

# CS201

## Data Structures and Algorithms

Revision Session 1

arrays vs linked lists

singly linked lists

## **main distinctions**

storage in memory and indexing

## **implementation:**

node and linked list

## **algorithms:**

searching for a value

getting the  $i$ th element

insertion (last, first, middle)

deletion (last, first, middle)

storage in memory

arrays

```
int[] exampleArray = new int[8];
```

```
array[0] = 4;
```

```
array[1] = 44;
```

```
array[2] = 3;
```

...



	1	2	3				

	1	2	3	h	e	l	l
o	,		w	o	r	l	d
\0							

linked lists



```
import java.util.LinkedList
```

```
...
```

```
LinkedList<Integer> list = new LinkedList<>();
```

```
list.add(4);
```

```
list.add(44);
```

```
list.add(3);
```

```
...
```

4

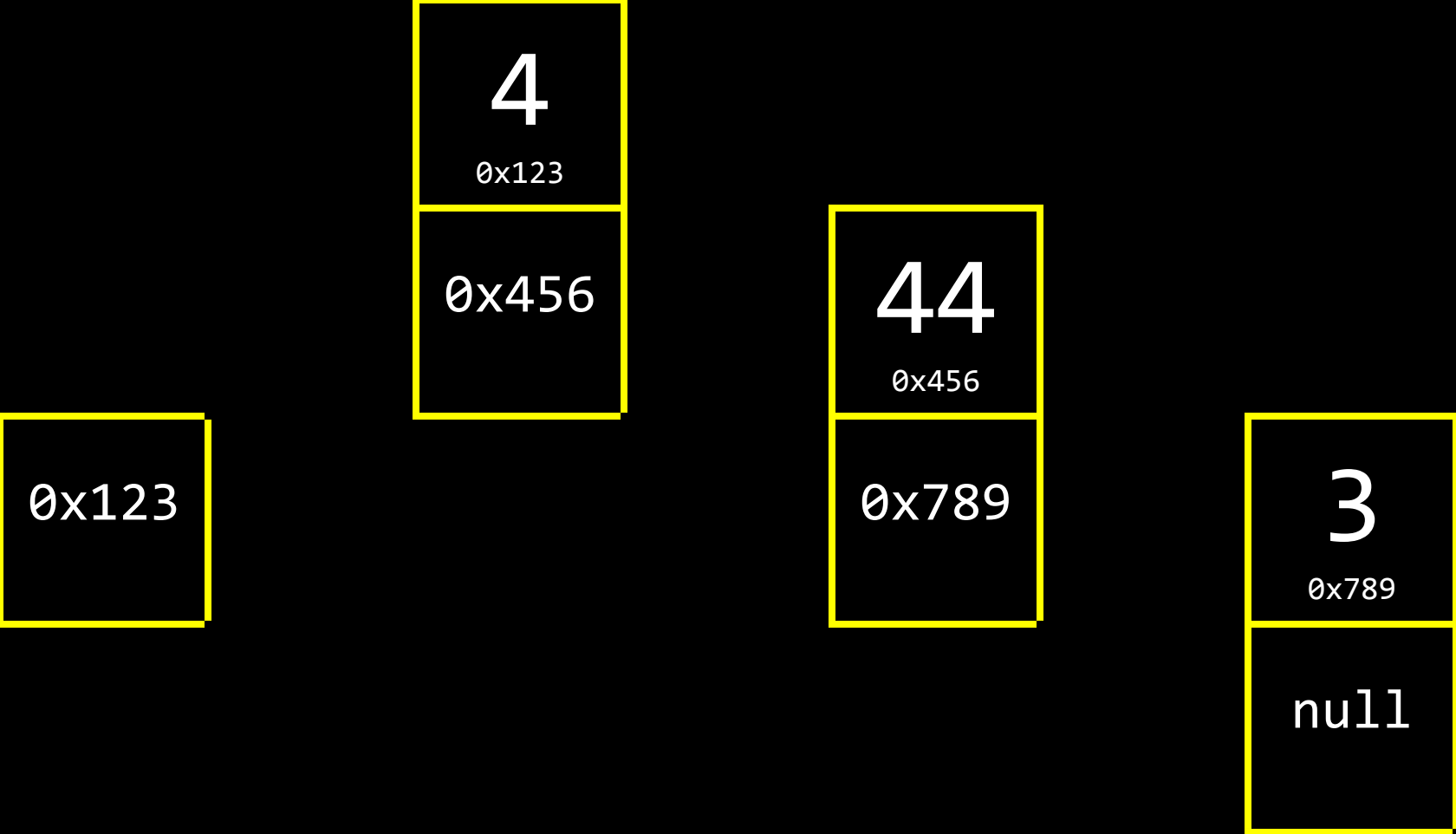
0x123

44

0x456

3

0x789



indexing

arrays

4	44	3	76	61			
0x2714	0x2718	0x2714	0x271C	0x2720			

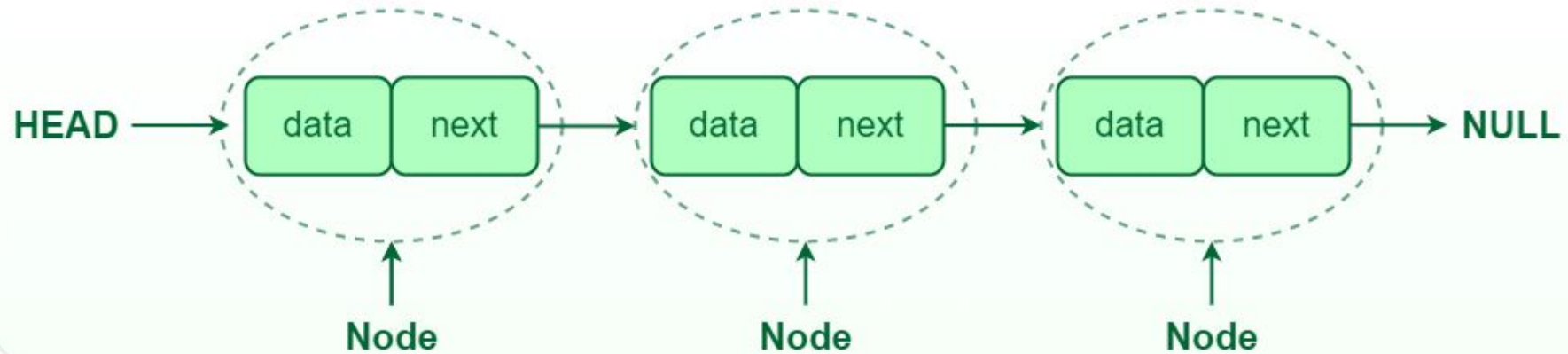
```
exampleArray[4];
```

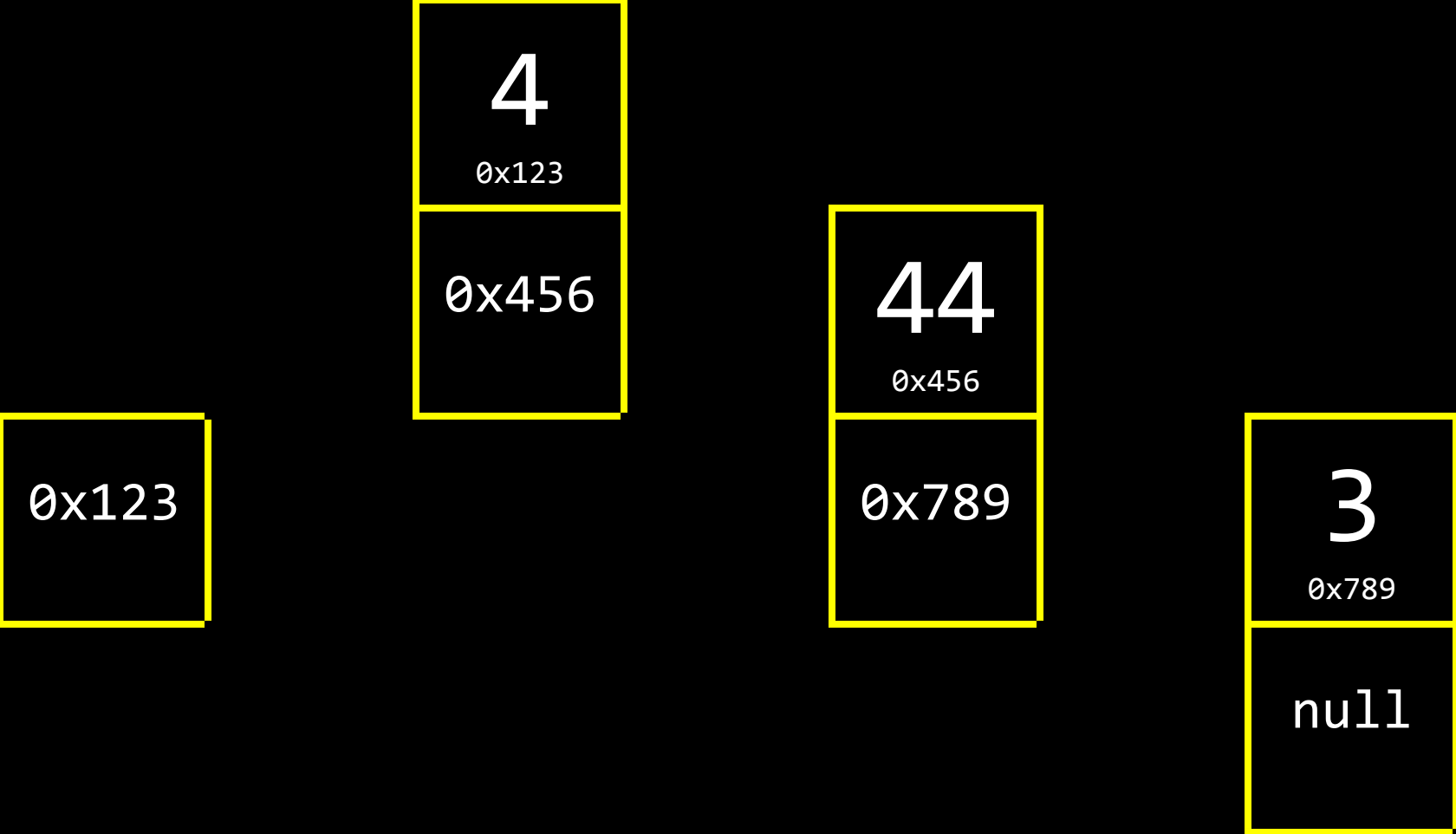
