Task 1

Pointers :

#include <stdio.h>

int main() {

int age = 27; // Declare an integer variable

int \*ptr = &age; // Declare a pointer and store the address of 'age'

printf("Value of age: %d\n", age); // Output: Value of age: 27

printf("Address of age: %p\n", &age); // Output: Address of age: (some memory address)

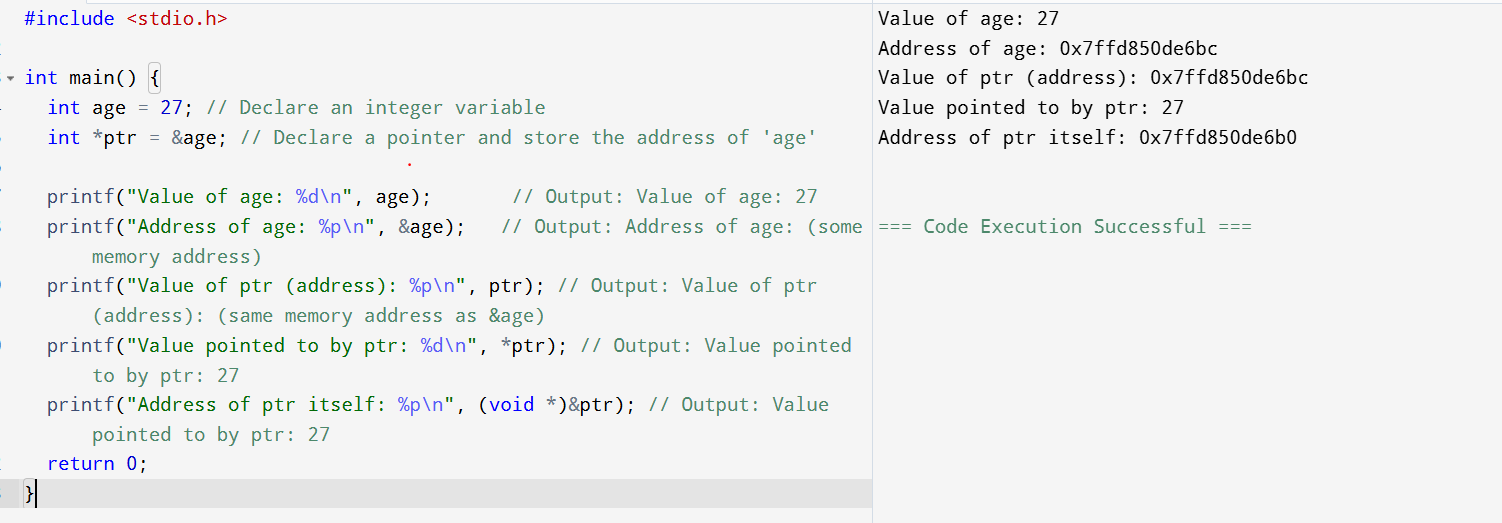
printf("Value of ptr (address): %p\n", ptr); // Output: Value of ptr (address): (same memory address as &age)

printf("Value pointed to by ptr: %d\n", \*ptr); // Output: Value pointed to by ptr: 27

printf("Address of ptr itself: %p\n", (void \*)&ptr); // Output: Value pointed to by ptr: 27

return 0;

}



Task 02:

Linked list in c++

#include <iostream>

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

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

};

// Define a Linked List class

class Linkedlist {

private:

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

public:

// Constructor to initialize an empty list

Linkedlist() {

head = nullptr;

}

// Insert a node at the end

void insertAtEnd(int value) {

Node\* newNode = new Node(value);

if (head == nullptr) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

}

}

// Delete a node by value

void deleteByValue(int value) {

if (head == nullptr) return;

if (head->data == value) {

Node\* temp = head;

head = head->next;

delete temp;

return;

}

Node\* temp = head;

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

temp = temp->next;

}

if (temp->next) {

Node\* nodeToDelete = temp->next;

temp->next = temp->next->next;

delete nodeToDelete;

}

}

// Display the list

void display() {

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << "NULL" << endl;

}

// Destructor to free memory

~Linkedlist() {

Node\* temp;

while (head != nullptr) {

temp = head;

head = head->next;

delete temp;

}

}

};

