Intro and How it works

Wednesday, 7 February 2024 11:57 AM

what is linked list? why do we need linked list?

→ In java, if we want to store data in a continous memory location, we go for arrays.

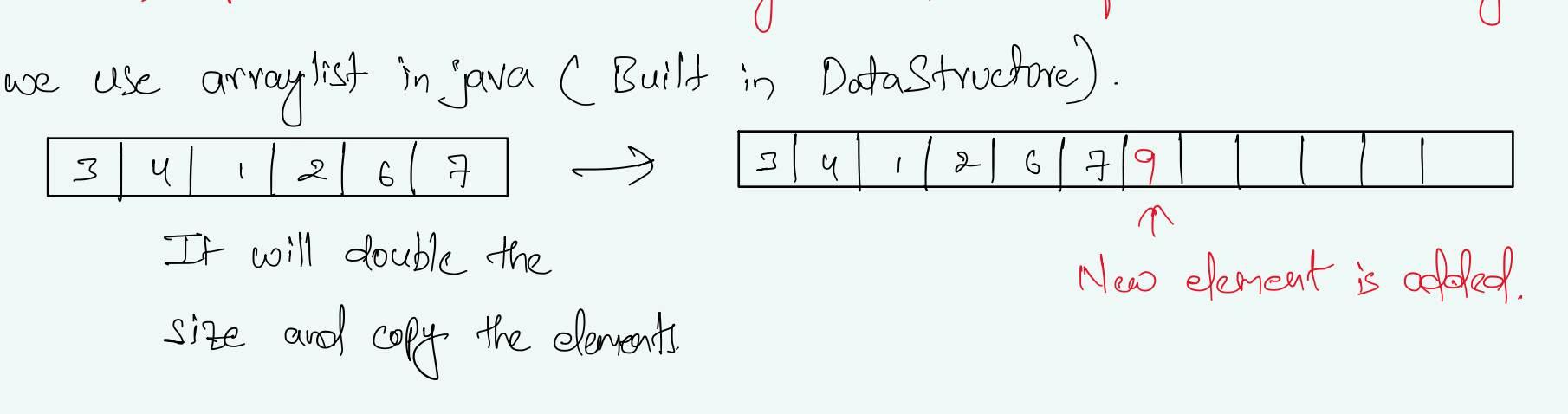
Ex: we want to store 6 elements

3	4	1	2	6	7

→ Now, if we want to add new element '9'.

→ If we add now, we will get index out of bound or basically array is full.

→ If we use arraylist in java (Built in DataStructure).



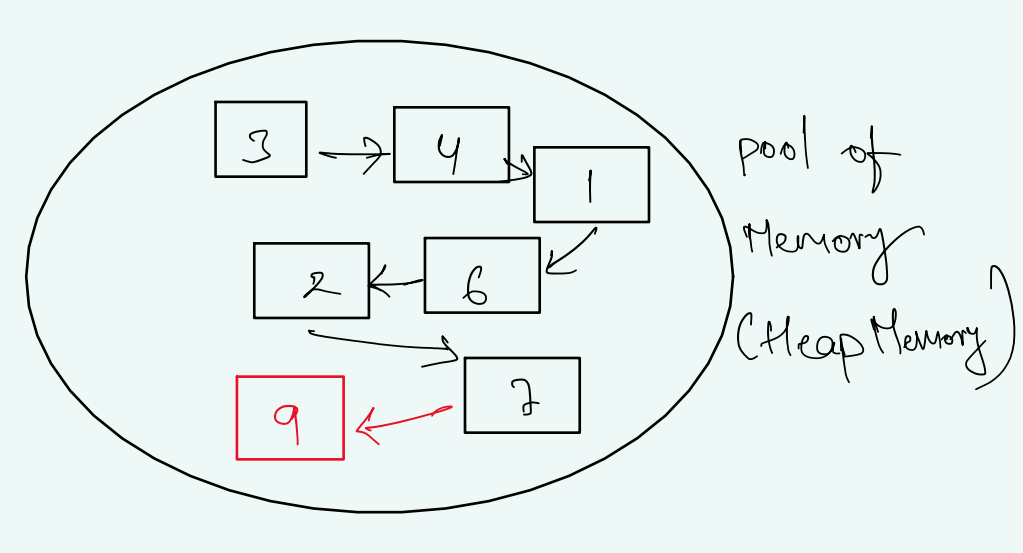
Time Complexity: O(1) avg but there are some limitations.

