Day 13, EMP\_ID: 1112916146

Task1

package HashTables.LinkedList;

import java.util.LinkedList;

public class LinkedListDemo2 {

public static void main(String[] args) {

// Create a Java LinkedList of String

LinkedList<String> flowers = new LinkedList<>();

// Add elements

flowers.add("Rose");

flowers.add("Lily");

flowers.addFirst("Lotus"); // Adds at the START

flowers.addLast("Sunflower"); // Adds at the END

// Get first and last elements

System.out.println("First Element is: " + flowers.getFirst());

System.out.println("Last Element is: " + flowers.getLast());

System.out.println("All flowers: " + flowers);

//remove some specific flowers

flowers.removeFirst(); // removes 1st

flowers.removeLast(); // removes last

for (String flower : flowers) {

System.out.println(flower);

}

}

}



Task2 Try to create a node and add a value to it..

package HashTables.LinkedList;

public class LinkedListDemo {

// STEP 1: Node class

static class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

//Step 2 : Create a Head Pointer

private Node head;

// STEP 3: METHOD TO ADD A NODE AT THE END

public void insertAtEnd(int value) {

Node newNode = new Node(value);

// If the list is empty, make the new node the head

if (head == null) {

head = newNode;

return;

}

// Otherwise, traverse to the end

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

// Link the new node

temp.next = newNode;

}

}

}

Task2

package HashTables.LinkedList;

import java.util.NoSuchElementException;

public class Task3 {

// STEP 1: Node class

static class Node<T> {

T data;

Node<T> next;

public Node(T data) {

this.data = data;

this.next = null;

}

}

// STEP 2: Custom Linked List

static class CustomLinkedList<T> {

private Node<T> head;

private int size = 0;

// Adding data to the end

public void addLast(T data) {

Node<T> newNode = new Node<>(data);

if (head == null) {

head = newNode;

} else {

Node<T> current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

size++;

}

// Adding data at the beginning

public void addFirst(T data) {

Node<T> newNode = new Node<>(data);

newNode.next = head;

head = newNode;

size++;

}

// Remove from the beginning

public T removeFirst() {

if (head == null) {

throw new NoSuchElementException("List is empty");

}

T removedData = head.data;

head = head.next;

size--;

return removedData;

}

// Remove from the end

public T removeLast() {

if (head == null) {

throw new NoSuchElementException("List is empty");

}

if (head.next == null) {

T removedData = head.data;

head = null;

size--;

return removedData;

}

Node<T> current = head;

while (current.next.next != null) {

current = current.next;

}

T removedData = current.next.data;

current.next = null;

size--;

return removedData;

}

// Get the size of the list

public int size() {

return size;

}

// Display the list

public void display() {

Node<T> current = head;

while (current != null) {

System.out.print(current.data + " -> ");

current = current.next;

}

System.out.println("null");

}

// Bounds checking method

private void checkBounds(int index) {

if (index < 0 || index >= size) {

throw new IndexOutOfBoundsException("Index out of bounds");

}

}

}

// MAIN METHOD

public static void main(String[] args) {

CustomLinkedList<String> flowerList = new CustomLinkedList<>();

flowerList.addFirst("Rose");

flowerList.addFirst("Lily");

flowerList.display();

flowerList.addLast("Lotus");

flowerList.addLast("Marigold");

flowerList.display();

// removing flowers

System.out.println("Removed First: " + flowerList.removeFirst());

flowerList.display();

System.out.println("Removed Last: " + flowerList.removeLast());

flowerList.display();

// Final size

System.out.println("Final size of the list: " + flowerList.size());

}

}



=========================================================================

**Home Task 👍**

## **Advantages of a Linked List**

1. **Dynamic Size**
   * The size can **grow or shrink** during program execution, unlike fixed-sized arrays.
2. **Efficient Insertion/Deletion**
   * Inserting or deleting a node doesn’t require shifting elements (as in arrays).
   * You just adjust pointers.
3. **Better Memory Utilization**
   * No pre-allocation required — memory is used as needed.
4. **Ease of Implementation for Certain Structures**
   * Enables easy implementation of **stacks**, **queues**, and other data structures.
5. **No Wasted Space**
   * You only use as much space as you have elements.