// Main function to demonstrate the list

int main() {

Linkedlist list;

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

cout << "Linked List: ";

list.display(); // Output: 10->20->30->NULL

list.deleteByValue(20);

cout << "After Deleting 20: ";

list.display(); // Output: 10->30->NULL

return 0;

}

**Task 03:**

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

// Define a Node class

class Node {

int data; // Data part

Node next; // Pointer to next node

// Constructor

Node(int value) {

data = value;

next = null;

}

}

// Define LinkedList class

class LinkedList {

private Node head; // Head of the list

// Constructor initializes empty list

public LinkedList() {

head = null;

}

// Insert node at the end

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;

}

}

// Delete node by value

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; // Java's GC will collect the unreferenced node

}

}

// Display the list

public void display() {

Node temp = head;

while (temp != null) {

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

temp = temp.next;

}

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

}

}

// Main class to test the LinkedList

public class LinkedListtestcase {

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(); // Output: 10->20->30->NULL

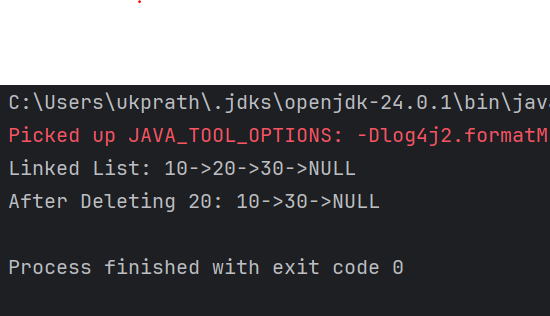
list.deleteByValue(20);

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

list.display(); // Output: 10->30->NULL

}

}



**Task 04:**

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

* ~~Create a node~~

11 pax done out of 36 @ 12.38

* Try to add element at the end of the list
* Remove the node
* Display all the elements of the List

11 pax done out of 36 @ 12.48

16 pax at 13.04

* Find size of the linked list
* Index out of bounds

// Generic Node class

class Node<T> {

T data;

Node<T> next;

public Node(T data) {

this.data = data;

this.next = null;

}

}

// Generic LinkedList class

class LinkedList<T> {

private Node<T> head;

private int size;

public LinkedList() {

head = null;

size = 0;

}

// Create and insert at the end

public void insertAtEnd(T value) {

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

if (head == null) {

head = newNode;

} else {

Node<T> temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

size++;

}

// Remove node by value

public void deleteByValue(T value) {

if (head == null) return;

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

head = head.next;

size--;

return;

}

Node<T> temp = head;

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

temp = temp.next;

}

if (temp.next != null) {

temp.next = temp.next.next;

size--;

}

}

// Display all elements

public void display() {

Node<T> temp = head;

while (temp != null) {

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

temp = temp.next;

}

System.out.println("NULL");

}

// Get size of the linked list

public int getSize() {

return size;

}

// Get element at specific index

public T get(int index) {

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

throw new IndexOutOfBoundsException("Index " + index + " is out of bounds!");

}

Node<T> temp = head;

int count = 0;

while (count < index) {

temp = temp.next;

count++;

}

return temp.data;

}

}

public class Main {

public static void main(String[] args) {

// Linked list that can hold any type of object

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

// Create nodes and insert at end

list.insertAtEnd(27); // Integer

list.insertAtEnd("Pratheesh"); // String

list.insertAtEnd(60.38); // Double

list.insertAtEnd(true); // Boolean

System.out.println("List after insertion:");

list.display(); // Output: 11 -> Pratheesh -> 60.38 -> true -> NULL

// Delete node

list.deleteByValue("Pratheesh");

System.out.println("After deleting 'Pratheesh':");

list.display(); // Output: 11 -> 60.38 -> true -> NULL

// Size of the list

System.out.println("Size of list: " + list.getSize()); // Output: 3

// Access valid index

System.out.println("Element at index 1: " + list.get(1)); // Output: 60.38

// Access invalid index

try {

System.out.println("Element at index 5: " + list.get(5));