→ Here, it is doing some doubling the size and copying the element.

→ Instead of storing in continous memory location, we can store in different locations.

Linked List: A linked list is a linear data structure, in which the elements are not stored in continous memory location. The elements are in a linked list are linked using pointers.

→ Now we want to store new element in different location which is attached to last element.



→ linked list will overcome limitations of array.

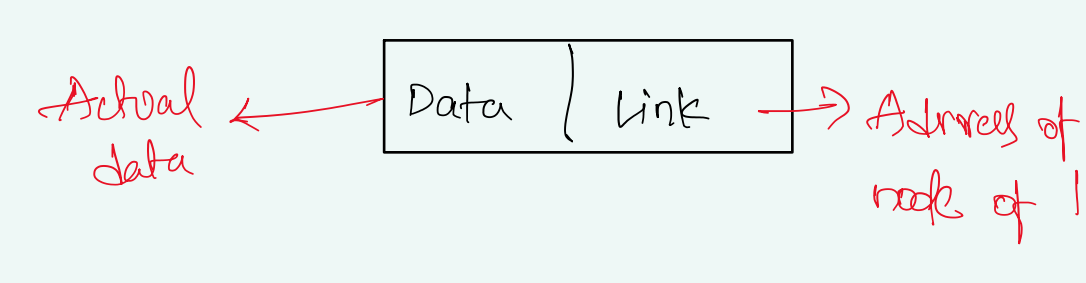
Types of linked list:

1) Single linked list → Navigation is forward only.
2) Doubly linked list → Forward and backward navigation possible.
3) Circular linked list → Last element linked to the first element (Circular loop of elements).

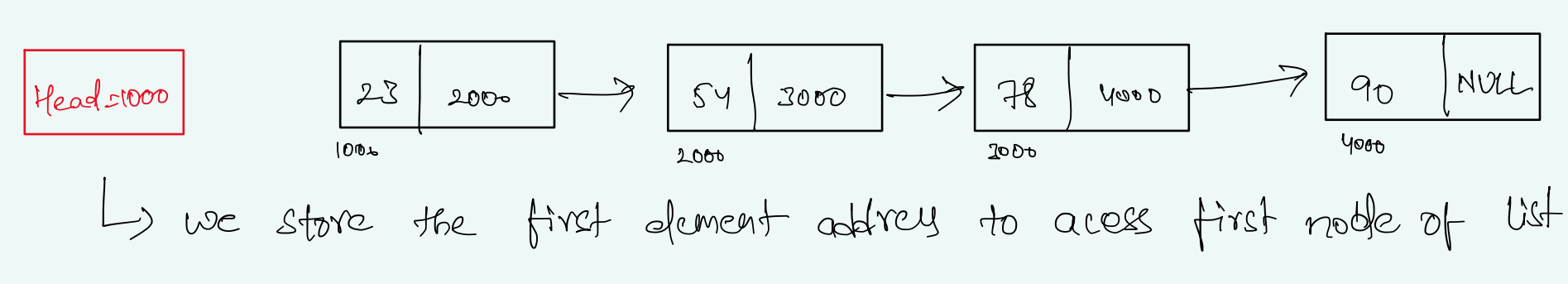
what is single linked list?

→ A single linked list is a list made up of nodes that contain two parts.

1) Data
2) Link to Next Node



Representation of Single LL: we want to store list of elements 23, 54, 78, 90.



↳ we store the first element address to acess first node of list. (Head pointer).