## **Disadvantages of a Linked List**

1. **Additional Memory for Pointers**
   * Each node needs extra space for a pointer/reference.
2. **Sequential Access Only**
   * No random access like an array (you must **traverse** from the head).
3. **Higher Time Complexity for Access**
   * To access an element by position, you must walk through the list, making it **O(n)**.
4. **Not Cache Friendly**
   * Elements aren’t stored contiguously, leading to potentially lower cache performance.
5. **More Complex Operations**
   * Managing pointers can be error-prone, leading to bugs like **memory leaks** or **null pointer exceptions**.

Home Task2

Real Life examples /Application of Linked List

1.Music Playlist

2. Web Browser

3.Traffic Navigation

4.**Photo Gallery or Slides**

**Home Task3 :**

### **📥 Insertion**

* **Add at the beginning** (addFirst, push): Insert a new node at the front (updates head).
* **Add at the end** (addLast, add, offer, enqueue): Traverse to the end, then append the new node.
* **Add at a specific position** (add(index, element), insertAt): Traverse to the target index and insert the new node.

### **🗑 Deletion**

* **Remove from the beginning** (removeFirst, pop, dequeue): Update head to head.next, effectively deleting the first node.
* **Remove from the end** (removeLast, pollLast): Traverse to the node before the last, then set its next to null.
* **Remove at a specific position** (remove(index)): Traverse to the node before the target and bypass it.
* **Remove by value** (remove(Object o), removeFirstOccurrence, removeLastOccurrence): Find the node with the given value and delete it.

### **🔍 Retrieval & Search**

* **Get element by index** (get(index), getFirst, getLast): Traverse to the specified index and return its data
* **Contains / indexOf** (contains, indexOf, lastIndexOf): Linear search through list for a value or its position.

### **🔄 Traversal & Display**

* **Iterate through the list** (iterator(), listIterator(), manual traversal): Walk each node from head to null.
* **Descend/Reverse iterator** (descendingIterator(), reverse traversal in doubly/circular lists)

### **ℹ️ Utility**

* **size()** / **isEmpty()**: Return the count of nodes or whether the list has none.
* **clear()**: Remove all elements.
* **clone()**: Create a shallow copy of the list.
* **toArray()**: Convert the list to an array.

### **🔁 Stack & Queue Specific (via LinkedList as Deque)**

* **push** and **pop**: Use the list as a stack (LIFO).
* **offer**, **poll**, **peek**: Use the list as a queue (FIFO).

### **🧮 Algorithmic Operations**

Advanced operations you might implement:

* **reverse(list)**: Invert the order of nodes.
* **findMiddle()**: Use slow/fast pointers to find middle node.
* **sort**: Reorder nodes by value.
* **merge**, **concat**: Combine multiple linked lists.

=====================================================================

Task 5 Wap to create a linked list add 5 elements to it and replace 3 rd element with different value..

package HashTables.LinkedList;

//Wap to create linked list add 5 elements to it and replace 3 rd element with different value..

import java.util.LinkedList;

public class Task5 {

public static void main(String[] args) {

LinkedList<String> list = new LinkedList<>();

// Step 1: Add 5 elements

list.add("One");

list.add("Two");

list.add("Three");

list.add("Four");

list.add("Five");

// Display before replacement

System.out.println("Before: " + list);

// Step 2: Replace 3rd element (index 2)

list.set(2, "NewValue");

// Display after replacement

System.out.println("After: " + list);

}

}



Task 6

package HashTables.LinkedList;

//Remove any element

import java.util.LinkedList;

public class Task6 {

public static void main(String[] args) {

LinkedList<String> list = new LinkedList<>();