```
output: 61
```

$O(1)$



linked lists





$O(n)$

implementation

# Node

```
public class Node {  
    public int data;  
    public Node next;  
  
    1 usage  neslihancesurr  
    public Node(int data){  
        this.data = data;  
        next = null;  
    }  
}
```

# Linked List

```
public class LinkedList {  
    27 usages  
    public Node head;  
    18 usages  
    public Node tail;  
  
    3 usages  neslihancesurr  
    public LinkedList(){  
        head = null;  
        tail = null;  
    }  
}
```

search algorithms

# searching with a value in arrays

- Linear search
- Binary search
- Jump search
- Interpolation search
- Exponential search
- ...



# searching with a value in linked lists

make temporary Node pointer that points to the head

Loop that stops when the last element points to null:

check the data in the node

if it equals value, return that node

temporary pointer points to the next node

Broken out of the loop without any node

Either return null or give a warning

```
public Node searchNode(int i){  
    Node tmp = head;  
  
    while (tmp != null){  
        if (tmp.data == i){  
            return tmp;  
        }  
        tmp = tmp.next;  
    }  
    return null;  
}
```

searching for the  $i$ th node arrays

`myArray[index]`

# searching for the $i$ th node in linked lists

make temporary Node pointer that points to the head

start an index count from 0

**While the node is not null AND the count is lower than index :**

    increment index count

    temporary pointer points to the next node

Once the counter reaches index we break out and return the node

```
public Node searchIthNode(int i) { // Index number starting from 0
    Node tmp = head;
    int count = 0;

    while (tmp != null && count < i) {
        count++;
        tmp = tmp.next;
    }

    if (tmp == null) {
        throw new IndexOutOfBoundsException("Index" + i + "is out of bounds.");
    }

    return tmp;
}
```