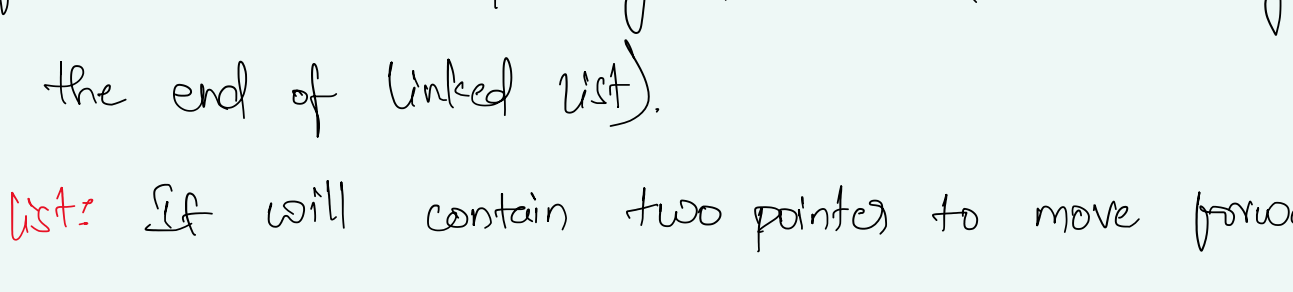
→ we can traverse through out the list using next address that are stored in link part of node.

Head → Reference variable pointing to first element.

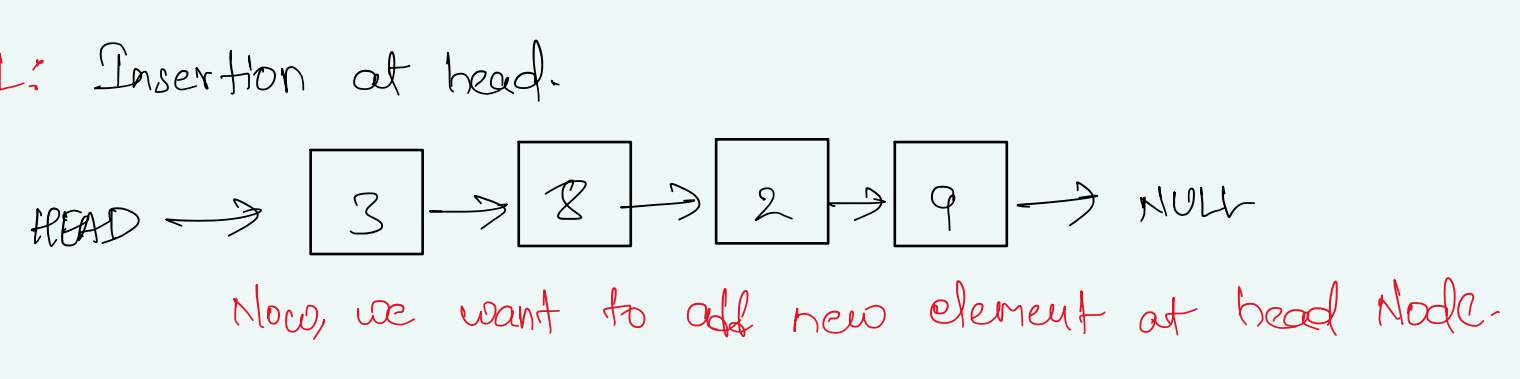
→ Here, we can't acess the random element in constant time.

Tail → Reference variable pointing to last element. (easy acess to add elements at the end of linked list).

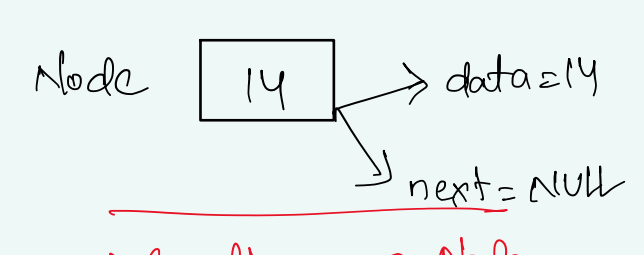
Doubly linked list: It will contain two pointer to move forward aswell as backwards



Insertion in SLL: Insertion at head.



