Experiment 1: Program to Insert, Retrieve and Remove Records using TreeSet

Aim

To write a Java program that demonstrates sorting and reversing a **LinkedList**.

Theory

- A **Set** in Java is a **collection** that **does not allow duplicate elements**.
- It is part of the **java.util package** and is defined as an **interface** (java.util.Set).
- Sets are useful when you need a **unique collection of items** (e.g., storing roll numbers, IDs, etc.).

Features of Set

- **No duplicates** allowed.
- **No guaranteed order** of elements (depends on implementation).
- **Can store null** (only one null allowed in most implementations).
- Common implementations:
    - HashSet → Unordered, uses hashing, faster.
    - LinkedHashSet → Maintains insertion order.
    - TreeSet → Maintains **sorted order**.

TreeSet in Java

- TreeSet is a **class** in Java that implements the **NavigableSet interface** and extends **AbstractSet**.
- It stores elements in a **sorted (ascending) order** using a **Red-Black Tree** (a self-balancing BST).

Features of TreeSet

- **No duplicates** allowed.
- **Sorted order** (Natural ordering or custom Comparator).
- **Null not allowed** (will throw NullPointerException if you insert null).
- Performance:
    - add, remove, contains → O(log n)
    - Faster lookup compared to List

Common Methods of TreeSet

- add(E e) → Insert element in sorted order
- remove(Object o) → Remove element
- first() → Retrieve first element
- last() → Retrieve last element
- higher(E e) → Find element greater than given one
- lower(E e) → Find element less than given one
- iterator() → Iterate in ascending order
- descendingIterator() → Iterate in descending order

Conclusion

In this experiment, we learned how to use the **TreeSet** class in Java to store unique elements in sorted order. We successfully performed **insertion, retrieval, and removal** of elements. This demonstrates the efficiency of TreeSet for maintaining a sorted, duplicate-free collection.

Experiment 2: Program to Sort and Reverse LinkedList Elements

Aim

To write a Java program that demonstrates sorting and reversing a **LinkedList**.

Theory

A **linked list** is a **linear data structure** where elements are **stored in nodes**, and each node contains:

1. **Data** (the actual value you store)
2. **A reference (or pointer)** to the **next node** in the list.

Unlike arrays, linked lists **do not store elements in contiguous memory** — instead, they are connected via pointers.

Basic Structure of a Node

In a **singly linked list**:

[Data | Next] → [Data | Next] → [Data | Next] → null

- **Data** → stores the element
- **Next** → reference to the next node
- The **last node** points to null (meaning the list ends)

Types of Linked Lists

1. **Singly Linked List**
    - Each node points to the next node only.
    - Can be traversed in **one direction**.

Example:

10 → 20 → 30 → null

2. **Doubly Linked List**
    - Each node points to **both** the previous and next node.
    - Can be traversed in **both directions**.

Example:

null ← 10 ↔ 20 ↔ 30 → null

3. **Circular Linked List**
    - Last node points back to the **first node**.
    - Can be **singly** or **doubly** circular.

Advantages of Linked Lists

-Dynamic size (can grow/shrink at runtime)
-Easy insertion/deletion (no shifting like arrays)

Disadvantages of Linked Lists

-Uses extra memory for storing pointers
-Slower access (must traverse nodes sequentially)
-No direct access by index (unlike arrays)

Sorting and reversing:

- Collections.sort(list) → sorts in ascending order.
- Collections.reverse(list) → reverses order.

Conclusion

In this experiment, we learned how to manipulate a **LinkedList** using the Collections utility class. By applying **sorting and reversing**, we observed how data can be rearranged dynamically. This demonstrates the flexibility of LinkedList in handling different data operations efficiently.

Experiment 3: Write a program in Java such that it demonstrates the event actions associated with the keyboard.

Aim:

To create a Java program that demonstrates event handling associated with keyboard actions using **AWT Event Listeners**.

Theory:

Event-Driven Programming in Java

In Java, many applications are designed based on the **Event-Driven Programming Model**. An event is an action or occurrence that happens in the system you are programming, which can be triggered by the user (such as pressing a key, clicking a mouse, etc.).

