Day 13 - 28th June 2025

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ **Pointers in c or c++**  @@@@@@@@@@@@@@@

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

**Task 01:**

Pointers : 10.07 to 10.12

**Answer:**

#include<stdio.h> //-- c lang

// #include<iostream.h> // -- c++

int method1(){

// declaration

int age; // variable

int \*ptr;// pointer variable

// assigning

age = 10;

ptr = &age;

printf("value of age is %d\n", age); // 10

printf("ptr is pointing to %d\n ", \*ptr); // 10

printf("address of age %p\n ", &age);// 145245

printf("value of ptr %p\n ", ptr);//same as above 145245

printf("value of &ptr %p\n ",&ptr); // ptr address --- 454578

return 0;

}

int main() {

method1();

return 0;

}

/\*

// For C++

#include<iostream>

using namespace std;

int method1(){

int age = 10;

int \*ptr = &age;

cout<<"value of age "<<age<<endl;

cout<<"ptr is pointing to "<<\*ptr<<endl;

cout<<"address of age "<<&age<<endl;

cout<<"value of ptr "<<ptr<<endl;

cout<<"value of &ptr "<<&ptr<<endl;

return 0;

}

\*/

**Task 02:**

Linked list in c++

**Answer:**

#include <bits/stdc++.h>

using namespace std;

// Define a Node class

class Node{

public:

int data; // Data part of the node

Node\* next; // Pointer to the next node

// Constructor for convenience

Node(int value) : data(value), next(nullptr) {}

};

// Class for singly linked list

class LinkedList{ // Fixed class name

private:

Node\* head; // Pointer to the head of the list

public:

// Constructor to initialize an empty list

LinkedList(){

head = nullptr;

}

// Function to insert a node at the end

void insertAtEnd(int value){

Node\* newNode = new Node(value);

if(head == nullptr){

head = newNode; // If list is empty, make newNode the head

}

else{

Node\* temp = head;

while (temp->next != nullptr){

temp = temp->next; // Traverse to the last node

}

temp->next = newNode; // Link the last node to newNode

}

}

// Function to delete a Node by Value

void deleteByValue(int value){

if(head == nullptr){

return;

}

if(head->data == value){

Node\* temp = head;

head = head->next; // Move head to the next node

delete temp; // Free memory of the deleted node

return;

}

Node\* temp = head;

while(temp->next && temp->next->data != value){

temp = temp->next; // Traverse to find the node to delete

}

if(temp->next){

Node\* nodeToDelete = temp->next;

temp->next = temp->next->next; // Unlink the node

delete nodeToDelete; //Free Memory

}

}

// Function to display the list

void display(){

Node\* temp = head;

while(temp != nullptr){

cout << temp->data << "->";

temp = temp->next;

}

cout << "NULL" <<endl;

}

// Destructor to free all allocated memory

~LinkedList() {

Node\* temp;

while (head) {

temp = head;

head = head->next;

delete temp;

}

}

};

int main() {

LinkedList list;

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

cout << "Linked List: ";

list.display();

list.deleteByValue(20);

cout << "After Deleting 20: ";

list.display();

return 0;

}

/\* Output:

Linked List: 10->20->30->NULL

After Deleting 20: 10->30->NULL

\*/

**Explanation:**

1. **Node Class**: Contains data and pointer to next node
2. **LinkedList Class**: Manages the list with head pointer
3. **insertAtEnd()**: Adds new node at the end of list
4. **deleteByValue()**: Removes node with specific value
5. **display()**: Shows all elements in the list
6. **Destructor**: Frees memory to prevent memory leaks

**Task 03:**

Use the above code to create a Java code which creates a linked list.

**Answer:**

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

class LinkedList {

private Node head;

public LinkedList() {

this.head = null;

}

public void insertAtEnd(int value) {

Node newNode = new Node(value);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

public void deleteByValue(int value) {

if (head == null) {

return;

}

if (head.data == value) {

head = head.next;

return;

}

Node temp = head;

while (temp.next != null && temp.next.data != value) {

temp = temp.next;

}

if (temp.next != null) {

temp.next = temp.next.next;

}

}

public void display() {

Node temp = head;

while (temp != null) {

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

temp = temp.next;

}

System.out.println("NULL");

}

}

public class Task03\_LinkedListJava {

public static void main(String[] args) {

LinkedList list = new LinkedList();

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

System.out.print("Linked List: ");

list.display();

list.deleteByValue(20);

System.out.print("After Deleting 20: ");

list.display();

}

}

**Task 04:**

Create a node and add a value to it. Which can take any kind of data in the Node.

