Chapter 2 - Linked Lists

// **Remove Dups -** Write code to remove duplicates from an unsorted linked list? How would you solve this problem if a temporary buffer is not allowed?

```
public class LinkedList<Item> implements Iterable<Item> {

    private int n;
    private Node pre;    // sentinel node before first item
    private Node post;  // sentinel node after last item

    public LinkedList() {
        pre = new Node();
        post = new Node();
        pre.next = post;
        post.previous = pre;
    }

    public boolean isEmpty() {
        return n == 0;
    }

    public int size() {
        return n;
    }

    public void add(Item item) {
        Node last = post.previous;
        Node x = new Node();
        x.item = item;
        x.previous = last;
        x.next = post;
        post.previous = x;
        last.next = x;
        n++;
    }

    public ListIterator<Item> iterator() {
        return new LinkedListIterator();
    }
```

```java
private class LinkedListIterator implements ListIterator<Item> {
        private Node current = pre.next;
        private Node lastAccessed = null;
        private int index = 0;

        public boolean hasNext() {
                return index < n;
        }

        public boolean hasPrevious() {
                return index > 0;
        }

        public int nextIndex() {
                return index;
        }

        public int previousIndex() {
                return index - 1;
        }

        public Item next() {
                if (!hasNext()) {
                        throw new NoSuchElementException();
                }
                lastAccessed = current;
                Item item = current.item;
                current = current.next;
                ++index;
                return item;
        }

        public Item previous() {
                if (!hasPrevious()) {
                        throw new NoSuchElementException();
                }
                current = current.previous;
                index--;
                lastAccessed = current;
                return current.item;
        }

        public void set(Item item) {
```

```
            // replace item for last accessed node
            if (lastAccessed == null) {
                throw new IllegalStateException();
            }
            lastAccessed.item = item;
        }

        public void remove() {
            // remove last element that was accessed by next() or previous()
            Node prev = lastAccessed.previous;
            Node next = lastAccessed.next;
            prev.next = next;
            next.previous = prev;
            n--;
            if (current == lastAccessed) {
                current = next;
            } else {
                index--;
            }
            lastAccessed = null;
        }

        public void add(Item item) {
            Node x = current.previous;
            Node y = new Node();
            Node z = current;

            y.item = item;
            x.next = y;
            y.next = z;
            z.previous = y;
            y.previous = x;
            n++;
            index++;
            lastAccessed = null;
        }
    }

    private class Node {
        private Item item;
        private Node next;
        private Node previous;
    }
```

```
}
```

// **Solution A** - using a set → runs in O(N) time

```
public class QuestionA {

	public static void removeDuplicates(LinkedList<Integer> list) {
		HashSet<Integer> marked = new HashSet();
		ListIterator<Integer> iterator = list.iterator();

		while (iterator.hasNext()) {
			Integer next = iterator.next();
			if (marked.contains(next)) {
				iterator.remove();
			} else {
				marked.add(next);
			}
		}
	}
}
```

// **Solution B** - without using a buffer
// You can use a second iterator (runner) for every element look for it in the list
// This solution uses O(1) space but runs in O(N^2) time

// **K-th to last** - Implement an algorithm to find the kth to last element of a singly linked list

// If we know the size of the linked list we just iterate through the list to position **(size - k)**

// Using recursion → takes O(N) space due to the recursive calls

```
public class LinkedList {

	public int printKthToLastIndex(Node x, int k) {
		if (x == null) {
			return 0;
		}
		int index = printKthToLastIndex(x.next, k) + 1;
		if (index == k) {
			System.out.println(x.item);
		}
		return index;
	}
}
```

// **Iterative solution** → using two pointers, p1 and p2. Move p1 k nodes into the list. Then move the pointers at the same pace. When p1 hits the end of the list → p2 will be the kth last element in the list

// **Delete Middle Node** → implement an algorithm to delete a node in the middle of a singly linked list, given only access to that node

```
public class LinkedList {

    public boolean deleteNode(Node x) {
        if (x == null || x.next == null) {
            return false;
        }
        Node next = x.next;
        x.item = next.item;
        x.next = next.next;
        return true;
    }
}
```

// **Partition** → write code to partition a linked list around a value x

```
public class LinkedList {

    public Node partition(Item x) {
        Node head = pre.next;
        Node tail = pre.next;
        Node current = pre.next;

        while (current != null) {
            Node next = current.next;
            if (current.item < item) {
                // Insert node at head
                current.next = head;
                head.previous = current;
                head = current;
            } else {
                // Insert node at tail
                tail.next = current;
                current.previous = tail;
                tail = current;
            }
```

```
                current = next;
            }
            pre.next = head;
            head.previous = pre;

            post.previous = tail;
            tail.next = post;
        }
}
```

