1:-- add last, get last , remove last

**package** com.example.demo;

**public** **class** LinkedListOpertion {

**static** **class** Node {

**int** data;

Node next;

}

**static** **class** LinkedList {

Node head;

Node tail;

**int** size;

**public** **int** size() {

**return** size;

}

**public** **void** addLast(**int** val) {

Node temp = **new** Node();

temp.data = val;

temp.next = **null**;

**if** (size == 0) {

head = tail = temp;

} **else** {

tail.next = temp;

tail = temp;

}

size++;

}

**public** **int** getLast() {

**if** (size == 0) {

System.***out***.println("list is empty");

**return** -1;

} **else** {

**return** tail.data;

}

}

**public** **void** removeLast() {

**if** (size == 0) {

System.***out***.println("list is empty");

} **else** **if** (size == 1) {

head = tail = **null**;

size = 0;

} **else** {

Node temp = head;

**for** (**int** i = 0; i < size - 2; i++) {

temp = temp.next;

}

tail = temp;

tail.next = **null**;

size--;

}

}

**public** **void** display() {

Node temp = head;

**while** (temp != **null**) {

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

temp = temp.next;

}

System.***out***.println();

}

}

**public** **static** **void** main(String[] args) {

LinkedList l1 = **new** LinkedList();

// add last

l1.addLast(2);

l1.addLast(3);

l1.addLast(4);

l1.display();

// get last

System.***out***.println("get last " + l1.getLast());

l1.removeLast();

System.***out***.println();

l1.display();

}

}

// output

2 3 4

get last 4

remove last

2 3

2:- add first , get first, remove first

**package** com.example.demo;

**public** **class** LinkedListOpertion {

**static** **class** Node {

**int** data;

Node next;

}

**static** **class** LinkedList {

Node head;

Node tail;

**int** size;

**public** **int** size() {

**return** size;

}

**public** **void** addLast(**int** val) {

Node temp = **new** Node();

temp.data = val;

temp.next = **null**;

**if** (size == 0) {

head = tail = temp;

} **else** {

tail.next = temp;

tail = temp;

}

size++;

}

**public** **void** addFirst(**int** val) {

Node temp = **new** Node();

temp.data = val;

temp.next = head;

head = temp;

**if** (size == 0) {

tail = temp;

}

size++;

}

**public** **int** getFirst() {

**if** (size == 0) {

System.***out***.println("linked list is empty");

**return** -1;

} **else** {

Node temp = head;

**int** val = temp.data;

**return** val;

}

}

**public** **void** removeFirst() {

**if** (size == 0) {

System.***out***.println("list is empty");

} **else** **if** (size == 1) {

head = tail = **null**;

size = 0;

} **else** {

head = head.next;

size--;

}

}

**public** **void** display() {

Node temp = head;

**while** (temp != **null**) {

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

temp = temp.next;

}

System.***out***.println();

}

}

**public** **static** **void** main(String[] args) {

LinkedList l1 = **new** LinkedList();

l1.addLast(2);

l1.addLast(3);

l1.addLast(4);

l1.display();

System.***out***.println("add first");

l1.addFirst(1);

l1.display();

System.***out***.println("get first " + l1.getFirst());

l1.removeFirst();

System.***out***.println("remove first");

l1.display();

}

}

2 3 4

add first

1 2 3 4

get first 1

remove first

2 3 4

3:-- add middle, get middle ,remove middle

**package** com.example.demo;

**public** **class** LinkedListOpertion {

**static** **class** Node {

**int** data;

Node next;

}

**static** **class** LinkedList {

Node head;

Node tail;

**int** size;

**public** **int** size() {

**return** size;

}

**public** **void** addLast(**int** val) {

Node temp = **new** Node();

temp.data = val;

temp.next = **null**;

**if** (size == 0) {

head = tail = temp;

} **else** {

tail.next = temp;

tail = temp;

}

size++;

}

**public** **void** removeLast() {

**if** (size == 0) {

System.***out***.println("list is empty");

} **else** **if** (size == 1) {

head = tail = **null**;

size = 0;

} **else** {

Node temp = head;

**for** (**int** i = 0; i < size - 2; i++) {

temp = temp.next;