**Answer:**

import java.util.NoSuchElementException;

class Node<T> {

T data;

Node<T> next;

public Node(T data) {

this.data = data;

this.next = null;

}

}

public class CustomLinkedList<T> {

private Node<T> head;

private int size = 0;

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;

}

size++;

}

public void addFirst(T data) {

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

newNode.next = head;

head = newNode;

size++;

}

public T removeFirst() {

if (head == null) {

throw new NoSuchElementException("List is empty");

}

T removedData = head.data;

head = head.next;

size--;

return removedData;

}

public T get(int index) {

checkBounds(index);

Node<T> current = head;

for (int i = 0; i < index; i++) {

current = current.next;

}

return current.data;

}

public int size() {

return size;

}

public void display() {

Node<T> temp = head;

while (temp != null) {

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

temp = temp.next;

}

System.out.println("NULL");

}

private void checkBounds(int index) {

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

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

}

}

public static void main(String[] args) {

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

liobj.add("Anitha");

liobj.add("Verma");

liobj.addFirst("Jack");

System.out.println("Linked List:");

liobj.display();

System.out.println("First Element: " + liobj.get(0));

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

liobj.removeFirst();

System.out.println("First Element after removal: " + liobj.get(0));

System.out.println("Size after removal: " + liobj.size());

// Test with different data types

CustomLinkedList<Integer> intList = new CustomLinkedList<>();

intList.add(10);

intList.add(20);

intList.add(30);

System.out.println("Integer List:");

intList.display();

}

}

**Task 05:**

List down all the methods of Linked list

**Answer:**

**Java LinkedList Methods:**

**Adding Elements:**

1. add(E e) - Appends element to end of list
2. add(int index, E element) - Inserts element at specified position
3. addFirst(E e) - Inserts element at beginning
4. addLast(E e) - Appends element to end
5. addAll(Collection<? extends E> c) - Adds all elements from collection
6. addAll(int index, Collection<? extends E> c) - Inserts all elements at position

**Removing Elements:**

1. remove() - Removes and returns first element
2. remove(int index) - Removes element at specified position
3. remove(Object o) - Removes first occurrence of specified element
4. removeFirst() - Removes and returns first element
5. removeLast() - Removes and returns last element
6. removeFirstOccurrence(Object o) - Removes first occurrence
7. removeLastOccurrence(Object o) - Removes last occurrence
8. clear() - Removes all elements

**Accessing Elements:**

1. get(int index) - Returns element at specified position
2. getFirst() - Returns first element
3. getLast() - Returns last element
4. peek() - Retrieves but doesn't remove first element
5. peekFirst() - Retrieves first element
6. peekLast() - Retrieves last element

**Stack Operations:**

1. push(E e) - Pushes element onto stack (adds to front)
2. pop() - Pops element from stack (removes from front)

**Queue Operations:**

1. offer(E e) - Adds element as tail
2. poll() - Retrieves and removes head
3. pollFirst() - Retrieves and removes first element
4. pollLast() - Retrieves and removes last element
5. element() - Retrieves but doesn't remove head

**Other Methods:**

1. set(int index, E element) - Replaces element at position
2. size() - Returns number of elements
3. isEmpty() - Checks if list is empty
4. contains(Object o) - Checks if list contains element
5. indexOf(Object o) - Returns index of first occurrence
6. lastIndexOf(Object o) - Returns index of last occurrence
7. toArray() - Returns array containing all elements
8. clone() - Returns shallow copy of LinkedList
9. iterator() - Returns iterator over elements
10. listIterator() - Returns list iterator
11. descendingIterator() - Returns iterator in reverse order
12. spliterator() - Creates spliterator

**Task 06:**

Create linked list using Pre defined class and add elements to it.

**Answer:**

import java.util.LinkedList;

public class Task06\_PreDefinedLinkedList {

public static void main(String[] args) {

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

// Adding elements

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

fruits.addFirst("Mango");

fruits.addLast("Grapes");

System.out.println("Fruits LinkedList: " + fruits);

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

}

}

/\* Output:

Fruits LinkedList: [Mango, Apple, Banana, Orange, Grapes]

Size: 5

\*/

**Task 07:**

Remove first and remove last element and display all elements in the linked list

**Answer:**

import java.util.LinkedList;

public class Task07\_RemoveElements {

public static void main(String[] args) {

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

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

fruits.add("Mango");

fruits.add("Grapes");

System.out.println("Original List: " + fruits);

// Remove first element

String removedFirst = fruits.removeFirst();

System.out.println("Removed first element: " + removedFirst);

System.out.println("After removing first: " + fruits);

// Remove last element

String removedLast = fruits.removeLast();