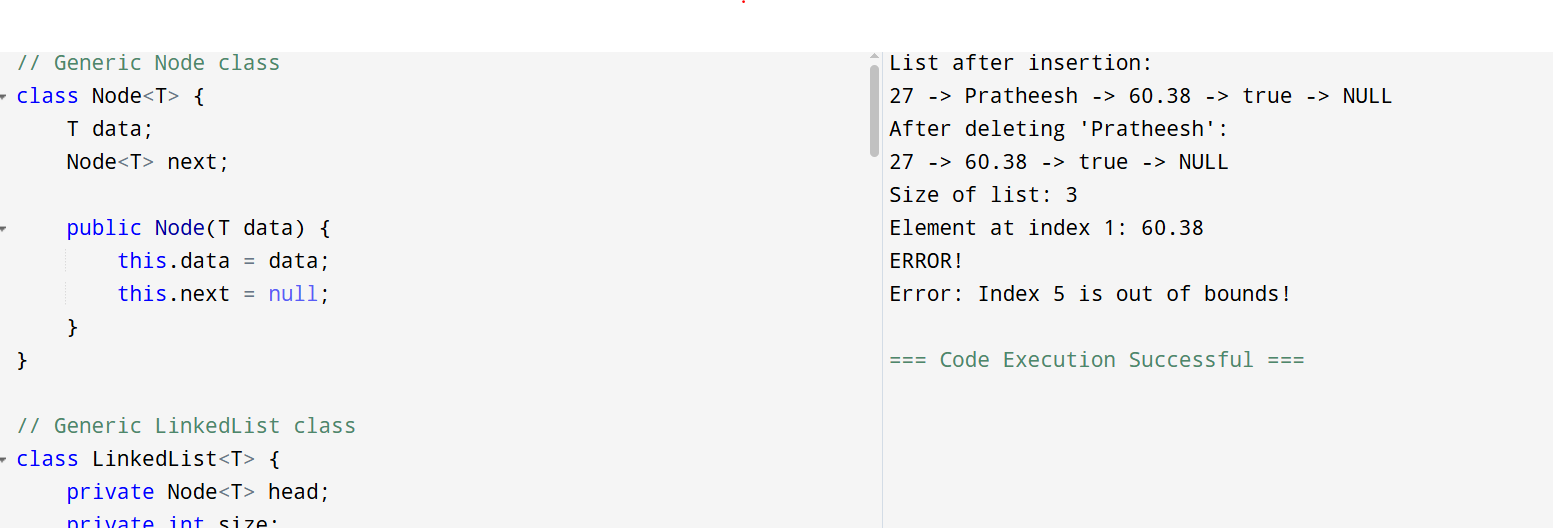
} catch (IndexOutOfBoundsException e) {

System.out.println("Error: " + e.getMessage());

}

}

}



**Task 05:**

**List down all the methods of Linked list**

### **Common LinkedList Methods**

#### **🔹 Adding Elements**

* add(E e) – Adds to the end.
* add(int index, E element) – Inserts at specific index.
* addFirst(E e) – Adds to the beginning.
* addLast(E e) – Adds to the end.
* offer(E e) – Adds to the end (returns boolean).
* offerFirst(E e) – Adds to the beginning.
* offerLast(E e) – Adds to the end.

#### **🔹 Accessing Elements**

* get(int index) – Gets element at index.
* getFirst() – Gets first element.
* getLast() – Gets last element.
* peek() – Retrieves head without removing.
* peekFirst() – Retrieves first without removing.
* peekLast() – Retrieves last without removing.

#### **🔹 Updating Elements**

* set(int index, E element) – Replaces the element at index.

#### **🔹 Removing Elements**

* remove() – Removes head.
* remove(int index) – Removes element at index.
* remove(Object o) – Removes first occurrence.
* removeFirst() – Removes first element.
* removeLast() – Removes last element.
* poll() – Retrieves and removes head.
* pollFirst() – Retrieves and removes first.
* pollLast() – Retrieves and removes last.
* clear() – Removes all elements.

#### **🔹 Searching**

* contains(Object o) – Checks if list contains element.
* indexOf(Object o) – First index of element.
* lastIndexOf(Object o) – Last index of element.
* isEmpty() – Checks if list is empty.
* size() – Returns number of elements.

#### **🔹 Iteration**

* iterator() – Returns an iterator.
* descendingIterator() – Reverse-order iterator.
* listIterator() – List iterator over elements.
* listIterator(int index) – List iterator starting at index.

