insertRec(root, data);
    }

    TreeNode insertRec(TreeNode root, Customer data) {
        if (root == null) {
            root = new TreeNode(data);
            return root;
        }
        if (data.getServiceCost() < root.data.getServiceCost()) {
            root.left = insertRec(root.left, data);
        } else if (data.getServiceCost() > root.data.getServiceCost()) {
            root.right = insertRec(root.right, data);
        }
        return root;
    }

    Customer treeMaximum(TreeNode node) {
        while (node.right != null) {
            node = node.right;
        }
        return node.data;
    }

    Customer treeMinimum(TreeNode node) {
        while (node.left != null) {
            node = node.left;
        }
        return node.data;
    }
}
// EOF
```

<References>
[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. Cambridge, MA: MIT Press, 2009, pp170-176.
[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed. Cambridge, MA: MIT Press, 2009, p291.
[3] Q. Charatan and A. Kans, Java in Two Semesters: Featuring JavaFX, 3rd ed. New York, NY: Springer, 2019, p223.