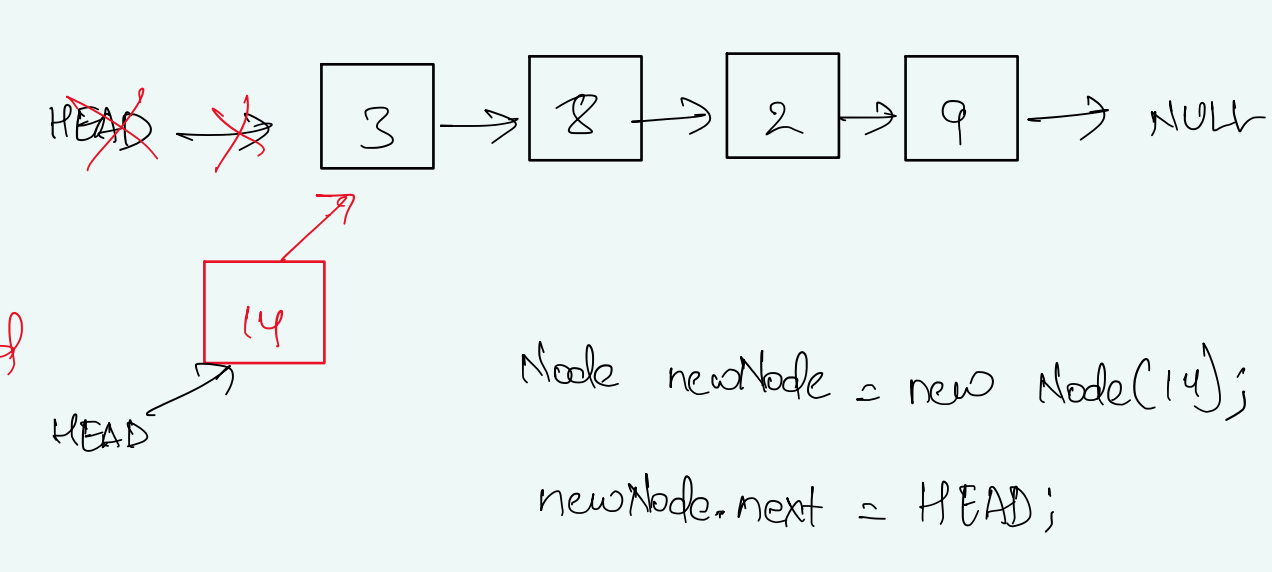
Now, we want to add new element at head Node.



→ Creating new Node

→ Add link b/w newnode and head pointer.

→ change the head pointer to newnode.



Node newNode = new Node(14);

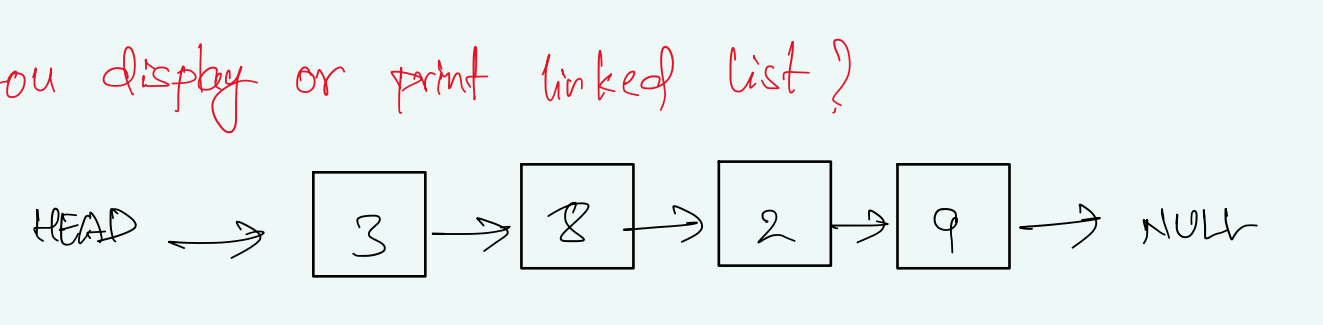
newNode.next = HEAD;

HEAD = newNode; size++; this will solve.

→ what if no element are in list: use a check if you are having tail pointer.

if (tail == null)
tail = head; → Constant time. O(1)

→ How will you display or print linked list?