**Task 6**

**import java.util.LinkedList;**

**public class task06 {**

**public static void main(String[] args) {**

**// Create LinkedList of strings**

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

**// Add elements to the list**

**fruits.add("Apple");**

**fruits.add("Banana");**

**fruits.add("Mango");**

**fruits.add("Orange");**

**// Display the list**

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

**// Add at first and last**

**fruits.addFirst("Pineapple");**

**fruits.addLast("Grapes");**

**System.*out*.println("After adding first and last: " + fruits);**

**}**

**}**

**Task7**

**import java.util.LinkedList;**

**public class task07 {**

**public static void main(String[] args) {**

**// Create LinkedList of strings**

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

**// Add elements to the list**

**fruits.add("Apple");**

**fruits.add("Banana");**

**fruits.add("Mango");**

**fruits.add("Orange");**

**// Display the list**

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

**// Add at first and last**

**fruits.addFirst("Pineapple");**

**fruits.addLast("Grapes");**

**System.*out*.println("After adding first and last: " + fruits);**

**// Removing the first element**

**fruits.removeFirst();**

**// Removing the last element**

**fruits.removeLast();**

**// Displaying the updated LinkedList**

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

**}**

**}**

**Task 8**

**import java.util.LinkedList;**

**public class task08 {**

**public static void main(String[] args) {**

**// Create LinkedList of strings**

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

**// Add elements to the list**

**fruits.add("Apple");**

**fruits.add("Banana");**

**fruits.add("Mango");**

**fruits.add("Orange");**

**// Display the list**

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

**// Add at first and last**

**fruits.addFirst("Pineapple");**

**fruits.addLast("Grapes");**

**System.*out*.println("After adding first and last: " + fruits);**

**// Removing the first element**

**fruits.removeFirst();**

**// Removing the last element**

**fruits.removeLast();**

**// Displaying the updated LinkedList**

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

**// Updating the first element**

**fruits.set(1, "Pineapple");**

**// Displaying the updated list**

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

**}**

**}**

**Task09**

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

**import java.util.LinkedList;**

**public class task09 {**

**public static void main(String[] args) {**

**// Creating and populating LinkedList**

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

**fruits.add("Apple");**

**fruits.add("Banana");**

**fruits.add("Mango");**

**fruits.add("Orange");**

**// 1. Display using get() and for loop**

**System.*out*.println("Displaying using get() and for loop:");**

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

**System.*out*.println(fruits.get(i));**

**}**

**// 2. Display using for-each loop**

**System.*out*.println("\nDisplaying using for-each loop:");**

**for (String fruit : fruits) {**

**System.*out*.println(fruit);**

**}**

**}**

**}**

**Task 10: display the elements of the linked list with out loops**

import java.util.LinkedList;

public class task010 {

public static void main(String[] args) {

// Create and populate the LinkedList

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

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Mango");

fruits.add("Orange");

// Display the LinkedList without using loops

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

}

}

task 11: convert the linked list to an array and display Hint : Object[] a = llobj.toArray();

import java.util.LinkedList;

import java.util.Arrays;

public class task011 {

public static void main(String[] args) {

// Create and populate the LinkedList

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

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Mango");

fruits.add("Orange");

// Convert LinkedList to array

Object[] fruitArray = fruits.toArray();

System.*out*.println(Arrays.*toString*(fruitArray));

// Display the array elements

System.*out*.println("Elements in array form:");

for (Object fruit : fruitArray) {

System.*out*.println(fruit);

}

}

}

Task012

import java.util.LinkedList;

public class task012 {

public static void main(String[] args) {

// Create and populate the original LinkedList

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

fruits.add("Apple");

fruits.add("Banana");

fruits.add("Mango");

// Clone the LinkedList

LinkedList<String> clonedList = (LinkedList<String>) fruits.clone();

// Display original and cloned lists

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

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

System.*out*.println("Same object? " + (fruits == clonedList));

}

}

Task013

import java.util.LinkedList;

public class task013 {

public static void main(String[] args) {

// Create and populate a LinkedList

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

fruits.add("Banana");

fruits.add("Mango");

fruits.add("Orange");

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

// Use push() to add an element at the front (like stack)

fruits.push("Apple"); // Same as addFirst()