// **Sum Lists** → Sum two numbers stored in linked lists in reverse order

```
public class LinkedListNode {
        public LinkedListNode next;
        public LinkedListNode previous;
        public LinkedListNode last;
        public int data;

        public LinkedListNode () {}

        public LinkedListNode(int data) {
            this.data = data;
        }

        public LinkedListNode(int data, LinkedListNode p, LinkedListNode n) {
            this.data = data;
            setNext(n);
            setPrevious(p);
        }

        public void setNext(LinkedListNode n) {
            next = n;
            if (this == last) {
                last = n;
            }
            if (n != null && n.previous != this) {
                n.setPrevious(this);
            }
        }

        public void setPrevious(LinkedListNode p) {
            previous = p;
            if (p != null && p.next != this) {
```

```
			p.setNext(this);
		}
	}

	public String printForward() {
		if (next != null) {
			return data + "->" + next.printForward();
		} else {
			return ((Integer) data).toString();
		}
	}
}

public class QuestionA {

	public static LinkedListNode addLists(LinkedListNode a, LinkedListNode b, int carry) {
		if (a == null && b == null && carry == 0) {
			return null;
		}

		LinkedListNode result = new LinkedListNode();
		int value = carry;

		if (a != null) {
			value += a.data;
		}

		if (b != null) {
			value += b.data;
		}

		result.data = value % 10;

		if (a != null || b != null) {
			LinkedListNode next = addLists(
				a == null ? null : a.next,
				b == null ? null : b.next,
				value / 10
			);
			result.setNext(next);
		}

		return result;
```

```
        }
}
```

// **Palindrome** - Implement a function to check if a linked list is a palindrome

// **First solution** is to reverse and compare the linked lists → if they are equal then the list is a palindrome → note that when comparing the lists we can only compare the first half (if we know the lngth)

// **Second Solution** → using a stack and the fast and slow runner technique

```
public class QuestionA {

        public static boolean isPalindrome(LinkedListNode head) {
                LinkedListNode fast = head;
                LinkedListNode slow = head;

                Stack<Integer> stack = new Stack<>();
                while (fast != null && fast.next != null) {
                        stack.push(slow.data);
                        slow = slow.next;
                        fast = fast.next.next;
                }

                // odd number of elements - skipping the middle element
                if (fast != null) {
                        slow = slow.next;
                }

                while (slow != null) {
                        int top = stack.pop();

                        if (top != slow.data) {
                                return false;
                        }
                        slow = slow.next;
                }
                return true;
        }
}
```

// **Intersection** → given two singly linked lists determine if they intersect (by reference not value)
// Determine if there is an intersection → hash table or both lists have the same last node

```
// Find the intersecting node

public class Question {

	class Result {
		public LinkedListNode tail;
		public int size;

		public Result(LinkedListNode tail, int size) {
			this.tail = tail;
			this.size = size;
		}
	}

	public static LinkedListNode findIntersection(LinkedListNode a, LinkedListNode b) {
		if (a == null || b == null) {
			return null;
		}

		// get tail and sizes
		Result resultA = getTailAndSize(a);
		Result resultB = getTailAndSize(b);

		// if different tail nodes we have no intersection
		if (resultA.tail != resultB.tail) {
			return null;
		}

		// set pointers to the start of each linked list
		LinkedListNode shorter = resultA.size < resultB.size ? a : b;
		LinkedListNode longer = resultA.size < resultB.size ? b : a;

		// Advance pointer for longer list by k positions
		longer = getKthNode(longer, Math.abs(resultA.size - resultB.size));

		while (shorter != longer) {
			shorter = shorter.next;
			longer = longer.next;
		}

		return longer;
	}
```

```
	private static Result getTailAndSize(LinkedListNode list) {
		if (list == null) {
			return null;
		}

		int size = 1;
		LinkedListNode current = list;
		while (current.next != null) {
			size++;
			current = current.next;
		}

		return new Result(current, size);
	}

	private static LinkedListNode getKthNode(LinkedListNode head, int k) {
		LinkedListNode current = head;

		while (k > 0 && current != null) {
			current = current.next;
			k--;
		}
		return current;
	}
}
```

// **Loop detection** → given a circular linked list implement an algorithm that returns the node at the beginning of the loop

```
public class Question {

	public static LinkedListNode findBeginningOfLoop(LinkedListNode head) {
		LinkedListNode slow = head;
		LinkedListNode fast = head;

		while (fast != null && fast.next != null) {
			slow = slow.next;
			fast = fast.next.next;  // moving at twice the speed
			if (slow == fast) {
				// Collision
				break;
			}
		}
```

```
		if (fast == null || fast.next == null) {
			// no meeting point - therefore no loop (reached the end)
			return null;
		}

		// move slow to head - keep fast at meeting point
		slow = head;
		// each are k steps from the loop start
		while (slow != fast) {
			slow = slow.next;
			fast = fast.next;
		}

		return fast;
	}
}
```