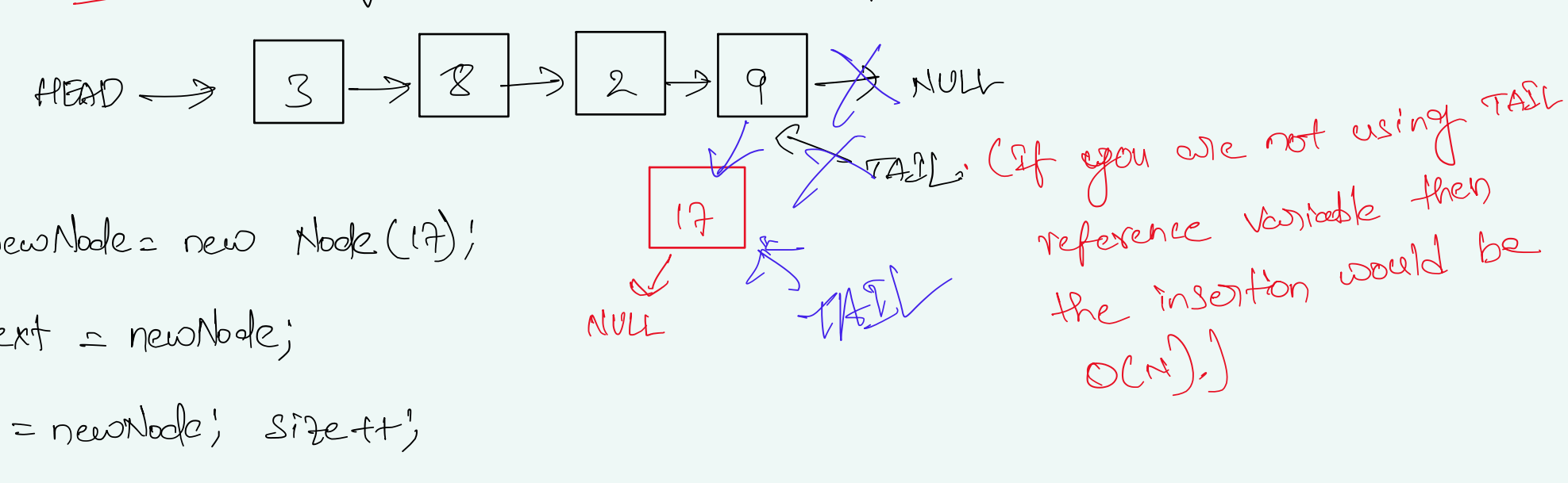
→ store the head pointer in temporary Node Variable and trave using next of each node. if node next is NULL we are at end of list.

```
Node node = HEAD;
while (node != NULL) {
    print (node.data);
    node = node.next;   → This will be null if we are at last node.
}
```

output: 3 8 2 9

Note: if we are using head pointer to travere, we can do the traversal only once. Instead we are storing the head into temprorary reference variable.

Insertion at End: Inserting new value at end of linked list. (SLL).



(If you are not using TAIL reference variable then the insertion would be O(N).)

