**Q1. Delete the middle element from a linked list**

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

this.next = null;

}

}

public class LinkedListOperations {

public static void deleteMiddle(ListNode head) {

if (head == null || head.next == null) return;

ListNode slow = head;

ListNode fast = head;

ListNode prev = null;

while (fast != null && fast.next != null) {

prev = slow;

slow = slow.next;

fast = fast.next.next;

}

if (prev != null) {

prev.next = slow.next;

}

}

}

**Q2. Reverse a linked list**

public static ListNode reverseList(ListNode head) {

ListNode prev = null;

ListNode current = head;

ListNode next = null;

while (current != null) {

next = current.next;

current.next = prev;

prev = current;

current = next;

}

return prev;

}

**Q3. Delete last occurrence of an element from a linked list**

public static void deleteLastOccurrence(ListNode head, int key) {

ListNode temp = head;

ListNode lastOccurrence = null;

ListNode lastOccurrencePrev = null;

while (temp != null) {

if (temp.val == key) {

lastOccurrencePrev = lastOccurrence;

lastOccurrence = temp;

}

temp = temp.next;

}

if (lastOccurrence != null) {

if (lastOccurrencePrev != null) {

lastOccurrencePrev.next = lastOccurrence.next;

} else {

head = head.next;

}

}

}

**Q4. Implement LinkedList in Java**

class Node {

int data;

Node next;

Node(int data) {

this.data = data;

this.next = null;

}

}

public class LinkedList {

private Node head;

public void add(int data) {

Node newNode = new Node(data);

if (head == null) {

head = newNode;

} else {

Node temp = head;

while (temp.next != null) {

temp = temp.next;

}

temp.next = newNode;

}

}

public void printList() {

Node temp = head;

while (temp != null) {

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

temp = temp.next;

}

}

}

**Q5. Remove duplicates from a linked list**

import java.util.HashSet;

public static void removeDuplicates(ListNode head) {

if (head == null) return;

HashSet<Integer> seen = new HashSet<>();

ListNode current = head;

ListNode prev = null;

while (current != null) {

if (seen.contains(current.val)) {

prev.next = current.next;

} else {

seen.add(current.val);

prev = current;

}

current = current.next;

}

}

**Q6. Write down the differences between ArrayList and LinkedList**

**Underlying Data Structure**:

* ArrayList: Resizable array.
* LinkedList: Doubly linked list.

**Performance**:

* ArrayList: Fast random access (O(1)), slow insertions/deletions (O(n)) except at the end.
* LinkedList: Slow random access (O(n)), fast insertions/deletions (O(1)) if node reference is known.

**Memory Overhead**:

* ArrayList: Less memory overhead as it stores only data elements.
* LinkedList: More memory overhead due to storage of node pointers (next and prev).

**Iteration**:

* Both support efficient iteration, but ArrayList can be more cache-friendly due to contiguous memory allocation.

**Usage**:

* ArrayList: Preferred for read-heavy scenarios where frequent access to elements by index is required.
* LinkedList: Preferred for scenarios involving frequent insertions/deletions.

**Q7. Sort a List using Bubble Sort technique, Also explain the**

**working of the bubble sort.**

public static void bubbleSort(int[] arr) {

int n = arr.length;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// swap arr[j] and arr[j+1]

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

}

}

}

}

// Bubble Sort Explanation:

// - The algorithm passes through the list multiple times.

// - During each pass, it compares adjacent elements and swaps them if they are in the wrong order.

// - This process is repeated until the list is sorted.

// - With each complete pass, the largest unsorted element is moved to its correct position.

**Q8. Write a program to find out how many swaps are required in a**

**given list to be sorted. (Using bubble sort)**

public static int bubbleSortWithSwaps(int[] arr) {

int n = arr.length;

int swapCount = 0;

for (int i = 0; i < n - 1; i++) {

for (int j = 0; j < n - i - 1; j++) {

if (arr[j] > arr[j + 1]) {

// swap arr[j] and arr[j+1]

int temp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = temp;

swapCount++;

}

}

}

return swapCount;

}

**Q9. Write a menu driven program to store phone numbers of**

**persons using Set data structure.**

import java.util.HashSet;

import java.util.Scanner;

import java.util.Set;

public class PhoneBook {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

Set<String> phoneNumbers = new HashSet<>();

int choice;

do {

System.out.println("1. Add phone number");

System.out.println("2. Display all phone numbers");

System.out.println("3. Exit");

System.out.print("Enter your choice: ");

choice = scanner.nextInt();

scanner.nextLine(); // Consume newline

switch (choice) {

case 1:

System.out.print("Enter phone number: ");

String phoneNumber = scanner.nextLine();

phoneNumbers.add(phoneNumber);

break;

case 2:

System.out.println("Phone Numbers:");

for (String number : phoneNumbers) {

System.out.println(number);

}

break;

case 3:

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

break;

default:

System.out.println("Invalid choice. Try again.");

}

} while (choice != 3);

scanner.close();

}

}

**Q10. Write down the principle and working of set data structure**

**Principle**: A Set is a collection that contains no duplicate elements. It models the mathematical set abstraction.

**Working**: Sets are typically implemented using hash tables. Elements are stored based on hash codes which allow for efficient insertion, deletion, and lookup operations. Common implementations include HashSet, LinkedHashSet, and TreeSet.

**Q11. Write all the methods available to print a linked list.**

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

this.next = null;

}

}

public class LinkedListMethods {

public static void printListIterative(ListNode head) {

ListNode temp = head;

while (temp != null) {

System.out.print(temp.val + " ");

temp = temp.next;

}

System.out.println();

}

public static void printListRecursive(ListNode head) {

if (head == null) return;

System.out.print(head.val + " ");

printListRecursive(head.next);

}

public static void printListWithStringBuilder(ListNode head) {

StringBuilder sb = new StringBuilder();

ListNode temp = head;

while (temp != null) {

sb.append(temp.val).append(" ");

temp = temp.next;

}

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

}

public static void printListUsingStream(ListNode head) {

ListNode temp = head;

Stream.Builder<Integer> builder = Stream.builder();

while (temp != null) {

builder.add(temp.val);

temp = temp.next;

}

String result = builder.build()

.map(String::valueOf)

.collect(Collectors.joining(" "));

System.out.println(result);

}

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(2);

head.next.next = new ListNode(3);

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

printListIterative(head);

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

printListRecursive(head);

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

printListWithStringBuilder(head);

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

printListUsingStream(head);

}

}