}

tail = temp;

temp.next = **null**;

size--;

}

}

**public** **void** addFirst(**int** val) {

Node temp = **new** Node();

temp.data = val;

temp.next = head;

head = temp;

**if** (size == 0) {

tail = temp;

}

size++;

}

**public** **void** removeFirst() {

**if** (size == 0) {

System.***out***.println("List is empty");

} **else** **if** (size == 1) {

head = tail = **null**;

size = 0;

} **else** {

head = head.next;

size--;

}

}

**public** **void** addAt(**int** idx, **int** val) {

**if** (idx < 0 || idx > size) {

System.***out***.println("invalid argument");

} **else** **if** (idx == 0) {

addFirst(val);

} **else** **if** (idx == size) {

addLast(val);

} **else** {

Node node = **new** Node();

node.data = val;

Node temp = head;

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

temp = temp.next;

}

node.next = temp.next;

temp.next = node;

size++;

}

}

**public** **int** getAt(**int** idx) {

**if** (size == 0) {

System.***out***.println("List is empty");

**return** -1;

} **else** **if** (idx < 0 || idx >= size) {

System.***out***.println("Invalid arguments");

**return** -1;

} **else** {

Node temp = head;

**for** (**int** i = 0; i < idx; i++) {

temp = temp.next;

}

**return** temp.data;

}

}

**public** **void** removeAt(**int** idx) {

**if** (idx < 0 || idx >= size) {

System.***out***.println("invalid argument");

} **else** **if** (idx == 0) {

removeFirst();

} **else** **if** (idx == size - 1) {

removeLast();

} **else** {

Node temp = head;

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

temp = temp.next;

}

temp.next = temp.next.next;

size--;

}

}

**public** **void** display() {

Node temp = head;

**while** (temp != **null**) {

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

temp = temp.next;

}

System.***out***.println();

}

}

**public** **static** **void** main(String[] args) {

LinkedList l1 = **new** LinkedList();

l1.addLast(2);

l1.addLast(3);

l1.addLast(4);

l1.display();

l1.addAt(1, 10);

System.***out***.println("10 added at first index");

l1.display();

System.***out***.println("element get at index 3");

System.***out***.println(l1.getAt(3));

System.***out***.println("remove at index 1 means element 10");

l1.removeAt(1);

l1.display();

}

}

//output

2 3 4

10 added at first index

2 10 3 4

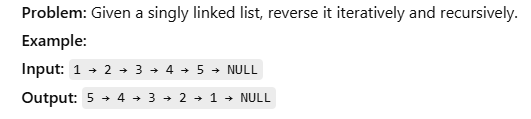
element get at index 3

4

remove at index 1 means element 10

2 3 4

4: Reverse a linked list



**package** com.example.demo;

**public** **class** LinkedListDemo {

**static** **class** Node {

**int** data;

Node next;

}

**static** **class** LinkedList {

Node head;

Node tail;

**int** size;

**public** **void** addLast(**int** val) {

Node temp = **new** Node();

temp.data = val;

temp.next = **null**;

**if** (size == 0) {

head = tail = temp;

} **else** {

tail.next = temp;

tail = temp;

}

size++;

}

**public** **int** size() {

**return** size;

}

**public** **void** display() {

Node temp = head;

**while** (temp != **null**) {

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

temp = temp.next;

}

System.***out***.println();

}

**public** **void** reverse() {

Node prev = **null**;

Node curr = head;

tail = head;

**while** (curr != **null**) {

Node next = curr.next;

curr.next = prev;

prev = curr;

curr = next;

}

head = prev;

}

}

**public** **static** **void** main(String[] args) {

LinkedList l1 = **new** LinkedList();

l1.addLast(1);

l1.addLast(2);

l1.addLast(3);

l1.addLast(4);

l1.addLast(5);

l1.display();

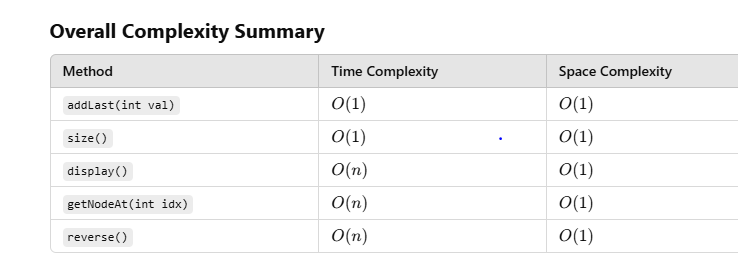
// reverse linkedlist iterative approach

l1.reverse();

l1.display();