insertion

# insertion in arrays (assuming there is extra space)

check to see if index is within bounds

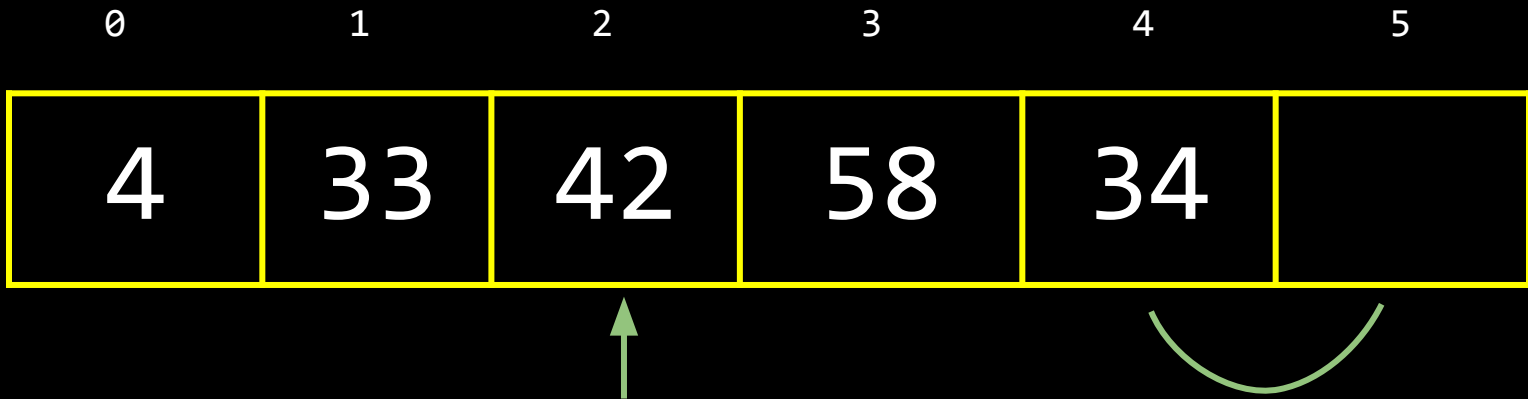
**for loop:**

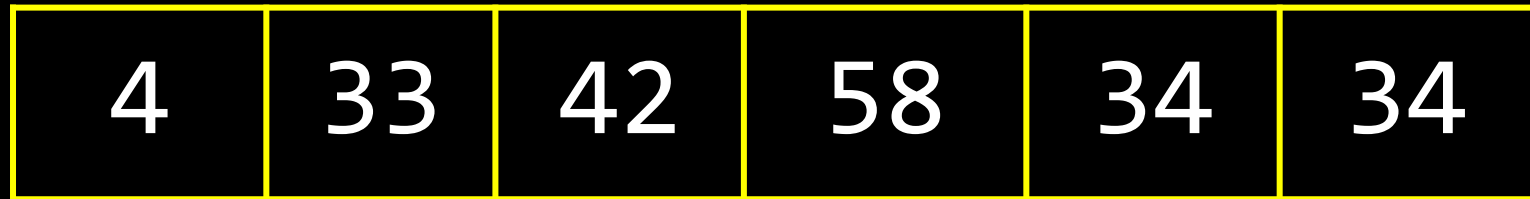
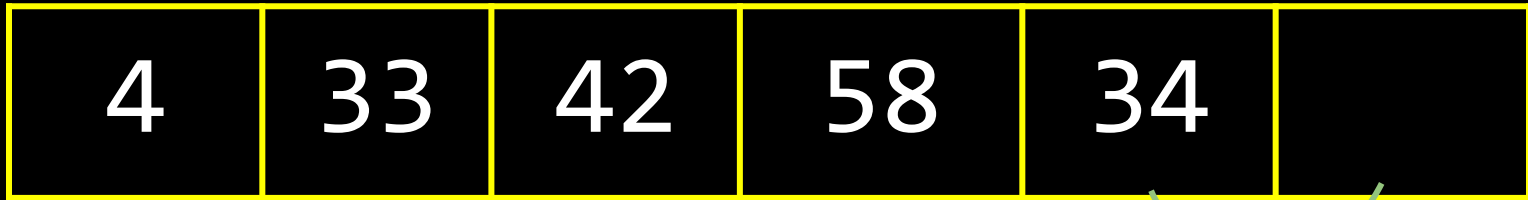
start shifting to the right from the end of the array

move until the the Kth element (included)