System.out.println("Removed last element: " + removedLast);

System.out.println("After removing last: " + fruits);

System.out.println("Final size: " + fruits.size());

}

}

/\* Output:

Original List: [Apple, Banana, Orange, Mango, Grapes]

Removed first element: Apple

After removing first: [Banana, Orange, Mango, Grapes]

Removed last element: Grapes

After removing last: [Banana, Orange, Mango]

Final size: 3

\*/

**Task 8:**

In the list update the 1st element to a new value

**Answer:**

import java.util.LinkedList;

public class Task08\_UpdateElement {

public static void main(String[] args) {

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

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

System.out.println("Original List: " + fruits);

System.out.println("Element at index 1: " + fruits.get(1));

// Update element at index 1

fruits.set(1, "Strawberry");

System.out.println("After updating index 1: " + fruits);

System.out.println("New element at index 1: " + fruits.get(1));

}

}

/\* Output:

Original List: [Apple, Banana, Orange]

Element at index 1: Banana

After updating index 1: [Apple, Strawberry, Orange]

New element at index 1: Strawberry

\*/

**Task 9:**

Display the list twice 1..... with get method in for loop and 2 ... for each loop

**Answer:**

import java.util.LinkedList;

public class Task09\_DisplayTwoWays {

public static void main(String[] args) {

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

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

fruits.add("Mango");

System.out.println("Method 1: Using get method with for loop:");

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

System.out.println("Index " + i + ": " + fruits.get(i));

}

System.out.println("\nMethod 2: Using for-each loop:");

for (String fruit : fruits) {

System.out.println("Fruit: " + fruit);

}

}

}

/\* Output:

Method 1: Using get method with for loop:

Index 0: Apple

Index 1: Banana

Index 2: Orange

Index 3: Mango

Method 2: Using for-each loop:

Fruit: Apple

Fruit: Banana

Fruit: Orange

Fruit: Mango

\*/

**Task 10:**

Display the elements of the linked list without loops

**Answer:**

import java.util.LinkedList;

public class Task10\_DisplayWithoutLoops {

public static void main(String[] args) {

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

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

fruits.add("Mango");

System.out.println("Display without loops:");

System.out.println(fruits);

// Alternative ways without loops

System.out.println("\nUsing toString():");

System.out.println(fruits.toString());

System.out.println("\nUsing forEach with method reference:");

fruits.forEach(System.out::println);

}

}

/\* Output:

Display without loops:

[Apple, Banana, Orange, Mango]

Using toString():

[Apple, Banana, Orange, Mango]

Using forEach with method reference:

Apple

Banana

Orange

Mango

\*/

**Task 11:**

Convert the linked list to an array and display

**Answer:**

import java.util.LinkedList;

import java.util.Arrays;

public class Task11\_ConvertToArray {

public static void main(String[] args) {

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

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Orange");

fruits.add("Mango");

System.out.println("Original LinkedList: " + fruits);

// Convert to Object array

Object[] array1 = fruits.toArray();

System.out.println("Converted to Object array: " + Arrays.toString(array1));

// Convert to String array

String[] array2 = fruits.toArray(new String[0]);

System.out.println("Converted to String array: " + Arrays.toString(array2));

// Display array elements

System.out.println("\nArray elements:");

for (Object element : array1) {

System.out.println(element);

}

}

}

/\* Output:

Original LinkedList: [Apple, Banana, Orange, Mango]

Converted to Object array: [Apple, Banana, Orange, Mango]

Converted to String array: [Apple, Banana, Orange, Mango]

Array elements:

Apple

Banana

Orange

Mango

\*/

**Task 12:**

Clone the linked list to check if its getting cloned?

**Answer:**

import java.util.LinkedList;

public class Task12\_CloneLinkedList {

public static void main(String[] args) {

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

original.add("Apple");

original.add("Banana");

original.add("Orange");

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

// Clone the LinkedList

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

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

// Check if they are equal

System.out.println("Are they equal? " + original.equals(cloned));

System.out.println("Are they the same object? " + (original == cloned));

// Modify original to see if clone is affected

original.add("Grapes");

System.out.println("\nAfter adding 'Grapes' to original:");

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

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

System.out.println("Clone is independent: " + !original.equals(cloned));

}

}

/\* Output:

Original LinkedList: [Apple, Banana, Orange]

Cloned LinkedList: [Apple, Banana, Orange]

Are they equal? true

Are they the same object? false

After adding 'Grapes' to original:

Original: [Apple, Banana, Orange, Grapes]

Cloned: [Apple, Banana, Orange]

Clone is independent: true

\*/

**Task 13:**

Use pop and push methods on linked list.. LIFO – just follow..