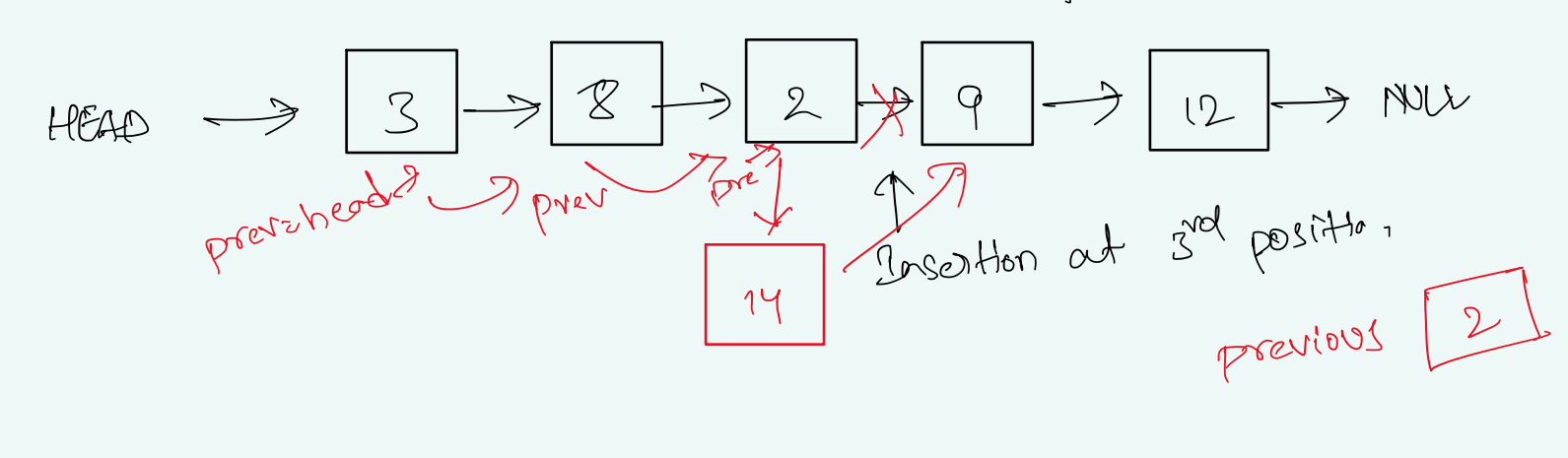
```
Node newNode = new Node(17);
TAIL.next = newNode;
TAIL = newNode; size++;
```

→ To handle empty list:

```
if (tail == NULL)
    insertFirst(data);
```

→ Inserting at head Node using previous function.

Insertion at particular position: you are given with position you need to add new node at that particular index.




```
if (pos == 0)
    call insertionFirst(data);   → Insertion at Head.
if (pos == size)
    call insertionLast(data);    → Insertion at Tail.
```

Algo

→ Traverse till previous node of position.

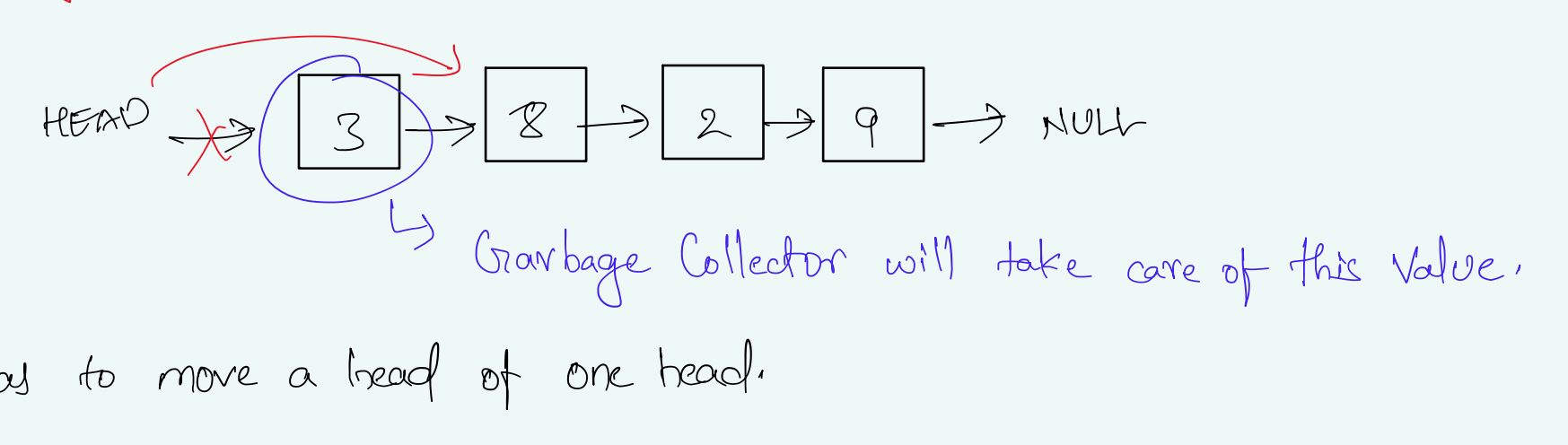
→ Store the next node of previous node before changing link.

