# **Day 13 - 28 June 2025**

**Document Name:**Day 13 - hmuvvala@ - Hari Gopal Muvvala

### Task 1: Create a node with any data type (object-based)

#### Program:

public class Task1\_CreateNode {

static class Node {

Object data;

Node next;

Node(Object data) {

this.data = data;

this.next = null;

}

}

public static void main(String[] args) {

Node node1 = new Node("Hari");

System.out.println("Node created with data: " + node1.data);

}

}

#### **Output:**

Node created with data: Hari

#### **Notes:**

This is the most basic representation of a node in a linked list. I used Object as the data type so that I can store any kind of value (String, int, etc.). This is the foundation of understanding how nodes are created before building full linked lists. I am not yet linking multiple nodes here; just learning how to define a node and set its data.

### Task 2: Add element at end of the list

#### Program:

public class Task2\_LinkedListAdd<T> {

static class Node<T> {

T data;

Node<T> next;

public Node(T data) {

this.data = data;

this.next = null;

}

}

Node<T> head = null;

public void add(T data) {

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

if (head == null) {

head = newNode;

} else {

Node<T> temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

public void display() {

Node<T> current = head;

while (current != null) {

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

current = current.next;

}

}

public static void main(String[] args) {

Task2\_LinkedListAdd<String> list = new Task2\_LinkedListAdd<>();

list.add("Hari");

list.add("Divya");

list.display();

}

}

#### Output:

Hari Divya

#### Notes:

This program shows how to add elements to the end of a singly linked list and display them.

We use generics so it can hold any data type.

### Task 3: Remove a node (first occurrence)

#### Program:

public class Task3\_RemoveNode<T> {

static class Node<T> {

T data;

Node<T> next;

public Node(T data) {

this.data = data;

this.next = null;

}

}

Node<T> head = null;

public void add(T data) {

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

if (head == null) {

head = newNode;

} else {

Node<T> temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

public void remove(T data) {

if (head == null) return;

if (head.data.equals(data)) {

head = head.next;

return;

}

Node<T> current = head;

while (current.next != null) {

if (current.next.data.equals(data)) {

current.next = current.next.next;

return;

}

current = current.next;

}

}

public void display() {

Node<T> current = head;

while (current != null) {

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

current = current.next;

}

}

public static void main(String[] args) {

Task3\_RemoveNode<String> list = new Task3\_RemoveNode<>();

list.add("Hari");

list.add("Divya");

list.add("Meher");

list.remove("Divya");

list.display();

}

}

#### Output:

Hari Meher

#### Notes:

This program removes the **first occurrence** of a given value.

We check if the head needs to be removed, then traverse to find and skip the node to remove.

### Task 4: Find size of the linked list

#### Program:

public class Task4\_LinkedListSize<T> {

static class Node<T> {

T data;

Node<T> next;

Node(T data) {

this.data = data;

this.next = null;

}

}

Node<T> head = null;

public void add(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;

}

}

public int size() {

int count = 0;

Node<T> current = head;

while (current != null) {

count++;

current = current.next;

}

return count;

}

public static void main(String[] args) {

Task4\_LinkedListSize<String> list = new Task4\_LinkedListSize<>();

list.add("Hari");

list.add("Divya");

list.add("Meher");

System.out.println("Size of list: " + list.size());

}

}

#### Output:

Size of list: 3

#### Notes:

This program counts the number of nodes in a linked list using a simple loop.

Each node visited increases the count.

### Task 5: List all methods of LinkedList (predefined)

#### Program:

import java.util.LinkedList;

public class Task5\_ListMethods {

public static void main(String[] args) {

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

list.add("Hari");

list.add("Divya");

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

System.out.println("First: " + list.getFirst());

System.out.println("Last: " + list.getLast());

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

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

list.addFirst("NewFirst");

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

System.out.println("Contains Divya? " + list.contains("Divya"));

System.out.println("Index of Divya: " + list.indexOf("Divya"));

}

}

#### Output:

List: [Hari, Divya]

First: Hari

Last: Divya

Removed: Hari

After removal: [Divya]

Updated List: [NewFirst, Divya]

Contains Divya? true

Index of Divya: 1

#### Notes:

This program uses predefined LinkedList methods like addFirst, getFirst, removeFirst, contains, etc.