// Step 1: Add 5 elements

list.add("One");

list.add("Two");

list.add("Three");

list.add("Four");

list.add("Five");

// Display before removing

System.out.println("Before: " + list);

// Step 2: removing 4th element (index 2)

list.remove(4);

// Display after removing

System.out.println("After: " + list);

}

}



Task7

package HashTables.LinkedList;

import java.util.LinkedList;

public class Task7 {

public static void main(String[] args) {

LinkedList<String> list = new LinkedList<>();

// Step 1 & 2: Add 5 elements

list.add("Apple");

list.add("Banana");

list.add("Cherry");

list.add("Date");

list.add("Mango");

// Display using for loop with get()

System.out.print("Using for-loop with get(): ");

for (int i = 0; i < list.size(); i++) {

System.out.print(list.get(i));

if (i < list.size() - 1) {

System.out.print(", ");

}

}

System.out.println();

// Display using enhanced for-each loop

System.out.print("Using for-each loop: ");

for (String fruit : list) {

System.out.print(fruit);

System.out.print(", ");

}

System.out.println();

}

}



Task 8 (code is correct mistakenly wrote task 7)

package HashTables.LinkedList;

import java.util.LinkedList;

public class Task7 {

public static void main(String[] args) {

LinkedList<String> list = new LinkedList<>();

// Step 1 & 2: Add 5 elements

list.add("Apple");

list.add("Banana");

list.add("Cherry");

list.add("Date");

list.add("Mango");

// Display using for loop with get()

System.out.print("Using for-loop with get(): ");

for (int i = 0; i < list.size(); i++) {

System.out.print(list.get(i));

if (i < list.size() - 1) {

System.out.print(", ");

}

}

System.out.println();

// Display using enhanced for-each loop

System.out.print("Using for-each loop: ");

for (String fruit : list) {

System.out.print(fruit);

System.out.print(", ");

}

System.out.println();

String[] arr = list.toArray(new String[list.size()]);