System.*out*.println("After push: " + fruits); // [Apple, Banana, Mango, Orange]

// Use pop() to remove the first element (like stack)

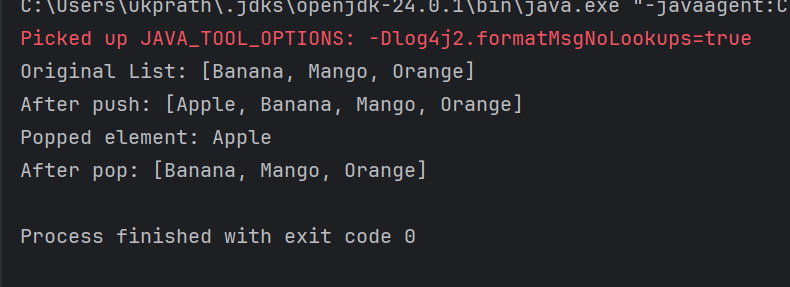
String popped = fruits.pop(); // Removes "Apple"

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

System.*out*.println("After pop: " + fruits); // [Banana, Mango, Orange]

}

}



Task 14:

Splititerator

import java.util.LinkedList;

import java.util.Spliterator;

import java.util.\*;

public class task014 {

public static void main(String[] args) {

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

l.add("Pratheesh");

l.add(".K");

l.add("BE");

l.add("MBA");

System.*out*.println(l);

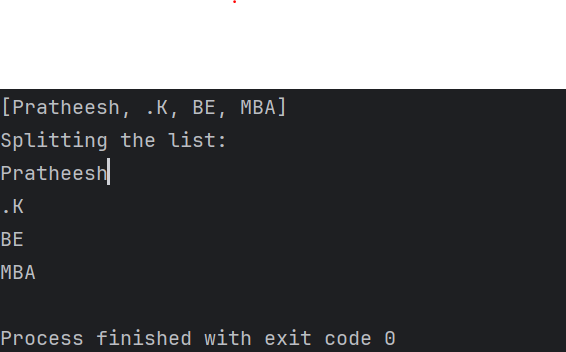
Spliterator<String> it = l.spliterator();

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

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

}

}



**Task 15:**

**tryAdvance()**

import java.util.LinkedList;

import java.util.Spliterator;

public class task015 {

public static void main(String[] args) {

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

llobj.add("Pratheesh");

llobj.add("K");

llobj.add("BE");

llobj.add("MBA");

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

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

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

while (itobj1 != null && itobj1.tryAdvance(n -> System.*out*.println(n)));

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

while (itobj2 != null && itobj2.tryAdvance(n -> System.*out*.println(n)));

}

}

Task 16:

Create a doubly linked list..

public class DoublyLinkedList {

// Node structure

private static class Node {

int data;

Node prev;

Node next;

Node(int d) {

data = d;

}

}

private Node head;

private Node tail;

// Add at beginning

public void insertAtBeginning(int data) {

Node node = new Node(data);

if (head == null) {

head = tail = node;

} else {

node.next = head;

head.prev = node;

head = node;

}

}

// Add at end

public void insertAtEnd(int data) {

Node node = new Node(data);

if (tail == null) {

head = tail = node;

} else {

tail.next = node;

node.prev = tail;

tail = node;

}

}

// Insert at specific position (1-based index)

public void insertAtPosition(int data, int position) {

if (position <= 1 || head == null) {

insertAtBeginning(data);

return;

}

Node current = head;

int idx = 1;

while (current.next != null && idx < position - 1) {

current = current.next;

idx++;

}

if (current.next == null) {

insertAtEnd(data);

} else {

Node node = new Node(data);

node.next = current.next;

node.prev = current;

current.next.prev = node;

current.next = node;

}

}

// Delete node at specific position (1-based index)

public void deleteAtPosition(int position) {

if (head == null) return;

if (position == 1) {

if (head == tail) {

head = tail = null;

} else {

head = head.next;

head.prev = null;

}

return;

}

Node current = head;

int idx = 1;

while (current != null && idx < position) {

current = current.next;

idx++;

}

if (current == null) return;

if (current == tail) {

tail = tail.prev;

tail.next = null;

} else {

current.prev.next = current.next;

current.next.prev = current.prev;

