Task 2: Linked List Middle Element Search

You are given a singly linked list. Write a function to find the middle element without using any extra space and only one traversal through the linked list.

SOLUTION :

```
package com.assignment.day1;

public class LinkedList {

private Node head;

private Node tail;

private int length;

class Node {

int value;

Node next;

public Node(int value) {

super();

this.value = value;

}

}

public LinkedList(int value) {

super();

Node newNode = new Node(value);

// System.out.println("Node:"+ newNode);

head = newNode;

tail = newNode;

length = 1;

}

public void getHead() {

System.out.println("Head:" + head.value);
```

```
}
public void getTail() {
System.out.println("Tail: " + tail.value);
}
public void getLength() {
System.out.println("Length :" + length);
}
public void printList() {
Node temp = head;
System.out.println("\n");
getHead();
getTail();
getLength();
System.out.println("Items in list:");
while (temp != null) {
System.out.print("-->" + temp.value + "\t");
temp = temp.next;
}
}
public void append(int value) {
Node newNode = new Node(value);
if (length == 0) {
head = newNode;
tail = newNode;
} else {
tail.next = newNode;
tail = newNode;
}
```

```
length++;
}
public Node removeLast() {
if (length == 0) {
return null;
}
Node temp = head;
Node pre = head;
while (temp.next != null) {
pre = temp;
temp = temp.next;
}
tail = pre;
tail.next = null;
length--;
if (length == 0) {
head = null;
tail = null;
}
return temp;
}
public void prepend(int value) {
Node newNode = new Node(value);
if (length == 0) {
head = newNode;
tail = newNode;
} else {
newNode.next = head;
```

```
head = newNode;
}
length++;
}
public Node removeFirst() {
if (length == 0) {
return null;
}
Node temp = head;
head = head.next;
temp.next = null;
length--;
if (length == 0) {
tail = null;
}
return temp;
}
public Node get(int index) {
if (index < 0 || index >= length) {
return null;
}
Node temp = head;
for (int i = 0; i < index; i++) {
temp = temp.next;
}
return temp;
}
public boolean set(int index, int value) {
```

```
Node temp = get(index);
if (temp != null) {
temp.value = value;
return true;
}
return false;
}
public boolean insert(int index, int value) {
if (index < 0 || index >= length) {
return false;
}
if (index == 0) {
prepend(value);
;
return true;
}
if (index == length) {
append(value);
return true;
}
Node newNode = new Node(value);
Node temp = get(index - 1);
newNode.next = temp.next;
temp.next = newNode;
length++;
return true;
}
public Node findMiddleElement() {
```

```
if (head == null) {
return null;
}
int count = 0;
Node temp = head;
while (temp != null) {
count++;
temp = temp.next;
}
temp = head;
for (int i = 0; i < count / 2; i++) {
temp = temp.next;
}
return temp;
}
public static void main(String[] args) {
LinkedList myll = new LinkedList(11);
myll.printList();
myll.append(3);
myll.append(23);
myll.append(7);
myll.printList();
System.out.println("Removed :" + myll.removeLast().value);
myll.printList();
myll.prepend(1);
myll.printList();
System.out.println("Removed :" + myll.removeFirst().value);
;
```

```
myll.printList();

System.out.println("Item at index 1 is " + myll.get(1).value);

System.out.println("Replace item at index 1 :" + myll.set(1, 33));

myll.printList();

System.out.println("\n Insert between 1 and 2 :" + myll.insert(2, 20));

myll.printList();

System.out.println("Middle element is: " + myll.findMiddleElement().value);

}

}
```

OUTPUT:

```
Head:11
Tail: 23
Length :3
Items in list:
-->11   -->3    -->23   Item at index 1 is  3
Replace item at index 1 :true


Head:11
Tail: 23
Length :3
Items in list:
-->11   -->33   -->23
 Insert between 1 and 2 :true


Head:11
Tail: 23
Length :4
Items in list:
-->11   -->33   -->20   -->23   Middle element is: 20
```

Task 3: Queue Sorting with Limited Space

You have a queue of integers that you need to sort. You can only use additional space equivalent to one stack. Describe the steps you would take to sort the elements in the queue.