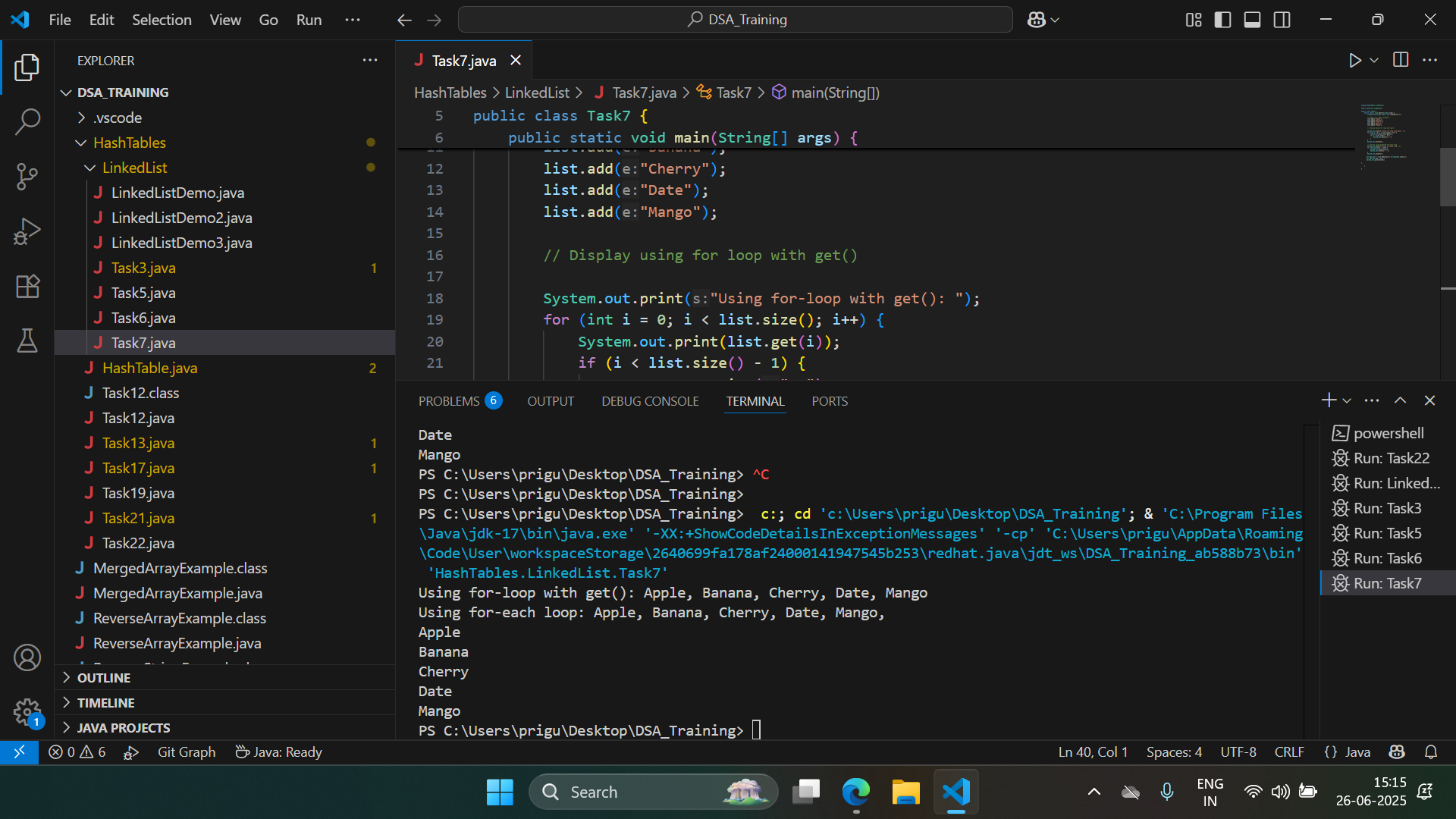
for (String ele : arr) {

System.out.println(ele);

}

}

}



Task9

package HashTables.LinkedList;

import java.util.LinkedList;

public class Task9 {

public static void main(String[] args) {

LinkedList<String> original = new LinkedList<>();

original.add("B");

original.add("B");

original.add("C");

LinkedList<String> cloned = (LinkedList<String>) original.clone();

System.out.println("Original: " + original);

System.out.println("Cloned: " + cloned);

}

}



Task 10

package HashTables.LinkedList;

import java.util.Iterator;

import java.util.LinkedList;

//Create linked list and iterate the values using ListIterator class in util package

public class Task10 {

public static void main(String[] args) {

LinkedList<String> list = new LinkedList<>();

// Step 1 & 2: Add 5 elements

list.add("Apple");

list.add("Banana");

list.add("Cherry");

list.add("Date");

list.add("Mango");

// Display using for loop with get(

System.out.print("Using for-loop with get(): ");

for (int i = 0; i < list.size(); i++) {

System.out.print(list.get(i));

if (i < list.size() - 1) {

System.out.print(", ");

}

}

System.out.println();

// Display using iterator class

Iterator<String> it = list.iterator();

System.out.println(" Using Iterator : ");

while (it.hasNext()) {

String item = it.next();

System.out.print(item + " ");

}

}

}

Task 11

package HashTables.LinkedList;

import java.util.LinkedList;

public class Task11 {

public static void main(String[] args) {

// 1️⃣ Create a LinkedList

LinkedList<String> list = new LinkedList<>();

// 2️⃣ Push elements

list.push("Apple");

list.push("Banana");

list.push("Cherry");

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

// 3️⃣ Pop an element

String popped = list.pop();

System.out.println("Popped element: " + popped);

// Final list:

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

}

}



Task12

Differences Between Iterator and Spliterator:

1. Purpose:

* Iterator is used for going through a collection one element at a time (one by one).
* Spliterator is used for going through a collection one element at a time, and can also split the data into parts for parallel processing.