→ Assign previous.next to newNode and newNode.next to stored node of previous next node (done in Second step).

Code:

```
Node previous = head;
for (i=0 to (pos-1)):
        previous = previous.next;        } → To find previous node.
Node previousNext = previous.next;  → Storing next node
previous.next = newNode;  → assign previous next to newNode.
newNode.next = previousNext;  → assign newNode next to previous Next.
Size++;
```

Deletion at Begining: Deletion of node (head node) and move the pointer to next node



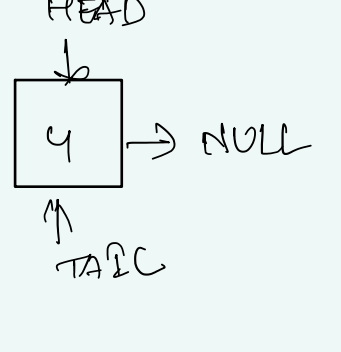
→ HEAD has to move a head of one head.

→ To return the value, store the value of HEAD.

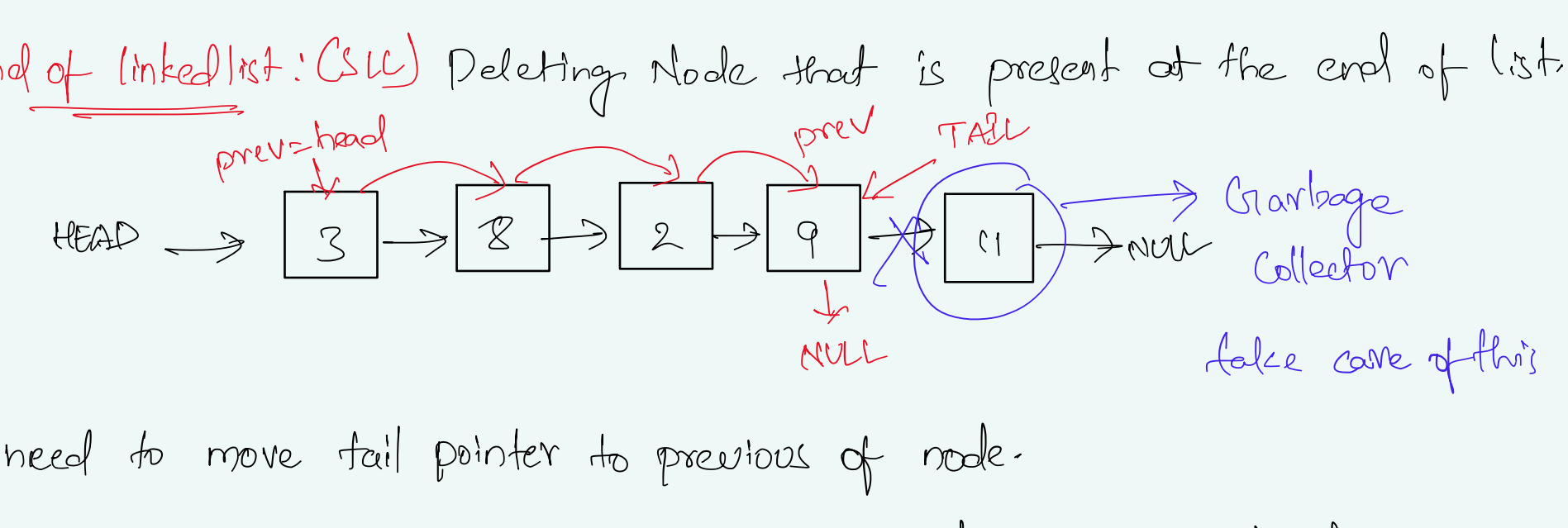
```
int data = HEAD.data;
HEAD = HEAD.next; size--;   Time Complexity O(1).
return data;
```

→ what if we have Single Node?

Now, we have to make tail node also null. if it is still pointing to head node.



Deletion at End of linkedlist: (SLL) Deleting Node that is present at the end of list.



Algo

→ we need to move tail pointer to previous of node.

