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.

ANS:
```
public class linkedlist_middleElement {

	private Node head;

	private Node tail;

	private int length;


	static class Node{

		int value;

		Node next;


		public Node(int value) {

			super();

			this.value=value;

		}

	}

	public linkedlist_middleElement(int value) {

		super();

		Node newNode=new Node(value);

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

		head=newNode;

		tail=newNode;

		length=1;

	}

	public Node getHead() {
  //System.out.println("Head:"+head.value);
  return head;
}
public void setHead(Node n) {
              this.head=n;
}
public Node getTail() {
                //System.out.println("Tail:"+tail.value);
              return tail;
              }
public void setTail(Node n) {
              this.tail=n;
}
public int getLength() {
              //  System.out.println("Length:"+length);
                return length;
              }
public void setLength(int length) {
              this.length=length;
}
public void printList() {
              Node temp=head;
              if(length>0) {
              //getHead();
                             System.out.println("Head:"+head.value);
```

```
        //getTail();
                System.out.println("Tail:"+tail.value);
        //getLength();
                System.out.println("Length:"+length);
        }
        System.out.print("HEAD");
        while(temp!=null) {
                System.out.print("--->"+temp.value);
                temp=temp.next;
        }
        System.out.println("--->NULL");
}
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(head==null) {
                return null;
        }

        else {
                Node temp=head;
                head=head.next;
                temp.next=null;
                length--;
                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 void addAtIndex(int index,int value) {

        Node newNode=new Node(value);

        if(index>length || head==null) {

                this.append(value);

        }

        else {

                Node temp=head;


                int count=0;

                while(count!=index-2) {

                        temp=temp.next;

                        count++;

                }
```

```
                    Node next=temp.next;

                    temp.next=newNode;

                    newNode.next=next;


          }

          length++;


}

public Node deleteAtIndex(int index) {

          if(index>length ||head==null) {

                    return null;

          }

          else if(index==length) {

                    this.removeLast();

          }

          else if(index==1) {

                    this.removeFirst();

          }

          else {

                    Node temp=head;

                    Node node=this.get(index-1);

                    Node prev=this.get(index-2);

                    prev.next=node.next;

                    node.next=null;

                    length--;
```

```
                return temp;


        }

        return null;

}

public Node middleElement() {

        if(head==null) {

                return null;

        }

        int index=(int)(length/2);

        Node node=this.get(index);

        if(length%2==0) {

                node=this.get(index-1);

        }

        return node;

}

 public static void main(String[] args) {

        linkedlist_middleElement ll=new linkedlist_middleElement(33);

        ll.append(44);

        ll.append(55);

        ll.append(66);

        ll.prepend(22);

        ll.prepend(11);

        ll.prepend(0);

        ll.printList();
```

```
                //System.out.println(ll.removeLast().value);

                //System.out.println(ll.removeLast().value);

                //ll.printList();

                System.out.println(ll.removeFirst().value);

                ll.printList();

                ll.addAtIndex(3, 43);

                ll.printList();

                System.out.println(ll.deleteAtIndex(3).value);

                ll.printList();

                System.out.println(ll.middleElement().value);

        }


}
```

#code by RUBY

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.

ANS: import java.util.LinkedList;

```
import java.util.Queue;

import java.util.Stack;


public class QueueSort_withonestack {

    public static void sortQueue(Queue<Integer> queue) {

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

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

```
        int current = queue.poll();

        while (!stack.isEmpty() && stack.peek() < current) {

            queue.offer(stack.pop());

        }

       stack.push(current);

    }

    while (!stack.isEmpty()) {

        queue.offer(stack.pop());

    }

}


public static void main(String[] args) {

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

    queue.offer(289);

    queue.offer(96);

    queue.offer(64);

    queue.offer(104);

    queue.offer(44);

    queue.offer(89);


    System.out.println("Queue before sorting: " + queue);


    sortQueue(queue);


    System.out.println("Queue after sorting: " + queue);
    }
}
/*-Dequeue the first element from the queue.
  -Push it onto the stack.
  -For each subsequent element in the queue:
  -Dequeue the next element.
  -While the stack is not empty and the top element of the stack is greater than the dequeued element,
pop the stack and enqueue the popped element back into the queue.
  -Push the dequeued element onto the stack.
  -After processing all elements in the queue, pop all elements from the stack and enqueue them back
into the queue.
  -This will sort the elements in ascending order.*/


//code-by-RUBY
```

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.

ANS: import java.util.Stack;

```
public class Stacksort {
            public static void sortStack(Stack<Integer> stack) {
                Stack<Integer> temp = new Stack<>();
                while (!stack.isEmpty()) {
        }
            int current = stack.pop();
            while (!temp.isEmpty() && temp.peek() < current) {
                stack.push(temp.pop());
            }
           temp.push(current);
        }
        while (!temp.isEmpty()) {
            stack.push(temp.pop());
        }
    }
  public static void main(String[] args) {
        Stack<Integer> stack=new Stack<>();
        stack.push(227);
        stack.push(24);
        stack.push(27);
        stack.push(127);
        stack.push(7);
        stack.push(733);

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

        sortStack(stack);

        System.out.println("Stack after sorting: " + stack);
    }
```

```
}

/*-Pop the first element from the main-stack.