SOLUTION :

Initialize: Start with an empty stack and the numbers in the queue.

Dequeue: Remove the front element from the queue.

Compare: Compare this element with the top of the stack.

- If the stack is empty or the dequeued element is greater than the top element of the stack, push it onto the stack.
- If the dequeued element is smaller than the top element of the stack, then we pop the elements from the stack and enqueue them at the end of the queue until we reach a state where the dequeued element is larger than the top element of the stack or the stack is empty. Then, we push the dequeued element onto the stack.

Repeat: Repeat Dequeue and compare until the queue is empty.

Transfer: Once the queue is empty, pop the elements from the stack and enqueue them into the queue. Now, the queue is sorted in descending order.

Reverse: To get the queue in ascending order, you can repeat the above process one more time.

This algorithm works because the stack is a LIFO (Last In First Out) data structure and the queue is a FIFO (First In First Out) data structure. By cleverly dequeuing, enqueuing, pushing, and popping elements between these two data structures, we can effectively sort the queue.

```
package com.assignment.day1;


import java.util.LinkedList;

import java.util.Queue;

import java.util.Stack;


public class QueueSorter {


	public static void main(String[] args) {

		Queue<Integer> queue = new LinkedList<>();

		queue.add(3);

		queue.add(1);

		queue.add(5);

		queue.add(2);

		queue.add(4);
```

```
        sortQueue(queue);

        System.out.println("Sorted Queue:");
        while (!queue.isEmpty()) {
            System.out.print(queue.poll() + " ");
        }
}

public static void sortQueue(Queue<Integer> queue) {
        if (queue == null || queue.isEmpty()) {
            return;
        }

        Stack<Integer> stack = new Stack<>();

        // Step 1: Divide the queue recursively
        while (!queue.isEmpty()) {
            int min = queue.poll();

            // Remove elements from stack and enqueue them into the queue if they are
            // smaller than 'min'
            while (!stack.isEmpty() && stack.peek() < min) {
                queue.add(stack.pop());
            }

            // Push 'min' onto the stack
            stack.push(min);
        }

        // Step 2: Merge the partitions back together in sorted order
        while (!stack.isEmpty()) {
            queue.add(stack.pop());
```

```
            }
        }
}
```

Output :

```
Sorted Queue:
1 2 3 4 5
```

Task 4: Stack Sorting In-Place

You must write a function to sort a stack such that the smallest items are on the top. You can use an additional temporary stack, but you may not copy the elements into any other data structure such as an array. The stack supports the following operations: push, pop, peek, and isEmpty.

SOLUTION :

```
package com.assignment.day1;

import java.util.Stack;

public class StackSort {

public static void sortStack(Stack<Integer> stack) {

Stack<Integer> tempStack = new Stack<>();

while (!stack.isEmpty()) {

int temp = stack.pop();

while (!tempStack.isEmpty() && tempStack.peek() > temp) {

stack.push(tempStack.pop());

}

tempStack.push(temp);

}

while (!tempStack.isEmpty()) {
```

```
stack.push(tempStack.pop());

}

}

public static void main(String[] args) {

Stack<Integer> stack = new Stack<>();

stack.push(34);

stack.push(3);

stack.push(31);

stack.push(98);

stack.push(92);

stack.push(23);

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

sortStack(stack);

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

}

}
```

Output :

```
Original Stack: [34, 3, 31, 98, 92, 23]
Sorted Stack: [98, 92, 34, 31, 23, 3]
```

Task 5: Removing Duplicates from a Sorted Linked List

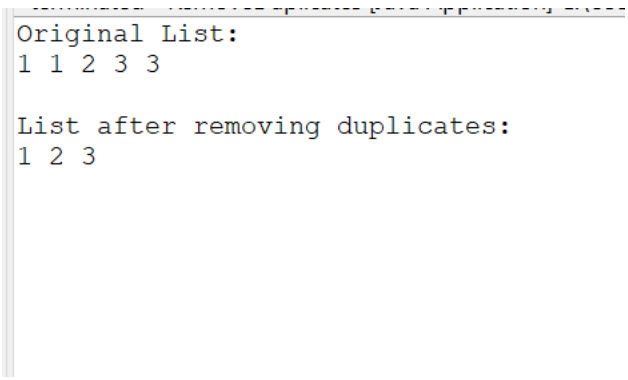
A sorted linked list has been constructed with repeated elements. Describe an algorithm to remove all duplicates from the linked list efficiently.