}

}

// Traverse forward

public void traverseForward() {

Node n = head;

while (n != null) {

System.*out*.print(n.data + " ");

n = n.next;

}

System.*out*.println();

}

// Traverse backward

public void traverseBackward() {

Node n = tail;

while (n != null) {

System.*out*.print(n.data + " ");

n = n.prev;

}

System.*out*.println();

}

// Sample usage

public static void main(String[] args) {

DoublyLinkedList dll = new DoublyLinkedList();

dll.insertAtEnd(10);

dll.insertAtEnd(20);

dll.insertAtBeginning(5);

dll.insertAtPosition(15, 3);

dll.traverseForward(); // 5 10 15 20

dll.traverseBackward(); // 20 15 10 5

dll.deleteAtPosition(2);

dll.traverseForward(); // 5 15 20

}

}

Task 17:

Create a Hash MAp of capacity 10.

Hint:

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

import java.util.HashMap;

public class task017 {

public static void main(String[] args) {

// Creating a HashMap with initial capacity of 10

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

// Adding some elements to the HashMap

hm.put("One", 1);

hm.put("Two", 2);

hm.put("Three", 3);

// Printing the HashMap

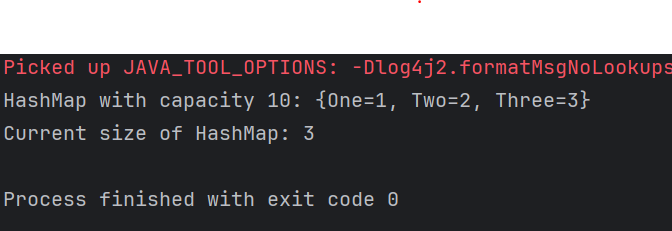
System.*out*.println("HashMap with capacity 10: " + hm);

// You can also check the size of the HashMap

System.*out*.println("Current size of HashMap: " + hm.size());

}

}



Task 18:

Copy data from one map to another map.

hint:

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

import java.util.HashMap;

public class task018 {

public static void main(String[] args) {

// Original map

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

hm2.put("Apple", 10);

hm2.put("Banana", 20);

hm2.put("Mango", 30);

// Copy data from hm2 to hm3 using constructor

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

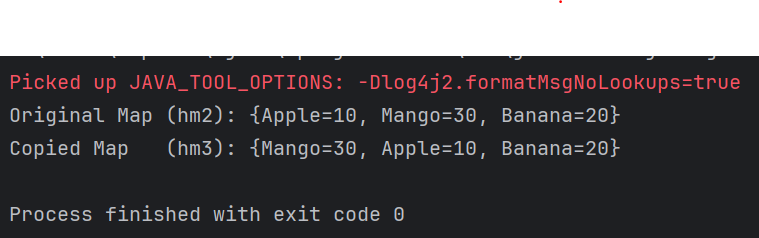
// Display the copied map

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

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

}

}



Task 19:

Create a hash map using a lod factor

Hint:

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

Initial capacity ===10

Load factor === 0.75f

import java.util.HashMap;

public class task019 {

public static void main(String[] args) {

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

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

// Calculate threshold

int initialCapacity = 10;

float loadFactor = 0.75f;

int threshold = (int)(initialCapacity \* loadFactor);

System.*out*.println("Initial Configuration:");

System.*out*.println("Initial Capacity: " + initialCapacity);

System.*out*.println("Load Factor: " + loadFactor);

System.*out*.println("Threshold before rehashing: " + threshold);

// Adding elements to HashMap

System.*out*.println("\nAdding elements to HashMap:");

hm.put("One", 1);

System.*out*.println("Added One: " + hm.size() + " elements");

hm.put("Two", 2);

System.*out*.println("Added Two: " + hm.size() + " elements");

hm.put("Three", 3);

System.*out*.println("Added Three: " + hm.size() + " elements");

hm.put("Four", 4);

System.*out*.println("Added Four: " + hm.size() + " elements");

hm.put("Five", 5);

System.*out*.println("Added Five: " + hm.size() + " elements");

hm.put("Six", 6);

System.*out*.println("Added Six: " + hm.size() + " elements");

hm.put("Seven", 7);

System.*out*.println("Added Seven: " + hm.size() + " elements");