-Push it onto the temp-stack.

-For each subsequent element in the main-stack:

-Pop the next element.

-While the temp-stack is not empty and the top element of the temp-stack is greater than the Popped element, pop the temp-stack and push the popped element back into the main-stack.

-Push the popped element onto the temp-stack.

-After processing all elements in the main-stack, pop all elements from the temp-stack and push them back into the main-stack.

-This will sort the elements in ascending order.*/



#code-by-RUBY
```

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.

ANS: linkedlist_middleElement.Node;

```
public class Duplicate_remover_from_linkedlist {

        public static void removeDuplicates(linkedlist_middleElement ll) {

    if (ll.getHead() == null) return;


    Node current = ll.getHead();
```

```
        int length=ll.getLength();
        while (current != null && current.next != null) {
            if (current.value == current.next.value) {
                current.next = current.next.next;
               length--;


            } else {
                current = current.next;
            }
        }
        ll.setLength(length);
    }


public static void main(String[] args) {
              linkedlist_middleElement ll=new linkedlist_middleElement(10);
              ll.append(20);
              ll.append(20);
              ll.append(30);
              ll.append(40);
              ll.append(40);
              ll.append(50);
              ll.append(60);
              ll.append(60);
              System.out.println("list with duplicates");
              ll.printList();
        removeDuplicates(ll);

        System.out.println("list without duplicates");

        ll.printList();

    }


}

//code-by-RUBY
```

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.

ANS: import java.util.Stack;

```
public class SequenceInStack {

        public static boolean sequencePresentOrNot(Stack<Integer> stack,int[] arr) {

                int max=-111111;

                if(stack.size()<arr.length || stack.size()==0)

                        return false;

                while(!stack.isEmpty()) {

                        while(stack.pop()!=arr[0]) {continue;}

                        for(int i=1;i<arr.length;i++) {


                                if(arr[i]==stack.pop()) {

                                        max=Math.max(max,i);
                                }


                                else {
                                        break;
                                }
                        }
                        if(max==arr.length-1)
                                return true;
                                        }
                return false;




        }
        public static void main(String[] args) {
                //Not taking user input here
                Stack<Integer> stack=new Stack<>();
                stack.push(50);
                stack.push(40);
                stack.push(30);
                stack.push(20);
                stack.push(10);
                int arr[]= {20,30,40};
                boolean b=sequencePresentOrNot(stack, arr);
                if(b) {
```

```
                    System.out.println("Sequence is present");

                }else

                System.out.println("Sequence not present");

        }


}
```

//code-by-RUBY

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).

ANS: public class MergeTwoSortedList {

```
        public static linkedlist_middleElement mergeLists(linkedlist_middleElement
list1,linkedlist_middleElement list2) {

    if (list1.getHead() == null) return list2;

    if (list2.getHead() == null) return list1;


    Node dummy =new Node(0);

    Node tail = dummy;


    Node l1 = list1.getHead();

    Node l2 = list2.getHead();



    while (l1 != null && l2 != null) {
```

```
    if (l1.value <= l2.value) {
        tail.next = l1;
        l1 = l1.next;
    } else {
        tail.next = l2;
        l2 = l2.next;
    }
    tail = tail.next;
}


if (l1 != null) {
    tail.next = l1;
} else {
    tail.next = l2;
}
while(tail.next!=null) {
      tail=tail.next;
}
list1.setTail(tail);


list1.setHead(dummy.next);
list1.setLength(list1.getLength()+list2.getLength());
return list1;
```

```
   }

        public static void main(String[] args) {

                linkedlist_middleElement ll1=new linkedlist_middleElement(10);

                ll1.append(20);

                ll1.append(40);

                ll1.append(70);

                ll1.append(80);

                linkedlist_middleElement ll2=new linkedlist_middleElement(30);

                ll2.append(50);

                ll2.append(60);

                ll2.append(90);

                ll2.append(100);

                System.out.println("lists befor merging : ");

                ll1.printList();

                ll2.printList();

                linkedlist_middleElement ll=mergeLists(ll1, ll2);

                System.out.println("lists after getting merged : ");

                ll.printList();

        }



}

//code-by-RUBY
```

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.

ANS:
```
public class CircularQueueBinarySearch {



 public static int circularQueueBinarySearch(int[] arr, int target) {

   int left = 0;

   int right = arr.length - 1;


   while (left <= right) {

     int mid = left + (right - left) / 2;



     if (arr[mid] == target) {

       return mid;

     }



     if (arr[left] <= arr[mid]) {

       if (target >= arr[left] && target < arr[mid]) {

         right = mid - 1;

       } else {
```

```
                left = mid + 1;
            }
        } else {
            if (target > arr[mid] && target <= arr[right]) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }
    }


    return -1;
}

public static void main(String[] args) {
    int[] arr = {10, 15, 20, 0, 5, 7 , 9};
    int target = 5;

    int result = circularQueueBinarySearch(arr, target);

    if (result != -1) {
        System.out.println("Element " + target + " found at index " + result);
    } else {
        System.out.println("Element " + target + " not found in the array");
        }
    }
}
//code-by-RUBY
```