}

}



5: - Find middle of linked list :

**Problem:** Use the fast and slow pointer approach to find the middle element.  
**Example:**  
**Input:** 1 → 2 → 3 → 4 → 5 → NULL  
**Output:** 3

**public** **int** mid() {

**if** (head == **null**) {

**throw** **new** IllegalStateException("list is empty");

}

Node slow = head;

Node fast = head;

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

slow = slow.next;

fast = fast.next.next;

}

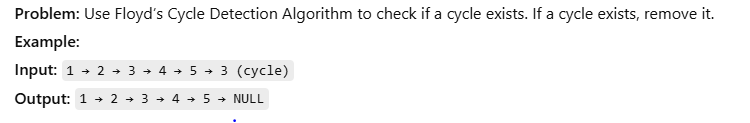
**return** slow.data;

}

**To summarize:**

* The correct condition is fast != null && fast.next != null because it ensures that you can safely access fast.next.next and doesn't risk a NullPointerException when fast reaches the end of the list.

6:- detect and remove cycle from linked list



Detect cycle

**public** **boolean** hasCycle() {

Node slow = head;

Node fast =head;

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

slow=slow.next;

fast =fast.next.next;

**if**(slow==fast) {

**return** **true**;

}

}

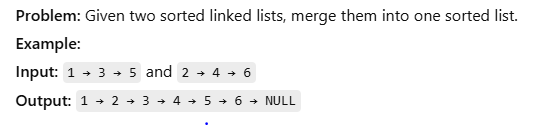
**return** **false**;

}

Remove cycle



7:--- merged two sorted Linked list



**private** **static** LinkedList mergedSortedLinkedList(LinkedList l1, LinkedList l2) {

Node one = l1.head;

Node two = l2.head;

LinkedList res = **new** LinkedList();

**while** (one != **null** && two != **null**) {

**if** (one.data < two.data) {

res.addLast(one.data);

one = one.next;

} **else** {

res.addLast(two.data);

two = two.next;

}

}

**while** (one != **null**) {

res.addLast(one.data);

one = one.next;

}

**while** (two != **null**) {

res.addLast(two.data);

two = two.next;

}

**return** res;

}

**public** **static** **void** main(String[] args) {

LinkedList l1 = **new** LinkedList();

LinkedList l2 = **new** LinkedList();

l1.addLast(1);

l1.addLast(3);

l2.addLast(2);

l2.addLast(4);

LinkedList merged = *mergedSortedLinkedList*(l1, l2);

merged.display();

}

**Time Complexity:**

1. **Traversing the lists**:
   * You iterate through both linked lists exactly once. Since the total number of nodes in both lists combined is n + m (where n is the number of nodes in l1 and m is the number of nodes in l2), you perform O(n + m) operations.
2. **Adding nodes to the result list**:
   * For each node in the original lists, you are adding it to the result list. This takes constant time for each node, so it's also O(n + m).

Therefore, the **time complexity** is O(n + m).

**Space Complexity:**

1. **Result List**:
   * The space used for the result list is proportional to the total number of nodes in the two lists combined (n + m). So, the space complexity for the result list is O(n + m).
2. **Auxiliary Space**:
   * The space used by the pointers (one, two, and res) is constant (O(1)), because you're only using a few extra variables for traversal and result list creation.

Thus, the **space complexity** is O(n + m) due to the result list

8:- Added two linked list in java

**private** **static** LinkedList addTwoLinkedList(LinkedList l1, LinkedList l2) {

Node one = l1.head;

Node two = l2.head;

**int** carry = 0;

LinkedList result = **new** LinkedList();

**while** (one != **null** || two != **null** || carry != 0) {

**int** sum = carry;

**if** (one != **null**) {

sum += one.data;

one = one.next;

}

**if** (two != **null**) {

sum += two.data;

two = two.next;

}

carry = sum / 10;

result.addLast(sum % 10);

}

**return** result;

}

**Time Complexity:**

* **O(n + m)**, where n is the length of l1 and m is the length of l2. You traverse both linked lists once.