1. Iteration:

* Iterator goes sequentially, from first to last.
* Spliterator can do sequential iteration and can also divide the data for parallel processing.

1. Splitting:

* Iterator cannot split the data.
* Spliterator can split the data into two or more parts.

1. Use Cases:

* Iterator is used for simple, sequential processing (single-threaded).
* Spliterator is used when dealing with large collections and when parallel processing is needed (like in parallel streams).

1. Methods:

* Iterator has methods like hasNext(), next(), remove().
* Spliterator has methods like tryAdvance(), forEachRemaining(), trySplit(), estimateSize(), characteristics().

Summary:  
 Use an Iterator when you just want to walk through a list one element at a time.  
 Use a Spliterator when you want the option to split the data for parallel processing, making it ideal for working with big collections or using parallel streams.

Task 13 mistakenly wrote Task12

package HashTables.LinkedList;

import java.util.LinkedList;

import java.util.Spliterator;

public class Task12 {

public static void main(String[] args) {

LinkedList<String> lobj = new LinkedList<>();

lobj.add("Prasunamba");

lobj.add("Meher");

lobj.add(".MK");

Spliterator<String> sitobj = lobj.spliterator();

System.out.println("Splitting the list:");

sitobj.forEachRemaining(System.out::println);

}

}



Task13

package HashTables.LinkedList;

import java.util.\*;

public class Task13 {

public static void main(String[] args) {

LinkedList<String> lobj = new LinkedList<>();

lobj.add("Prasunamba");

lobj.add("Meher");

lobj.add(".MK");

lobj.add("Priya");

Spliterator<String> sitobj = lobj.spliterator();

System.out.println("Splitting the list:");

sitobj.forEachRemaining(System.out::println);

}

}



Task14

package HashTables.LinkedList;

import java.util.LinkedList;

import java.util.Spliterator;

public class Task14 {

public static void main(String[] args) {

// 1️⃣ Create a LinkedList

LinkedList<String> list = new LinkedList<>();

list.add("A");

list.add("B");

list.add("C");

list.add("D");

// 2️⃣ Get the first spliterator

Spliterator<String> itobj1 = list.spliterator();

// 3️⃣ Split it into two

Spliterator<String> itobj2 = itobj1.trySplit();

// 4️⃣ Print the first part (itobj2) using forEachRemaining

System.out.println("First Part:");

itobj2.forEachRemaining(System.out::println);

// 5️⃣ Print the second part (itobj1) using tryAdvance

System.out.println("\nSecond Part:");

while (itobj1.tryAdvance((n) -> {

System.out.println(n);

})) {

// Keep looping until no more elements

}

}

}

Task15

A **pointer** is a variable that stores the memory address of another variable. Instead of holding a direct value, it holds the location in memory where the value is stored. This allows for efficient memory management and manipulation of data structures.

### **Why Pointers Matter (Simply)**

* They let programs work with data **without copying it**, which is faster and saves memory.
* They enable advanced structures like **linked lists** and **trees**, as well as features like passing data into functions by reference so the function can modify it directly.

Task17

### **What does & do?**

* **& is the "address-of" operator.** It gives you the memory address of a variable.

### **What does \* do?**

1. **In declarations**, it specifies that a variable is a pointer type:

Use **&** when you want a variable's **memory address**.

Use **\*** to **declare** a pointer **or** to **dereference** a pointer to get/set its value.

Task 18 (done in online compiler)

#include <iostream>

using namespace std;

int main() {

int x = 10;

int\* p = &x; // p is a pointer to x

cout << "x = " << x << "\n";

cout << "p (address of x) = " << p << "\n";

cout << "\*p (value at that address) = " << \*p << "\n";

\*p = 50; // change x through the pointer

cout << "Now x = " << x << "\n";

return 0;

}