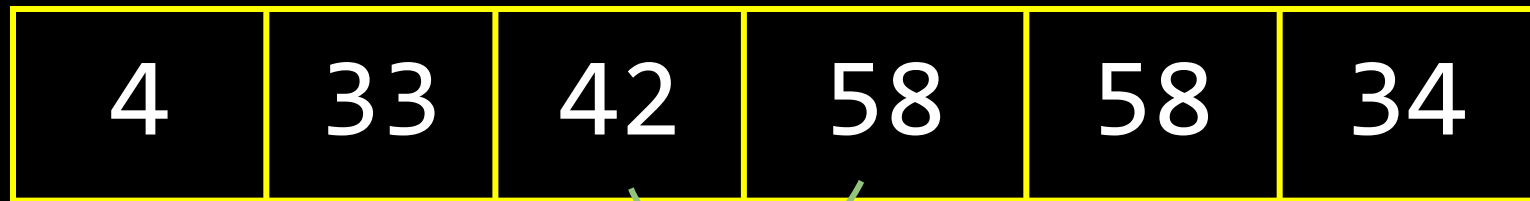
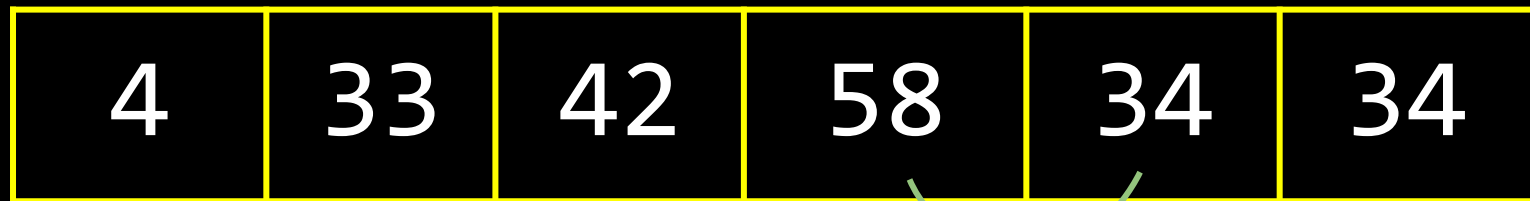
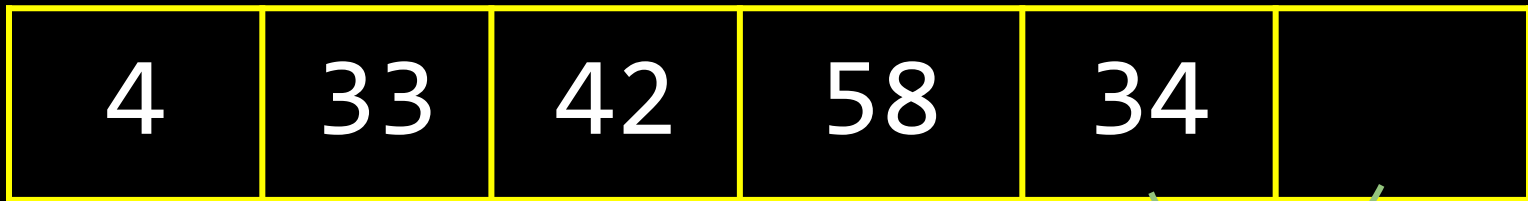
Place the new data in the Kth place

14	42	34	62	-70	3	80	5	6	9					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14









4	33	42	42	58	34
---	----	----	----	----	----

4	33	17	42	58	34
---	----	----	----	----	----



# insertion in linked lists (first)

1. next field of new node

points to the head (FIRST CHANGE) }

2. the head pointer points to the new node

3. if the tail is null (if the list is empty)

the tail pointer also points to the new node

```
public void insertFirst(Node newNode) {  
    if (tail == null) {  
        tail = newNode;  
    }  
    newNode.next = head;  
    head = newNode;  
}
```