- Examples of events:
    - Mouse clicks
    - Keyboard key presses
    - Window opening or closing

Java uses a **Delegation Event Model**, where the event source (e.g., a TextField or Button) generates an event object, and event listeners respond to those events by executing specific code.

Keyboard Events Overview

A **keyboard event** occurs when the user interacts with the keyboard.

Types of Keyboard Events:

1. **keyPressed(KeyEvent e)**
   Triggered when a key is pressed down.
2. **keyReleased(KeyEvent e)**
   Triggered when a key is released after being pressed.
3. **keyTyped(KeyEvent e)**
   Triggered when a key is typed (i.e., when a key press and release happen and the key corresponds to a character).

Key Components in Keyboard Event Handling

- **KeyEvent Class**
  Represents a keyboard event and contains methods to get information about the key pressed (e.g., key code, character, etc.).

- **KeyListener Interface**
  To handle keyboard events, a class must implement the KeyListener interface which includes the following methods:

  void keyPressed(KeyEvent e);

  void keyReleased(KeyEvent e);

  void keyTyped(KeyEvent e);

- **Event Registration**
  Components like TextField, Frame, etc., can generate keyboard events.
  The program registers an object of a class implementing KeyListener to the component using:

  textField.addKeyListener(this);

How Keyboard Event Handling Works (Step by Step)

1. **User Action:**
   The user presses a key on the keyboard.
2. **Event Generation:**
   The component (like a TextField) generates a KeyEvent object containing information about the key event (which key was pressed, released, or typed).
3. **Event Dispatching:**
   The event is dispatched to all registered event listeners for that component.
4. **Event Handling:**
   The appropriate method in the KeyListener (i.e., keyPressed(), keyReleased(), or keyTyped()) is called, where the programmer defines the desired action.

Example of KeyEvent Methods

Method	**Purpose**
getKeyCode()	Returns an integer code representing the key pressed/released.
getKeyChar()	Returns the character representation of the key typed.
KeyEvent.getKeyText(int keyCode)	Returns a human-readable name of the key.

Conclusion: we successfully demonstrated how Java handles keyboard events using KeyListener. We observed how each method (keyPressed, keyReleased, and keyTyped) reacts to keyboard interactions in real-time.

Experiment 4: Write a program in Java such that it demonstrates the event actions associated with the Mouse.

Aim:

To create a Java program that demonstrates event handling associated with mouse actions using the MouseListener Interface.

Theory:

Event-Driven Programming in Java

- Java applications respond to user-generated events (mouse clicks, key presses).
- Components like **Button**, **Panel**, **Frame** can generate mouse events.
- The **Delegation Event Model** is used:
    - Event Source → Event Object → Event Listener → Event Handling Method.

Mouse Events Overview

The **MouseListener Interface** listens to mouse-related events:

Method	Description
mouseClicked(MouseEvent e)	Invoked when mouse is clicked (pressed and released).
mousePressed(MouseEvent e)	Invoked when a mouse button is pressed.
mouseReleased(MouseEvent e)	Invoked when a mouse button is released.
mouseEntered(MouseEvent e)	Invoked when the mouse enters a component area.
mouseExited(MouseEvent e)	Invoked when the mouse exits a component area.

The MouseEvent Class

The MouseEvent object provides details about the mouse interaction:

- getX() and getY() – Gives the coordinates of the mouse pointer when the event occurred.
- getClickCount() – Number of clicks during the event.
- getButton() – Which mouse button was clicked (left, middle, right).

The MouseListener Interface

To handle mouse events, a class implements the MouseListener interface, which forces us to override all five methods:

public void mouseClicked(MouseEvent e) { ... }

public void mousePressed(MouseEvent e) { ... }

public void mouseReleased(MouseEvent e) { ... }

public void mouseEntered(MouseEvent e) { ... }

public void mouseExited(MouseEvent e) { ... }

Example Use Cases of Mouse Event Handling

- Games (moving characters, shooting, selecting menu items).
- Drawing applications (paintbrush effect).
- Interactive forms (hover effects, tooltips).
- Context menus on right-click.

Conclusion:

we successfully demonstrated how to handle mouse events using the MouseListener interface in Java. We learned to detect and process mouse actions such as click, press, release, enter, and exit in a GUI application.