SOLUTION:

```
package com.assignment.day1;

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

this.next = null;

}

}

public class RemoveDuplicates {

public static void main(String[] args) {

ListNode head = new ListNode(1);

head.next = new ListNode(1);

head.next.next = new ListNode(2);

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

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

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

printList(head);

removeDuplicates(head);

System.out.println("\nList after removing duplicates:");

printList(head);

}

public static void removeDuplicates(ListNode head) {

ListNode current = head;

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

if (current.val == current.next.val) {

current.next = current.next.next;

} else {
```

```
current = current.next;

}

}

}

public static void printList(ListNode head) {

ListNode current = head;

while (current != null) {

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

current = current.next;

}

System.out.println();

}

}
```

OUTPUT:



Task 6: Searching for a Sequence in a Stack

Given a stack and a smaller array representing a sequence, write a function that determines if the sequence is present in the stack. Consider the sequence present if, upon popping the elements, all elements of the array appear consecutively in the stack.

Solution :

```
package com.assignment.day1;
```

```
import java.util.Stack;

public class StackSequence {

public static void main(String[] args) {

Stack<Integer> stack = new Stack<>();

int[] sequence = {4, 3, 2};

stack.push(1);

stack.push(2);

stack.push(3);

stack.push(4);

System.out.println("Stack data:");

displayStack(stack);

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

displaySequence(sequence);

boolean present = isSequencePresent(stack, sequence);

System.out.println("\nSequence present: " + present);

}

public static boolean isSequencePresent(Stack<Integer> stack, int[]
sequence) {

Stack<Integer> tempStack = new Stack<>();

for (int num : sequence) {

boolean found = false;

while (!stack.isEmpty()) {

int top = stack.pop();

tempStack.push(top);

if (top == num) {

found = true;

break;

}
```

```
}
if (!found) {
return false;
}
while (!tempStack.isEmpty()) {
stack.push(tempStack.pop());
}
}
return true;
}
public static void displaySequence(int[] sequence) {
int number = 0;
for (int num : sequence) {
number = number * 10 + num;
}
System.out.print(number);
}
public static void displayStack(Stack<Integer> stack) {
Stack<Integer> tempStack = new Stack<>();
while (!stack.isEmpty()) {
int element = stack.pop();
tempStack.push(element);
System.out.print(element + " ");
}
while (!tempStack.isEmpty()) {
stack.push(tempStack.pop());
}
}
```

```
}
```

OUTPUT:

```
Stack data:
4 3 2 1
Sequence: 432
Sequence present: true
```

Task 7: Merging Two Sorted Linked Lists

You are provided with the heads of two sorted linked lists. The lists are sorted in ascending order. Create a merged linked list in ascending order from the two input lists without using any extra space (i.e., do not create any new nodes).