## insertion in linked lists (last)

1. in any circumstance point  
the tail pointer to new node
2. if head is null (list is empty)  
make head point to new node
3. if the list is not empty make next field of tail point to new  
node

```
public void insertLast(Node newNode){  
    if (head == null){  
        head = newNode;  
    }  
    else {  
        tail.next = newNode;  
    }  
    tail = newNode;  
}
```

# insertion in linked lists

(middle) the previous node needs to be provided as a parameter

1. next field of new node points to the next field of previous node  
(the order is important! If you run the second line first, you lose the address of the following node)
2. the next field of previous node points to the new node

```
public void insertMiddle(Node newNode, Node previous){  
  
    newNode.next = previous.next;  
    previous.next = newNode;  
}
```

deletion

# deletion of the k'th element in arrays

Check to see if the index is within bounds

**For loop:**

Start shifting to the left from  $k+1$  to  $k$

until you reach the final index

make sure the final index does not repeat

14	42	34	62	-70	3	80	5	6	9					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

4	33	42	58	34	98
---	----	----	----	----	----





4	33	42	58	34	98
---	----	----	----	----	----




4	33	58	58	34	98
---	----	----	----	----	----




4	33	42	58	34	98
---	----	----	----	----	----



4	33	58	58	34	98
---	----	----	----	----	----



4	33	58	34	34	98
---	----	----	----	----	----



4	33	42	58	34	98
---	----	----	----	----	----

4	33	58	34	98	98
---	----	----	----	----	----



4	33	58	34	98	98
---	----	----	----	----	----



4	33	58	34	98	0
---	----	----	----	----	---

# deletion in linked lists (first)

1. if list is empty, return
2. make head pointer point to the second element
3. if head is now null,

it means there are no other elements (there was one and we deleted it)

in that case, make the tail point to null because there are no elements left

```
public void deleteFirst(){  
    //different from the code in the book  
    if (head == null){  
        System.out.println("The list is already empty.");  
        return;  
    }  
    head = head.next;  
  
    if (head == null){  
        tail = null;  
    }  
}
```

# deletion in linked lists (last)

1. create pointer tmp for traversal and previous for node before last
2. after loop, tmp is the last, previous is before last
3. if previous is null (only one element), set head and tail to null
4. next field of before last (previous) is null and tail is before last (previous)

```
1 usage  neslihanesurr *
public void deleteLast(){
    Node tmp = head;
    Node previous = null;

    if (head == null){
        System.out.println("The list is already empty!");
    }

    while (tmp != tail && tmp != null){
        previous = tmp;
        tmp = tmp.next;
    }

    if (previous == null){
        head = null;
        tail = null;
    } else {
        previous.next = null;
        tail = previous;
    }
}
```

# deletion in linked lists (middle)

1. create tmp for traversal, previous for before deleted node
2. after loop, tmp is deleted and previous is before deleted
3. if there is more than 2 elements, set next of prev to the one after deleted. set next of deleted to null (optional)

2 usages neslihancesurr

```
public void deleteMiddle(Node deleted){  
    Node tmp = head;  
    Node previous = null;  
  
    while (tmp != deleted){  
        previous = tmp;  
        tmp = tmp.next;  
    }  
  
    if (previous != null) {  
        previous.next = deleted.next;  
        deleted.next = null;  
    }  
}
```

# number of nodes

1. create a **counter** variable and a **pointer** to traverse the list
2. **traverse** the entire list until the tail and increment the count
3. **return** count

```
no usages  neslihancesurr  
public int nodeCount(){  
    int count = 0;  
    Node tmp = head;  
  
    while (tmp != null){  
        tmp = tmp.next;  
        count++;  
    }  
    return count;  
}
```



# merging two linked lists

1. if one list is empty, return the other list
2. create new list
3. head of L1 attaches to head of newList
4. tail of L2 attaches to the tail of newList
5. next field of tail (L1) points to the head of L2
6. return newList

```
no usages  neslihancesurr
public LinkedList merge(LinkedList L1, LinkedList L2){
    if (L1.head == null){
        return L2;
    }
    if (L2.head == null){
        return L1;
    }

    LinkedList newList = new LinkedList();

    newList.head = L1.head;
    newList.tail = L2.tail;
    L1.tail.next = L2.head;

    return newList;
}
```