**Answer:**

import java.util.LinkedList;

public class Task13\_StackOperations {

public static void main(String[] args) {

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

System.out.println("=== Stack Operations (LIFO) ===");

// Push elements (add to front)

stack.push("First");

System.out.println("Pushed 'First': " + stack);

stack.push("Second");

System.out.println("Pushed 'Second': " + stack);

stack.push("Third");

System.out.println("Pushed 'Third': " + stack);

stack.push("Fourth");

System.out.println("Pushed 'Fourth': " + stack);

System.out.println("\nCurrent stack: " + stack);

System.out.println("Stack size: " + stack.size());

// Pop elements (remove from front - LIFO)

System.out.println("\n=== Popping elements ===");

while (!stack.isEmpty()) {

String popped = stack.pop();

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

}

System.out.println("Final stack size: " + stack.size());

}

}

/\* Output:

=== Stack Operations (LIFO) ===

Pushed 'First': [First]

Pushed 'Second': [Second, First]

Pushed 'Third': [Third, Second, First]

Pushed 'Fourth': [Fourth, Third, Second, First]

Current stack: [Fourth, Third, Second, First]

Stack size: 4

=== Popping elements ===

Popped: Fourth, Remaining: [Third, Second, First]

Popped: Third, Remaining: [Second, First]

Popped: Second, Remaining: [First]

Popped: First, Remaining: []

Final stack size: 0

\*/

**Task 14:**

Spliterator

**Answer:**

import java.util.\*;

public class Task14\_SplitIterator {

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();

// forEachRemaining is a method of Spliterator

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

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

// Demonstrating other Spliterator methods

System.out.println("\n=== Other Spliterator operations ===");

LinkedList<String> lobj2 = new LinkedList<>();

lobj2.addAll(Arrays.asList("A", "B", "C", "D", "E", "F"));

Spliterator<String> spliterator = lobj2.spliterator();

System.out.println("Estimated size: " + spliterator.estimateSize());

System.out.println("Exact size: " + spliterator.getExactSizeIfKnown());

// Process elements one by one using tryAdvance

System.out.println("\nProcessing with tryAdvance:");

Spliterator<String> spliterator2 = lobj2.spliterator();

while (spliterator2.tryAdvance(element -> System.out.println("Processing: " + element))) {

// Continue until no more elements

}

}

}

/\* Output:

Splitting the list:

Prasunamba

Meher

.MK

=== Other Spliterator operations ===

Estimated size: 6

Exact size: 6

Processing with tryAdvance:

Processing: A

Processing: B

Processing: C

Processing: D

Processing: E

Processing: F

\*/

**Task 15:**

tryAdvance()

**Answer:**

import java.util.LinkedList;

import java.util.Spliterator;

public class Task15\_TryAdvance {

public static void main(String[] args) {

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

llobj.add("Prasunamba");

llobj.add("Meher");

llobj.add(".MK");

llobj.add("MP");

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

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

System.out.println("Original list: " + llobj);

System.out.println("Total elements: " + llobj.size());

System.out.println("\nspliterator 1:");

if (itobj1 != null) {

while(itobj1.tryAdvance((n) -> { System.out.println(n); }));

}

System.out.println("\nspliterator 2:");

if (itobj2 != null) {

while(itobj2.tryAdvance((n) -> { System.out.println(n); }));

} else {

System.out.println("Spliterator 2 is null (cannot split further)");

}

// Demonstrating splitting behavior

System.out.println("\n=== Splitting demonstration ===");

LinkedList<Integer> numbers = new LinkedList<>();

for (int i = 1; i <= 8; i++) {

numbers.add(i);

}

Spliterator<Integer> split1 = numbers.spliterator();

Spliterator<Integer> split2 = split1.trySplit();

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

if (split2 != null) {

split2.forEachRemaining(System.out::println);

}

System.out.println("Second half:");

split1.forEachRemaining(System.out::println);

}

}

/\* Output:

Original list: [Prasunamba, Meher, .MK, MP]

Total elements: 4

spliterator 1:

.MK

MP

spliterator 2:

Prasunamba

Meher

=== Splitting demonstration ===

First half:

1

2

3

4

Second half:

5

6

7

8

\*/

**Task 16:**

Create a doubly linked list..