Experiment 5: Write a program to display "AllTheBest" in 5 different colors on screen using AWT.

Aim:

Write a Java program that displays the text **"AllTheBest"** in 5 different colors on the screen using the **AWT Graphics class**.

Theory:

What is AWT?

- **AWT (Abstract Window Toolkit)** is the **oldest GUI (Graphical User Interface) library in Java**.
- It is part of the **Java Foundation Classes (JFC)** and comes built-in with the Java Development Kit (JDK).
- AWT allows developers to create **windows, buttons, menus, text fields, checkboxes, etc.** for desktop applications.

How AWT Works

- AWT is **platform-dependent** because it uses the **native system's GUI resources** (called **peer classes**).
    - Example: On Windows, AWT components use Windows OS native GUI elements.
    - On Linux, they use Linux system GUI elements.
- This makes AWT programs look slightly different depending on the operating system.

AWT Hierarchy (Important Classes)

The root class of AWT is **java.awt package**.
Some important classes:

1. **Container Classes** (can hold other components)
    - Frame → main window
    - Panel → sub-container inside a frame
    - Dialog → popup window
2. **Component Classes** (user interface controls)
    - Button
    - Label

- TextField
- TextArea
- Checkbox
- Choice (drop-down)
- List

3. **Layout Managers**
    - Define how components are arranged inside a container.
    - Examples: FlowLayout, BorderLayout, GridLayout, CardLayout.
4. **Event Handling Classes**
    - KeyListener, MouseListener, ActionListener etc.
    - Handle user actions like key presses, mouse clicks, button clicks.

Graphics Class in AWT

- The **Graphics** class is used to draw shapes, text, and images on components such as Frame or Panel.
- Common methods:
    - setColor(Color c) → Sets the color of the graphics context.
    - drawString(String str, int x, int y) → Displays the specified text at position (x, y).

Colors in AWT

- Java provides the predefined Color class with many color constants:
    - Color.RED, Color.BLUE, Color.GREEN, Color.ORANGE, Color.MAGENTA, etc.
- We can use these colors to display the text in different colors.

Why Paint Method is Important

- The paint(Graphics g) method is automatically called when the Frame is displayed.
- We override paint(Graphics g) to customize what is displayed on the Frame.

Conclusion:

We learned how to:

- Use the **AWT Frame** and **Graphics class** to display text.
- Set different colors using g.setColor(Color).

- Draw the same text in multiple colors on the screen using g.drawString().

This demonstrates basic graphical programming in Java using AWT.

Experiment 6: Write a java program using swing to create a frame having three text fields. Accept number in first textfield and display previous number in second textfield and next number in the third text field.

Aim:

Create a Java Swing application that accepts a number in the first text field and automatically displays:

- The previous number in the second text field.

Theory:

What is Swing in Java?

- Swing is part of Java Foundation Classes (JFC) used for creating GUI-based applications.
- It provides components like JFrame, JTextField, JButton, JLabel, etc.
- Swing is lightweight, has a consistent appearance across platforms, and is widely used for desktop applications.

Event-Driven Programming

- In Swing, actions like button clicks or text entry generate events.
- These events are handled by implementing ActionListener or other relevant listener interfaces.

Components Used

Component	Purpose
JFrame	Main window of the application.
JTextField	Allows the user to input and display text.
JButton	Executes an action when clicked (optional in this case).
ActionListener	Detects actions (like pressing Enter key in text fields).

Features of Swing

1. Platform-independent → Looks the same on Windows, Linux, or Mac.
2. Lightweight components → Does not rely heavily on OS's native GUI (unlike AWT).

3. Rich set of widgets → Buttons, Labels, Text Fields, Tables, Trees, Tabs, Sliders, Progress Bars, etc.
4. Pluggable Look and Feel (L&F) → You can change how components look (e.g., Windows style, Metal style, Nimbus theme).
5. MVC architecture → Uses Model-View-Controller pattern internally for flexibility.
6. Event Handling → Same as AWT, but more robust.

Conclusion:

we successfully created a Java Swing application that demonstrates:

- How to use **JTextField** for input and output.
- How to handle **ActionListener** events.
- How to perform simple arithmetic operations based on user input.
- How to display results in real time.