Trainer asked us to explore these as part of understanding the built-in API.

### Task 6: Get first and last elements, and display all

#### Program:

import java.util.LinkedList;

public class Task6\_GetFirstLast\_Display {

public static void main(String[] args) {

LinkedList<Object> items = new LinkedList<>();

items.add("Hari");

items.add(123);

items.add(45.6);

System.out.println("All Elements:");

for (Object item : items) {

System.out.println(item);

}

System.out.println("First: " + items.getFirst());

System.out.println("Last: " + items.getLast());

}

}

#### Output:

All Elements:

Hari

123

45.6

First: Hari

Last: 45.6

#### Notes:

This task uses built-in LinkedList methods like getFirst() and getLast() to access the ends.

Using Object allows us to store multiple types like String, Integer, Double, etc.

### Task 7: Remove first and last element, then display

#### Program:

import java.util.LinkedList;

public class Task7\_RemoveFirstLast {

public static void main(String[] args) {

LinkedList<Object> items = new LinkedList<>();

items.add("Hari");

items.add(123);

items.add(45.6);

items.removeFirst();

items.removeLast();

System.out.println("After removing first and last:");

for (Object item : items) {

System.out.println(item);

}

}

}

#### Output:

After removing first and last:

123

#### Notes:

The removeFirst() and removeLast() methods are used to modify the ends of the list.

For loops or for-each can be used to print remaining elements.

### Task 8: Update first element

#### Program:

import java.util.LinkedList;

public class Task8\_UpdateFirst {

public static void main(String[] args) {

LinkedList<Object> items = new LinkedList<>();

items.add("OldValue");

items.add("Second");

items.set(0, "UpdatedValue");

System.out.println("After updating first element:");

for (Object item : items) {

System.out.println(item);

}

}

}

#### Output:

After updating first element:

UpdatedValue

Second

#### Notes:

We used set(index, value) to update the first element in the list.

Index 0 refers to the first element.

### Task 9: Display using get (for loop) and for-each loop

#### Program:

import java.util.LinkedList;

public class Task9\_DisplayBothWays {

public static void main(String[] args) {

LinkedList<Object> items = new LinkedList<>();

items.add("Hari");

items.add("Divya");

items.add("Meher");

System.out.println("Using get(index) with for loop:");

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

System.out.println(items.get(i));

}

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

for (Object item : items) {

System.out.println(item);

}

}

}

#### Output:

Using get(index) with for loop:

Hari

Divya

Meher

Using for-each loop:

Hari

Divya

Meher

#### Notes:

We demonstrated two ways to iterate over the list –

1. for loop with get(i)
2. for-each loop directly on list.

### Task 10: Display list without using loops

#### Program:

import java.util.LinkedList;

public class Task10\_DisplayWithoutLoop {

public static void main(String[] args) {

LinkedList<Object> items = new LinkedList<>();

items.add("Hari");

items.add("Divya");

System.out.println(items); // directly prints in square bracket format

}

}

#### Output:

[Hari, Divya]

#### Notes:

By directly passing the list object to System.out.println(),

Java internally uses toString() method to print the list contents without using any loop.

### Task 11: Convert LinkedList to Array and display

#### Program:

import java.util.LinkedList;

import java.util.Arrays;

public class Task11\_ConvertToArray {

public static void main(String[] args) {

LinkedList<Object> items = new LinkedList<>();

items.add("Hari");

items.add(123);

items.add(45.6);

Object[] array = items.toArray();

System.out.println("Array version: " + Arrays.toString(array));

}

}

#### Output:

Array version: [Hari, 123, 45.6]

#### Notes:

The toArray() method converts the LinkedList into an array.

We use Arrays.toString() to print the array contents in a readable format.