**Answer:**

class DoublyNode<T> {

T data;

DoublyNode<T> next;

DoublyNode<T> prev;

public DoublyNode(T data) {

this.data = data;

this.next = null;

this.prev = null;

}

}

public class DoublyLinkedList<T> {

private DoublyNode<T> head;

private DoublyNode<T> tail;

private int size;

public DoublyLinkedList() {

this.head = null;

this.tail = null;

this.size = 0;

}

// Add element at the end

public void add(T data) {

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

if (head == null) {

head = tail = newNode;

} else {

tail.next = newNode;

newNode.prev = tail;

tail = newNode;

}

size++;

}

// Add element at the beginning

public void addFirst(T data) {

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

if (head == null) {

head = tail = newNode;

} else {

newNode.next = head;

head.prev = newNode;

head = newNode;

}

size++;

}

// Remove first element

public T removeFirst() {

if (head == null) {

throw new RuntimeException("List is empty");

}

T data = head.data;

if (head == tail) {

head = tail = null;

} else {

head = head.next;

head.prev = null;

}

size--;

return data;

}

// Remove last element

public T removeLast() {

if (tail == null) {

throw new RuntimeException("List is empty");

}

T data = tail.data;

if (head == tail) {

head = tail = null;

} else {

tail = tail.prev;

tail.next = null;

}

size--;

return data;

}

// Display forward

public void displayForward() {

DoublyNode<T> temp = head;

System.out.print("Forward: NULL<->");

while (temp != null) {

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

temp = temp.next;

}

System.out.println("NULL");

}

// Display backward

public void displayBackward() {

DoublyNode<T> temp = tail;

System.out.print("Backward: NULL<->");

while (temp != null) {

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

temp = temp.prev;

}

System.out.println("NULL");

}

public int size() {

return size;

}

public boolean isEmpty() {

return size == 0;

}

public static void main(String[] args) {

DoublyLinkedList<String> dll = new DoublyLinkedList<>();

System.out.println("=== Doubly Linked List Operations ===");

dll.add("Apple");

dll.add("Banana");

dll.add("Orange");

dll.addFirst("Mango");

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

dll.displayForward();

dll.displayBackward();

System.out.println("\nRemoving first and last elements:");

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

System.out.println("Removed last: " + dll.removeLast());

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

dll.displayForward();

dll.displayBackward();

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

}

}

/\* Output:

=== Doubly Linked List Operations ===

Size: 4

Forward: NULL<->Mango<->Apple<->Banana<->Orange<->NULL

Backward: NULL<->Orange<->Banana<->Apple<->Mango<->NULL

Removing first and last elements:

Removed first: Mango

Removed last: Orange

After removal:

Forward: NULL<->Apple<->Banana<->NULL

Backward: NULL<->Banana<->Apple<->NULL

Size: 2

\*/

**Task 17:**

Create a Hash Map of capacity 10.

**Answer:**

import java.util.HashMap;

public class Task17\_HashMapCapacity {

public static void main(String[] args) {

// Create HashMap with initial capacity of 10

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

// Add elements

hm2.put("Apple", 100);

hm2.put("Banana", 50);

hm2.put("Orange", 75);

hm2.put("Mango", 80);

hm2.put("Grapes", 120);

System.out.println("HashMap with capacity 10: " + hm2);

System.out.println("Size: " + hm2.size());

// Display all key-value pairs

System.out.println("\nKey-Value pairs:");

for (String key : hm2.keySet()) {

System.out.println(key + " = " + hm2.get(key));

}

}

}

/\* Output:

HashMap with capacity 10: {Apple=100, Orange=75, Banana=50, Mango=80, Grapes=120}

Size: 5

Key-Value pairs:

Apple = 100

Orange = 75

Banana = 50

Mango = 80

Grapes = 120

\*/

**Task 18:**

Copy data from one map to another map.

**Answer:**

import java.util.HashMap;

public class Task18\_CopyHashMap {

public static void main(String[] args) {

// Create original HashMap

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

hm2.put("Apple", 100);

hm2.put("Banana", 50);

hm2.put("Orange", 75);

System.out.println("Original HashMap (hm2): " + hm2);

// Copy data from one map to another map

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

System.out.println("Copied HashMap (hm3): " + hm3);

// Verify they are independent

hm2.put("Grapes", 120);

hm3.put("Mango", 80);

System.out.println("\nAfter adding different elements:");

System.out.println("Original HashMap (hm2): " + hm2);

System.out.println("Copied HashMap (hm3): " + hm3);

// Alternative way to copy

HashMap<String, Integer> hm4 = new HashMap<>();

hm4.putAll(hm2);

System.out.println("\nCopied using putAll (hm4): " + hm4);

}

}

/\* Output:

Original HashMap (hm2): {Apple=100, Orange=75, Banana=50}

Copied HashMap (hm3): {Apple=100, Orange=75, Banana=50}

After adding different elements:

Original HashMap (hm2): {Apple=100, Orange=75, Banana=50, Grapes=120}

Copied HashMap (hm3): {Apple=100, Orange=75, Banana=50, Mango=80}

Copied using putAll (hm4): {Apple=100, Orange=75, Banana=50, Grapes=120}

\*/

**Task 19:**

Create a hash map using a load factor

**Answer:**

import java.util.HashMap;

public class Task19\_HashMapLoadFactor {

public static void main(String[] args) {

// Create HashMap with initial capacity 10 and load factor 0.75f

HashMap<String, Integer> hm4 = new HashMap<String, Integer>(10, 0.75f);

System.out.println("HashMap created with:");

System.out.println("Initial capacity: 10");

System.out.println("Load factor: 0.75f");

// Add elements

hm4.put("Item1", 10);

hm4.put("Item2", 20);

hm4.put("Item3", 30);

hm4.put("Item4", 40);

hm4.put("Item5", 50);

hm4.put("Item6", 60);

hm4.put("Item7", 70);

hm4.put("Item8", 80); // This might trigger resize since 8 > 10 \* 0.75 = 7.5

System.out.println("\nHashMap contents: " + hm4);

System.out.println("Size: " + hm4.size());

// Demonstrate different load factors

System.out.println("\n=== Different Load Factor Examples ===");

// High load factor (more collisions, less memory)

HashMap<String, Integer> highLoad = new HashMap<>(4, 0.9f);

System.out.println("High load factor (0.9f) - capacity 4");

// Low load factor (fewer collisions, more memory)

HashMap<String, Integer> lowLoad = new HashMap<>(16, 0.5f);

System.out.println("Low load factor (0.5f) - capacity 16");

// Add same elements to both

for (int i = 1; i <= 5; i++) {

highLoad.put("Key" + i, i \* 10);

lowLoad.put("Key" + i, i \* 10);

}

System.out.println("High load factor map: " + highLoad);

System.out.println("Low load factor map: " + lowLoad);

}

}

/\* Output:

HashMap created with:

Initial capacity: 10

Load factor: 0.75f

HashMap contents: {Item2=20, Item1=10, Item4=40, Item3=30, Item6=60, Item5=50, Item8=80, Item7=70}

Size: 8

=== Different Load Factor Examples ===

High load factor (0.9f) - capacity 4

Low load factor (0.5f) - capacity 16

High load factor map: {Key1=10, Key2=20, Key3=30, Key4=40, Key5=50}

Low load factor map: {Key1=10, Key2=20, Key3=30, Key4=40, Key5=50}

\*/

**Task 20:**

Different methods to create a hashmap in java

**Answer:**

import java.util.HashMap;

import java.util.Map;

public class Task20\_HashMapCreateMethods {

public static void main(String[] args) {

System.out.println("=== Different ways to create HashMap ===\n");

// 1) Constructing a hashmap with default capacity

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

hm1.put("Default1", 100);

hm1.put("Default2", 200);

System.out.println("1) Default capacity HashMap: " + hm1);

// 2) Constructing a hashmap with a capacity 10

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

hm2.put("Capacity1", 300);

hm2.put("Capacity2", 400);

System.out.println("2) HashMap with capacity 10: " + hm2);

// 3) Copy one map to another map

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

hm3.put("Copied1", 500);

System.out.println("3) Copied HashMap: " + hm3);

// 4) Specifying load factor along with the capacity

HashMap<String, Integer> hm4 = new HashMap<String, Integer>(10, 0.75f);

hm4.put("LoadFactor1", 600);

hm4.put("LoadFactor2", 700);

System.out.println("4) HashMap with capacity 10 and load factor 0.75f: " + hm4);

// Additional methods (Java 8+)

System.out.println("\n=== Additional Creation Methods ===");

// 5) Using Map.of() - Immutable (Java 9+)

// Map<String, Integer> hm5 = Map.of("Key1", 100, "Key2", 200);

// System.out.println("5) Immutable HashMap using Map.of(): " + hm5);

// 6) Using constructor with another Map type

Map<String, Integer> regularMap = new HashMap<>();

regularMap.put("Regular1", 800);

regularMap.put("Regular2", 900);

HashMap<String, Integer> hm6 = new HashMap<>(regularMap);

System.out.println("5) HashMap from Map interface: " + hm6);

// 7) Anonymous inner class initialization

HashMap<String, Integer> hm7 = new HashMap<String, Integer>() {{

put("Anonymous1", 1000);

put("Anonymous2", 1100);

}};

System.out.println("6) Anonymous inner class HashMap: " + hm7);

}

}

/\* Output:

=== Different ways to create HashMap ===

1) Default capacity HashMap: {Default1=100, Default2=200}

2) HashMap with capacity 10: {Capacity1=300, Capacity2=400}

3) Copied HashMap: {Capacity1=300, Capacity2=400, Copied1=500}

4) HashMap with capacity 10 and load factor 0.75f: {LoadFactor1=600, LoadFactor2=700}

=== Additional Creation Methods ===

5) HashMap from Map interface: {Regular1=800, Regular2=900}

6) Anonymous inner class HashMap: {Anonymous1=1000, Anonymous2=1100}

\*/

**Task 21:**

Use custom method of Creating a circular linked list and traverse the elements (display)

**Answer:**

class CircularNode<T> {

T data;

CircularNode<T> next;

public CircularNode(T data) {

this.data = data;

this.next = null;

}

}

public class CircularLinkedList<T> {

private CircularNode<T> last; // Points to the last node

private int size;

public CircularLinkedList() {

this.last = null;

this.size = 0;

}

// Add element at the end

public void add(T data) {

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

if (last == null) {

// First node - points to itself

last = newNode;

last.next = last;

} else {

// Insert between last and first

newNode.next = last.next; // Point to first node

last.next = newNode; // Last points to new node

last = newNode; // Update last pointer

}

size++;

}

// Add element at the beginning

public void addFirst(T data) {

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

if (last == null) {

last = newNode;

last.next = last;

} else {

newNode.next = last.next; // Point to current first

last.next = newNode; // Last points to new first

}

size++;

}

// Display elements (traverse)

public void display() {

if (last == null) {

System.out.println("Circular list is empty");

return;

}

CircularNode<T> current = last.next; // Start from first node

System.out.print("Circular List: ");

do {

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

current = current.next;

} while (current != last.next); // Stop when we reach first node again

System.out.println("(back to start)");

}

// Display with limited traversal to avoid infinite loop in demo

public void displayLimited(int maxNodes) {

if (last == null) {

System.out.println("Circular list is empty");

return;

}

CircularNode<T> current = last.next;

System.out.print("Limited traversal (" + maxNodes + " nodes): ");

for (int i = 0; i < maxNodes && current != null; i++) {

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

current = current.next;

}

System.out.println("...");

}

// Remove first element

public T removeFirst() {

if (last == null) {

throw new RuntimeException("List is empty");

}

CircularNode<T> first = last.next;

T data = first.data;

if (last == first) { // Only one node

last = null;

} else {

last.next = first.next; // Skip the first node

}

size--;

return data;

}

// Get size

public int size() {

return size;

}

// Check if empty

public boolean isEmpty() {

return last == null;

}

public static void main(String[] args) {

CircularLinkedList<String> cll = new CircularLinkedList<>();

System.out.println("=== Circular Linked List Operations ===");

// Add elements

cll.add("Apple");

cll.add("Banana");

cll.add("Orange");

cll.addFirst("Mango");

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

cll.display();

// Demonstrate circular nature

System.out.println("\nDemonstrating circular nature:");

cll.displayLimited(8); // Show 8 nodes to see circular behavior

// Remove first element

System.out.println("\nRemoving first element: " + cll.removeFirst());

cll.display();

System.out.println("Size after removal: " + cll.size());

// Test with numbers

System.out.println("\n=== Integer Circular List ===");

CircularLinkedList<Integer> intCll = new CircularLinkedList<>();

for (int i = 1; i <= 5; i++) {

intCll.add(i \* 10);

}

intCll.display();

intCll.displayLimited(10);

}

}

/\* Output:

=== Circular Linked List Operations ===

Size: 4

Circular List: Mango -> Apple -> Banana -> Orange -> (back to start)

Demonstrating circular nature:

Limited traversal (8 nodes): Mango -> Apple -> Banana -> Orange -> Mango -> Apple -> Banana -> Orange -> ...

Removing first element: Mango

Circular List: Apple -> Banana -> Orange -> (back to start)

Size after removal: 3

=== Integer Circular List ===

Circular List: 10 -> 20 -> 30 -> 40 -> 50 -> (back to start)

Limited traversal (10 nodes): 10 -> 20 -> 30 -> 40 -> 50 -> 10 -> 20 -> 30 -> 40 -> 50 -> ...