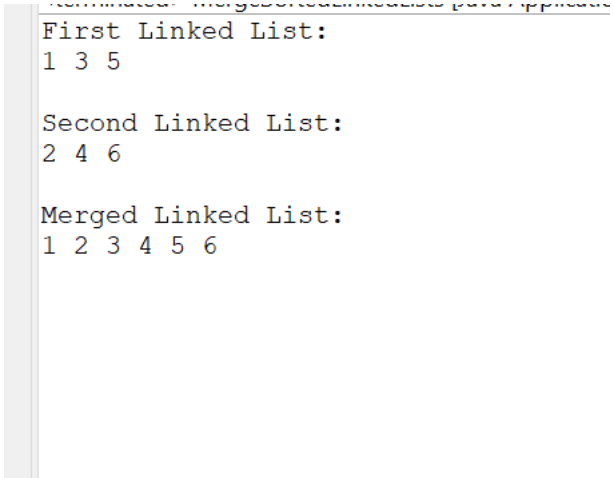
SOLUTION :

```
package com.assignment.day1;

class LinkedListNode {

int val;

LinkedListNode next;

LinkedListNode(int val) {

this.val = val;

this.next = null;

}

}

public class MergeSortedLinkedLists {

public static void main(String[] args) {

LinkedListNode head1 = new LinkedListNode(1);

head1.next = new LinkedListNode(3);

head1.next.next = new LinkedListNode(5);
```

```
LinkedListNode head2 = new LinkedListNode(2);

head2.next = new LinkedListNode(4);

head2.next.next = new LinkedListNode(6);

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

printList(head1);

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

printList(head2);

LinkedListNode mergedHead = mergeLists(head1, head2);

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

printList(mergedHead);

}

public static LinkedListNode mergeLists(LinkedListNode head1,
LinkedListNode head2) {

LinkedListNode dummy = new LinkedListNode(0);

LinkedListNode current = dummy;

while (head1 != null && head2 != null) {

if (head1.val < head2.val) {

current.next = head1;

head1 = head1.next;

} else {

current.next = head2;

head2 = head2.next;

}

current = current.next;

}

if (head1 != null) {

current.next = head1;

} else {
```

```
current.next = head2;

}

return dummy.next;

}

public static void printList(LinkedListNode head) {

while (head != null) {

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

head = head.next;

}

System.out.println();

}

}
```

OUTPUT :



Task 8: Circular Queue Binary Search

Consider a circular queue (implemented using a fixed-size array) where the elements are sorted but have been rotated at an unknown index. Describe an approach to perform a binary search for a given element within this circular queue.

SOLUTION :

```
function searchInRotatedCircularQueue(queue, target):
    // Find the pivot index
    pivot_index = findPivotIndex(queue)

    // Determine the search range based on the target element and pivot index
    if target >= queue[0] && target <= queue[pivot_index]:
        // Search in the left part of the circular queue
        return binarySearch(queue, target, 0, pivot_index)
    else:
        // Search in the right part of the circular queue
        return binarySearch(queue, target, pivot_index + 1, size - 1)

function findPivotIndex(queue):
    low = 0
    high = size - 1
    while low < high:
        mid = (low + high) / 2
        if queue[mid] < queue[high]:
            high = mid
        else:
            low = mid + 1
    return low

function binarySearch(queue, target, low, high):
    while low <= high:
        mid = (low + high) / 2
        if queue[mid] == target:
            return mid
        else if queue[mid] < target:
```

low = mid + 1

else:

high = mid - 1

return -1 // Target not found

```
package com.assignment.day1;

public class CircularQueueBinarySearch {

public static void main(String[] args) {

int[] queue = {6, 7, 8, 9, 10, 1, 2, 3, 4, 5};

int target = 3;

int index = searchInRotatedCircularQueue(queue, target);

if (index != -1) {

System.out.println("Element " + target + " found at index " + index);

} else {

System.out.println("Element " + target + " not found in the circular
queue.");

}

}

public static int searchInRotatedCircularQueue(int[] queue, int target) {

int pivotIndex = findPivotIndex(queue);

if (target >= queue[0] && target <= queue[pivotIndex]) {

return binarySearch(queue, target, 0, pivotIndex);

} else {

return binarySearch(queue, target, pivotIndex + 1, queue.length - 1);

}

}

public static int findPivotIndex(int[] queue) {

int low = 0;
```

```
int high = queue.length - 1;

while (low < high) {

int mid = (low + high) / 2;

if (queue[mid] < queue[high]) {

high = mid;

} else {

low = mid + 1;

}

}

return low;

}

public static int binarySearch(int[] queue, int target, int low, int high)
{

while (low <= high) {

int mid = (low + high) / 2;

if (queue[mid] == target) {

return mid;

} else if (queue[mid] < target) {

low = mid + 1;

} else {

high = mid - 1;

}

}

return -1;

}

}
```

OUTPUT :