### Task 12: Clone the LinkedList

#### Program:

import java.util.LinkedList;

public class Task12\_CloneLinkedList {

public static void main(String[] args) {

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

original.add("Hari");

original.add("Divya");

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

System.out.println("Cloned list: " + clone);

}

}

#### Output:

Cloned list: [Hari, Divya]

#### Notes:

Cloning creates a shallow copy of the LinkedList.

Use clone() method and cast it back to the appropriate type.

### Task 13: Demonstrate push and pop on LinkedList

#### Program:

import java.util.LinkedList;

public class Task13\_PushPop {

public static void main(String[] args) {

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

stack.push("First");

stack.push("Second");

stack.push("Third");

System.out.println("After pushing: " + stack);

String popped = stack.pop();

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

System.out.println("Remaining: " + stack);

}

}

#### Output:

After pushing: [Third, Second, First]

Popped: Third

Remaining: [Second, First]

#### Notes:

LinkedList can be used as a stack using push() and pop() methods.

Push adds to front, Pop removes from front (LIFO).

### Task 14: Spliterator example using forEachRemaining

#### Program:

import java.util.LinkedList;

import java.util.Spliterator;

public class Task14\_SpliteratorExample {

public static void main(String[] args) {

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

names.add("Hari");

names.add("Divya");

names.add("Meher");

Spliterator<String> sp = names.spliterator();

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

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

}

}

#### Output:

Using Spliterator:

Hari

Divya

Meher

#### Notes:

Spliterator can be used to traverse collections.

forEachRemaining() applies the action to remaining elements.

### Task 15: Split list into two Spliterators using trySplit()

#### Program:

import java.util.LinkedList;

import java.util.Spliterator;

public class Task15\_SpliteratorSplit {

public static void main(String[] args) {

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

data.add("Hari");

data.add("Divya");

data.add("Meher");

data.add("MK");

Spliterator<String> s1 = data.spliterator();

Spliterator<String> s2 = s1.trySplit();

System.out.println("Spliterator 1:");

while (s1.tryAdvance(System.out::println));

System.out.println("Spliterator 2:");

while (s2.tryAdvance(System.out::println));

}

}

#### Output:

Spliterator 1:

Hari

Divya

Spliterator 2:

Meher

MK

#### Notes:

trySplit() divides the Spliterator into two parts for potential parallel processing.

tryAdvance() processes elements one at a time.

### Task 16: Create a Doubly Linked List

#### Program:

public class Task16\_DoublyLinkedList {

static class Node {

int data;

Node prev, next;

Node(int data) {

this.data = data;

}

}

Node head = null, tail = null;

void add(int data) {

Node newNode = new Node(data);

if (head == null) {

head = tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

}

void displayForward() {

Node current = head;

while (current != null) {

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

current = current.next;

}

}

void displayBackward() {

Node current = tail;

while (current != null) {

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

current = current.prev;

}

}

public static void main(String[] args) {

Task16\_DoublyLinkedList list = new Task16\_DoublyLinkedList();

list.add(10);

list.add(20);

list.add(30);

System.out.print("Forward: ");

list.displayForward();

System.out.print("\nBackward: ");

list.displayBackward();

}

}

#### Output:

Forward: 10 20 30

Backward: 30 20 10

#### Notes:

A doubly linked list stores links to both previous and next nodes. This allows traversal in both directions. It’s helpful when we want efficient backward movement through data.

### Task 17: Copy data from one map to another map

#### Program:

import java.util.HashMap;

public class Task17\_CopyMap {

public static void main(String[] args) {

HashMap<String, Integer> original = new HashMap<>();

original.put("Hari", 10);

original.put("Divya", 20);

HashMap<String, Integer> copy = new HashMap<>(original);

System.out.println("Copied Map: " + copy);

}

}

#### Output:

Copied Map: {Hari=10, Divya=20}

#### Notes:

We can copy the contents of one map into another using the constructor. It helps quickly duplicate data and create backups or new modified versions.