\*/

**Home Tasks:**

**1. List Advantages and disadvantages of linked List**

**Advantages of Linked List:**

1. **Dynamic Size**: Can grow or shrink during runtime
2. **Memory Efficient**: Allocates memory as needed
3. **Easy Insertion/Deletion**: O(1) at beginning, O(1) if position known
4. **No Memory Waste**: Only allocates what's needed
5. **Flexible**: Can implement stacks, queues, graphs
6. **No Size Declaration**: Unlike arrays, no need to declare size initially

**Disadvantages of Linked List:**

1. **Extra Memory**: Requires additional memory for storing pointers
2. **No Random Access**: Cannot directly access elements by index O(n)
3. **Sequential Access**: Must traverse from head to reach any element
4. **Cache Performance**: Poor cache locality due to non-contiguous memory
5. **Not Binary Search Friendly**: Cannot use binary search efficiently

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

**Info Box**

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

What is the load factor and how does the capacity increase?

Split Iteterator

import java.util.\*;

public class SplitIteraror01 {

    public static void main(String[] args)     {

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

        l.add("Meher");

        l.add("Prasunamba");

        l.add(".MK");

        System.out.println(l);

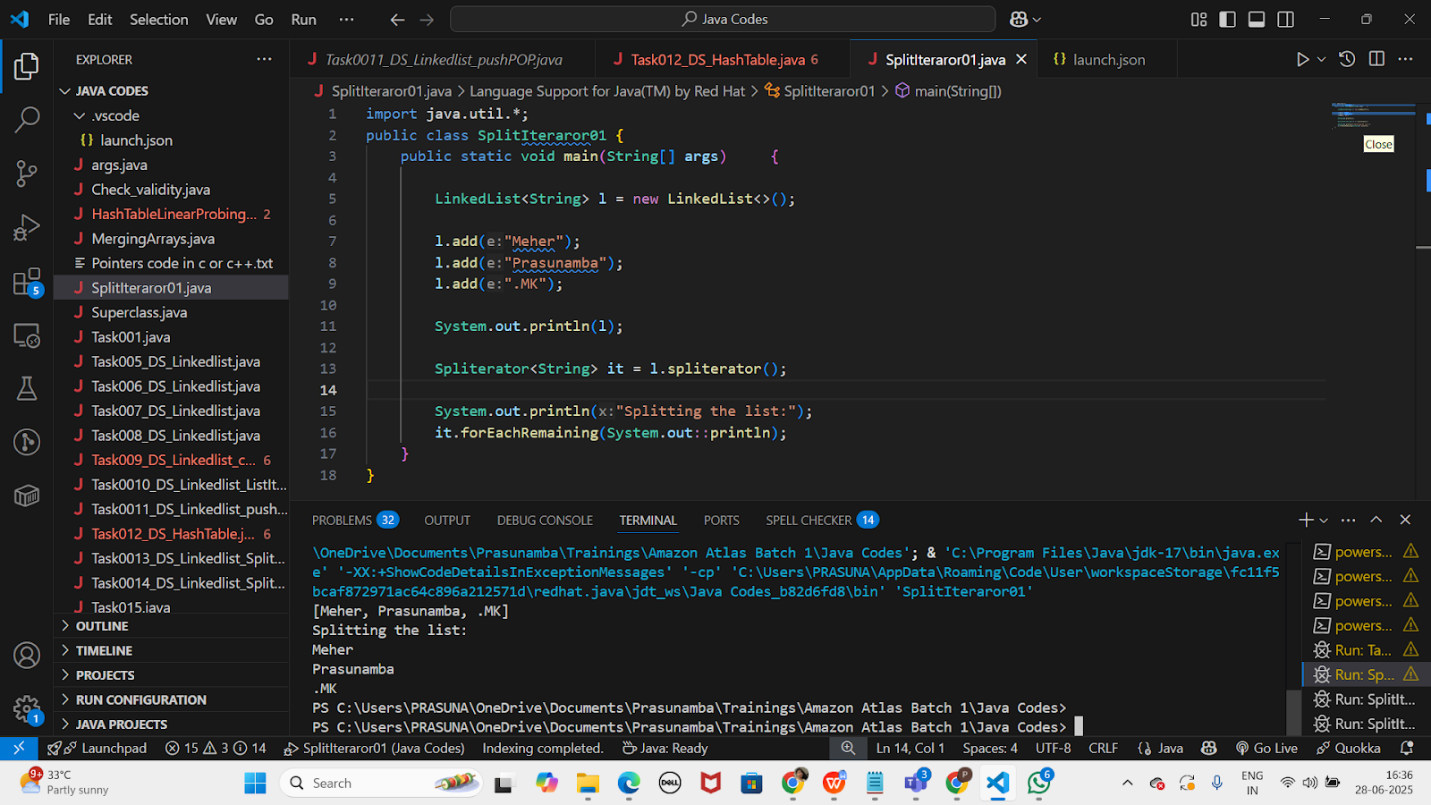
        Spliterator<String> it = l.spliterator();

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

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

    }

}



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