// After threshold (7.5), HashMap will rehash

hm.put("Eight", 8);

System.*out*.println("Added Eight: " + hm.size() + " elements (Rehashing may occur)");

// Display final HashMap

System.*out*.println("\nFinal HashMap contents:");

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

System.*out*.println("Key: " + key + ", Value: " + hm.get(key));

}

// HashMap operations

System.*out*.println("\nHashMap Operations:");

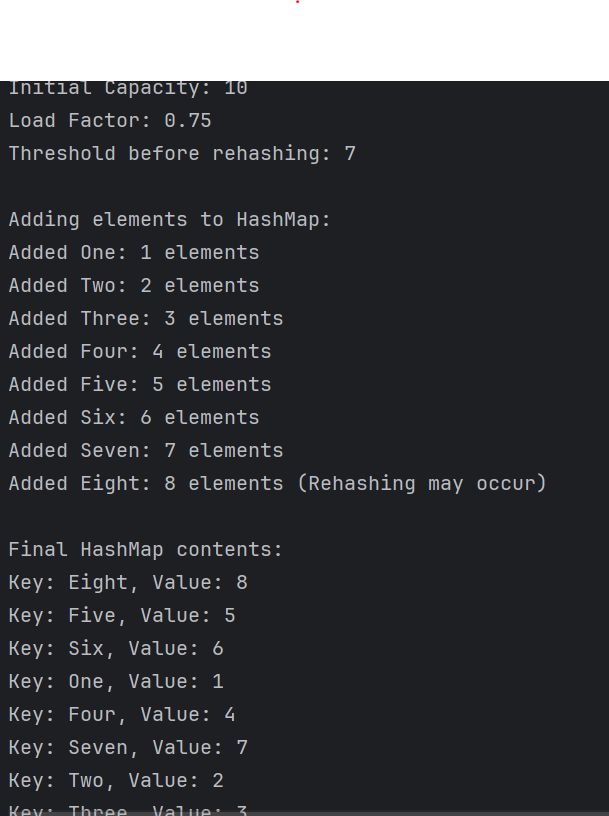
System.*out*.println("Size of HashMap: " + hm.size());

System.*out*.println("Contains key 'Three'? " + hm.containsKey("Three"));

System.*out*.println("Value for key 'Four': " + hm.get("Four"));

}

}



**Task020\_DS\_HashMapCreateMethods:**

Different methods to create a hashmap in java :

1) Constructing a hashmap with default capacity

ex:

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

2) Constructing a hashmap with a capacity 10

ex:

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

3)copy one map to another map

ex:

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

4)

Specifying load factor along with the capacity

ex:

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

Initial capacity ===10

Load factor === 0.75f

**Task 21**

public class CircularLinkedList {

// Node class

class Node {

int data;

Node next;

public Node(int data) {

this.data = data;

this.next = null;

}

}

private Node head = null;

private Node tail = null;

// Method to add a node to the circular linked list

public void addNode(int data) {

Node newNode = new Node(data);

// If list is empty

if (head == null) {

head = newNode;

tail = newNode;

newNode.next = head; // Make it circular

} else {

tail.next = newNode; // Add new node after tail

tail = newNode; // Make new node as tail

tail.next = head; // Connect tail to head

}

}

// Method to display the circular linked list

public void display() {

if (head == null) {

System.*out*.println("Circular Linked List is empty");

return;

}

Node current = head;

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

do {

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

current = current.next;

} while (current != head);

System.*out*.println("(back to head)");

}

// Method to count nodes in the circular linked list

public int countNodes() {

if (head == null) return 0;

int count = 0;

Node current = head;

do {

count++;

current = current.next;

} while (current != head);

return count;

}

public static void main(String[] args) {

CircularLinkedList cll = new CircularLinkedList();

// Adding nodes

cll.addNode(1);

cll.addNode(2);

cll.addNode(3);

cll.addNode(4);

cll.addNode(5);

// Display the list

cll.display();

// Display count of nodes

System.*out*.println("Number of nodes: " + cll.countNodes());

// Demonstrate circular nature

System.*out*.println("\nDemonstrating circular nature (10 iterations):");

Node current = cll.head;

for(int i = 0; i < 10; i++) {

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

current = current.next;

}

}

}