→ we can't directly access previous node at tail. we need to traverse through linked list.

→ Assign previous to TAIL and TAIL of next to NULL.

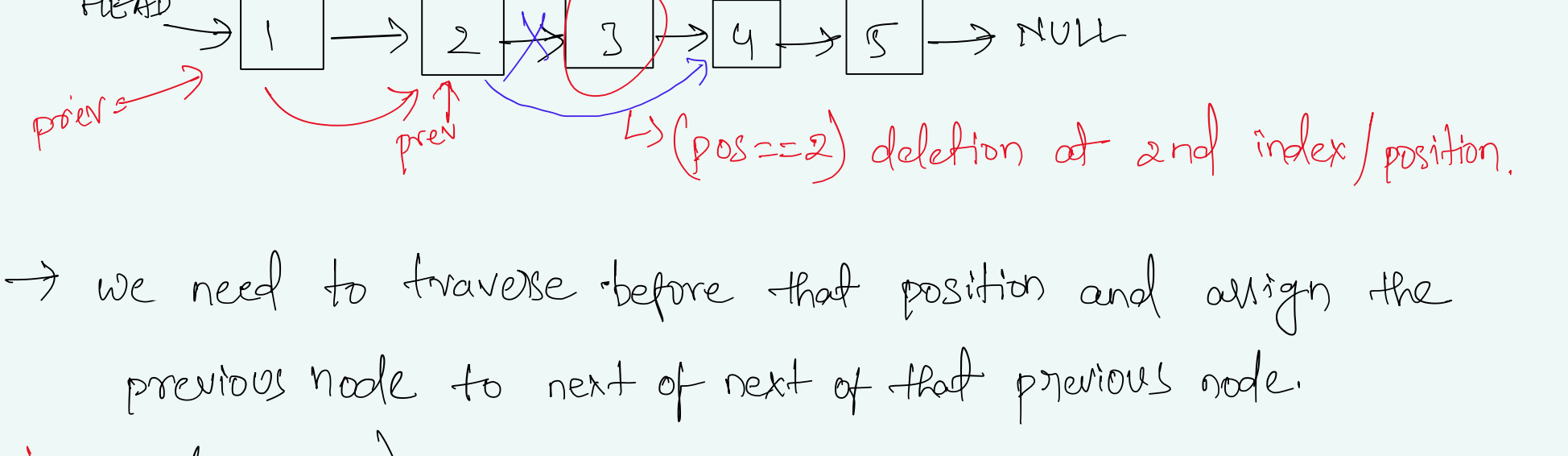
Code:

```
Node prev = HEAD;
for (i=1 to size-1)
        prev = prev.next;
int data = TAIL.data;
TAIL = prev;
TAIL.next = NULL;
size--;
```

Time Complexity is O(N) it is because of we are traversing one node before of tail pointer. → (N-1) = O(N)

Deletion of node at given position: you are given with a position you need to delet that particular node.

→ if pos == 0 then deletion at Head (or) if pos == (size-1) then call function deletion at Tail.



→ we need to traverse before that position and assign the previous node to next of next of that previous node.

Code:

```
if (pos == 0)
    DeletionAtHead();
if (pos == Size-1)
    DeletionAtTail();
Node prev = HEAD;
for (i=0 to pos-1)
        prev = prev.next;       } → Traversing till previous node of given position.
int data = prev.next.data;  → Storing data to return.
prev.next = prev.next.next;  → changing link prev to position node of next.
size--;
return data;
```

Time Complexity is O(N) → because of traversing till previous Node.

→ Best Case : O(1) deletion at HEAD Node. ✓

→ for deletion at TAIL it will take O(N).

(*) To find a Node you are given with data and head Node.

→ Simply traversing through out linked list will give you result.

```
Node find (Node HEAD, int data) {
    Node temp = HEAD;
    while (temp != NULL) {              → Traversing till End.
        if (temp.data == data)          → checking if node data equals to finding data.
            return temp;
        else
            temp = temp.next;           → if not found move to next node.
    }
    return NULL;    → finally returning null if didn't found node with given data.
}
```