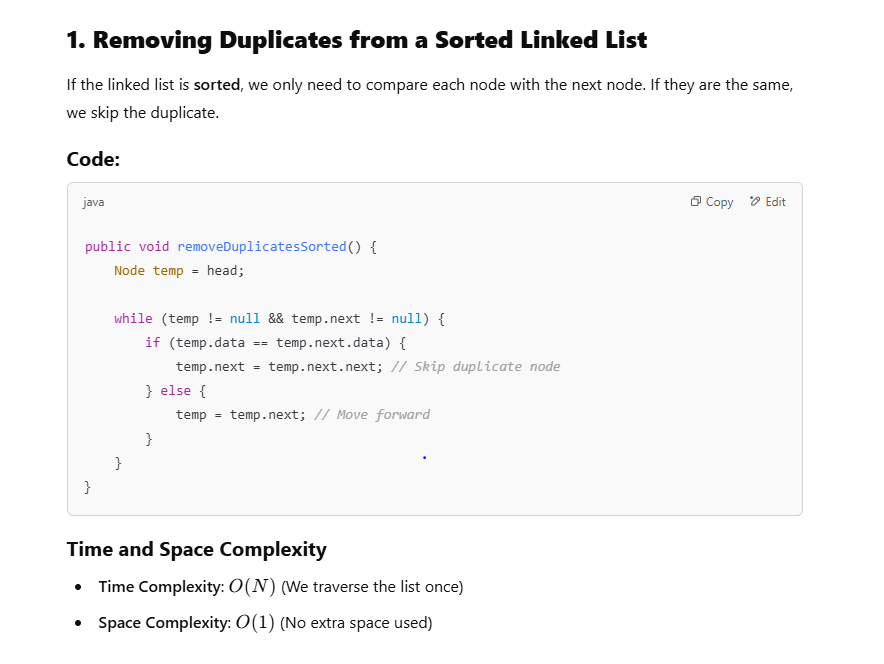
**Space Complexity:**

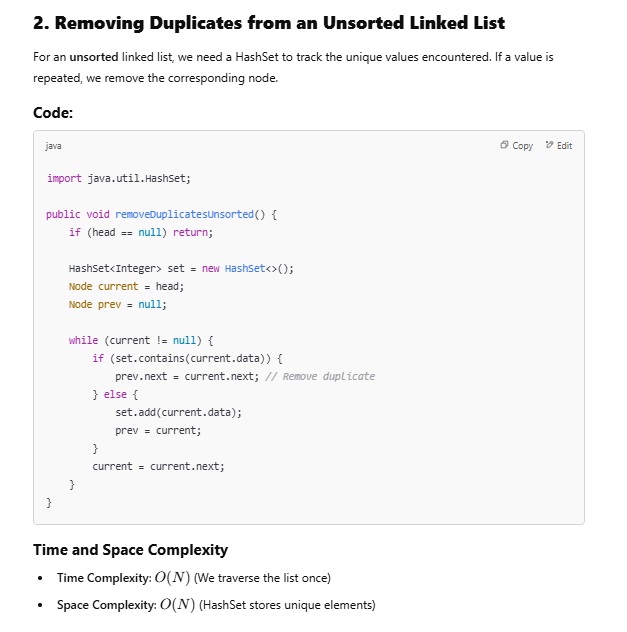
* **O(n + m)**, where n and m are the lengths of the input lists, as you're creating a new list for the result.

9:--- remove duplicate from Linked List

There are two common scenarios for removing duplicates from a linked list:

1. **Sorted Linked List** - We can remove duplicates by simply skipping over duplicate nodes.
2. **Unsorted Linked List** - We need to use a HashSet to track already seen values.





10:- kth from last

**public** **int** kthFromLast(**int** k) {

Node slow = head;

Node fast = head;

**for** (**int** i = 0; i < k; i++) {

fast = fast.next;

}

**while** (fast != tail) {

slow = slow.next;

fast = fast.next;

}

**return** slow.data;

}