

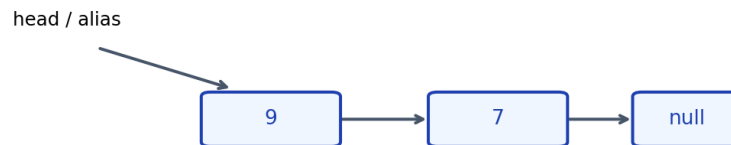
Solutions

3.1 Nodes and References

Reference example

```
class Node {  
    int value; Node next;  
    Node(int value, Node next) { this.value = value; this.next = next; }  
}  
Node head = new Node(4, new Node(7, null));  
Node alias = head;  
alias.value = 9;  
System.out.println(head.value + ":" + head.next.value);
```

References and a linked chain



After the value update, both references observe the same first node.

3.1.1 - Explain

After `alias = head.next`, does assigning `alias = null` remove the second node from the list?

No. It only changes the local `alias` variable. The link stored in `head.next` still refers to that node.

3.1.2 - Explain

What condition ends a straightforward singly linked traversal?

The current reference becomes null after the last node, assuming the list has no cycle. A cycle requires different termination logic.

Solutions

3.2 Insertion and Removal

Reference example

```
class Node {
    int value; Node next;
    Node(int v, Node n) { value = v; next = n; }
}
Node head = new Node(2, new Node(5, null));
head = new Node(1, head);
head.next = head.next.next;
System.out.println(head.value + ":" + head.next.value);
```

3.2.1 - Explain

If p refers to the predecessor of a node to remove, what reassignment bypasses that node in a singly linked list?

When p and p.next are non-null, use p.next = p.next.next. Removing the head instead requires updating head.

3.2.2 - Write code

Build the singly linked chain 2 -> 5 -> 8, remove its head, and print the remaining values.

Removing the head requires assigning the head reference to its successor.

```
class Node {
    int value; Node next;
    Node(int v, Node n) { value = v; next = n; }
}
Node head = new Node(2, new Node(5, new Node(8, null)));
head = head.next;
for (Node p = head; p != null; p = p.next)
    System.out.print(p.value + " ");
```

Expected output

5 8

Solutions

3.3 Singly Versus Doubly Linked

Reference example

Singly linked: value + next
Doubly linked: previous + value + next
Known doubly linked node: can update both neighbors
Find a value: still may require a scan

3.3.1 - Explain

Which links must be updated when removing an interior doubly linked node x ?

Connect $x.previous.next$ to $x.next$ and $x.next.previous$ to $x.previous$, with separate handling for boundaries and any sentinel conventions.

3.3.2 - Explain

Does adding previous references make searching an unsorted linked list logarithmic?

No. Without an additional indexing structure, locating a value still requires a linear scan in the worst case.

Solutions

3.4 Edge Cases

Reference example

```
LinkedList<Integer> values = new LinkedList<Integer>();
values.addFirst(8);
System.out.println(values.removeFirst());
System.out.println(values.isEmpty());
```

3.4.1 - Explain

Why can removing the sole node leave a bug when tail is not reset?

The structure may claim to be empty through head while tail still refers to an old node, breaking later insertions or consistency checks.

3.4.2 - Write code

Use LinkedList to add and remove one value, then safely inspect the empty front with peekFirst.

The non-throwing peek reports null when no element exists.

```
LinkedList<String> q = new LinkedList<String>();
q.add("A"); q.removeFirst();
System.out.println(q.peekFirst());
```

Expected output

null