### Task 18: Create HashMap with capacity and load factor

#### Program:

import java.util.HashMap;

public class Task18\_HashMapWithLoadFactor {

public static void main(String[] args) {

HashMap<String, Integer> hm = new HashMap<>(10, 0.75f);

hm.put("A", 1);

hm.put("B", 2);

System.out.println(hm);

}

}

#### Output:

Forward: 10 20 30

Backward: 30 20 10

#### Notes:

We can specify initial capacity and load factor while creating a HashMap. Load factor controls how full the map can get before it resizes (default is 0.75). Capacity is the initial number of buckets.

### Task 19: Demonstrate rehashing using capacity and load factor

#### Program:

import java.util.HashMap;

public class Task19\_RehashingExample {

public static void main(String[] args) {

HashMap<Integer, String> map = new HashMap<>(4, 0.75f);

map.put(1, "A");

map.put(2, "B");

map.put(3, "C");

map.put(4, "D");

map.put(5, "E"); // triggers rehashing internally

System.out.println(map);

}

}

#### Output:

{1=A, 2=B, 3=C, 4=D, 5=E}

#### Notes:

When the number of entries exceeds capacity × load factor, the map resizes automatically. Here, rehashing happens after 3 entries (4 × 0.75 = 3), so on the 4th insert, internal rehashing happens.

### Task 20: Understand different ways to create HashMap

#### Program:

import java.util.HashMap;

public class Task20\_HashMapCreateMethods {

public static void main(String[] args) {

HashMap<String, Integer> hm1 = new HashMap<>(); // default

HashMap<String, Integer> hm2 = new HashMap<>(10); // with capacity

HashMap<String, Integer> hm3 = new HashMap<>(hm2); // copy

HashMap<String, Integer> hm4 = new HashMap<>(10, 0.75f); // capacity + load factor

System.out.println("HashMaps created using different constructors");

}

}

#### Output:

HashMaps created using different constructors

#### Notes:

This program shows the 4 ways to create HashMaps — useful when we need performance tuning, cloning maps, or preventing resizing.

### Task 21: Create a Circular Linked List and display elements

#### Program:

public class Task21\_CircularLinkedList {

static class Node {

int data;

Node next;

Node(int data) {

this.data = data;

}

}

Node head = null, tail = null;

void add(int data) {

Node newNode = new Node(data);

if (head == null) {

head = tail = newNode;

tail.next = head;

} else {

tail.next = newNode;

tail = newNode;

tail.next = head;

}

}

void display() {

if (head == null) return;

Node current = head;

do {

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

current = current.next;

} while (current != head);

}

public static void main(String[] args) {

Task21\_CircularLinkedList list = new Task21\_CircularLinkedList();

list.add(10);

list.add(20);

list.add(30);

list.display();

}

}

#### Output:

10 20 30

#### Notes:

Circular linked list means the last node links back to the first node (head). This allows looping endlessly. A do-while loop is used to ensure the head is printed even if it’s the only node.

### Home Tasks

#### Advantages of Linked List:

* Dynamic size: Can grow or shrink at runtime.
* Efficient insertion/deletion: Easier than arrays, especially at the beginning or middle.
* No memory wastage: Allocated as needed (no fixed size).
* Useful for implementing other data structures like stacks, queues, graphs, etc.

#### Disadvantages of Linked List:

* Random access not allowed: Need to traverse from the beginning.
* Extra memory for pointers: Each node stores data and a reference.
* More complex to implement than arrays.

#### Applications of Linked List:

* Implementing stacks, queues, hash tables, and graphs.
* Dynamic memory allocation.
* Undo functionality in editors.
* Navigating browser history or playlists.

### Summary Notes (What I learned today):

* How to build a **custom singly and doubly linked list** from scratch.
* Concept of **generic nodes** to accept any data type.
* Used **LinkedList class** from Java’s util package with methods like add, getFirst, getLast, removeFirst, removeLast, push, pop, set, toArray, clone, etc.
* Explored **Spliterator**, forEachRemaining(), trySplit() and tryAdvance() methods.
* Understood how **HashMap** works, and what capacity and load factor mean.
* Learned how **rehashing** happens and how to avoid it with proper settings.
* Created and traversed a **circular linked list**.

These concepts helped me understand how data is connected and accessed internally, and how Java supports both custom and built-in data structures.