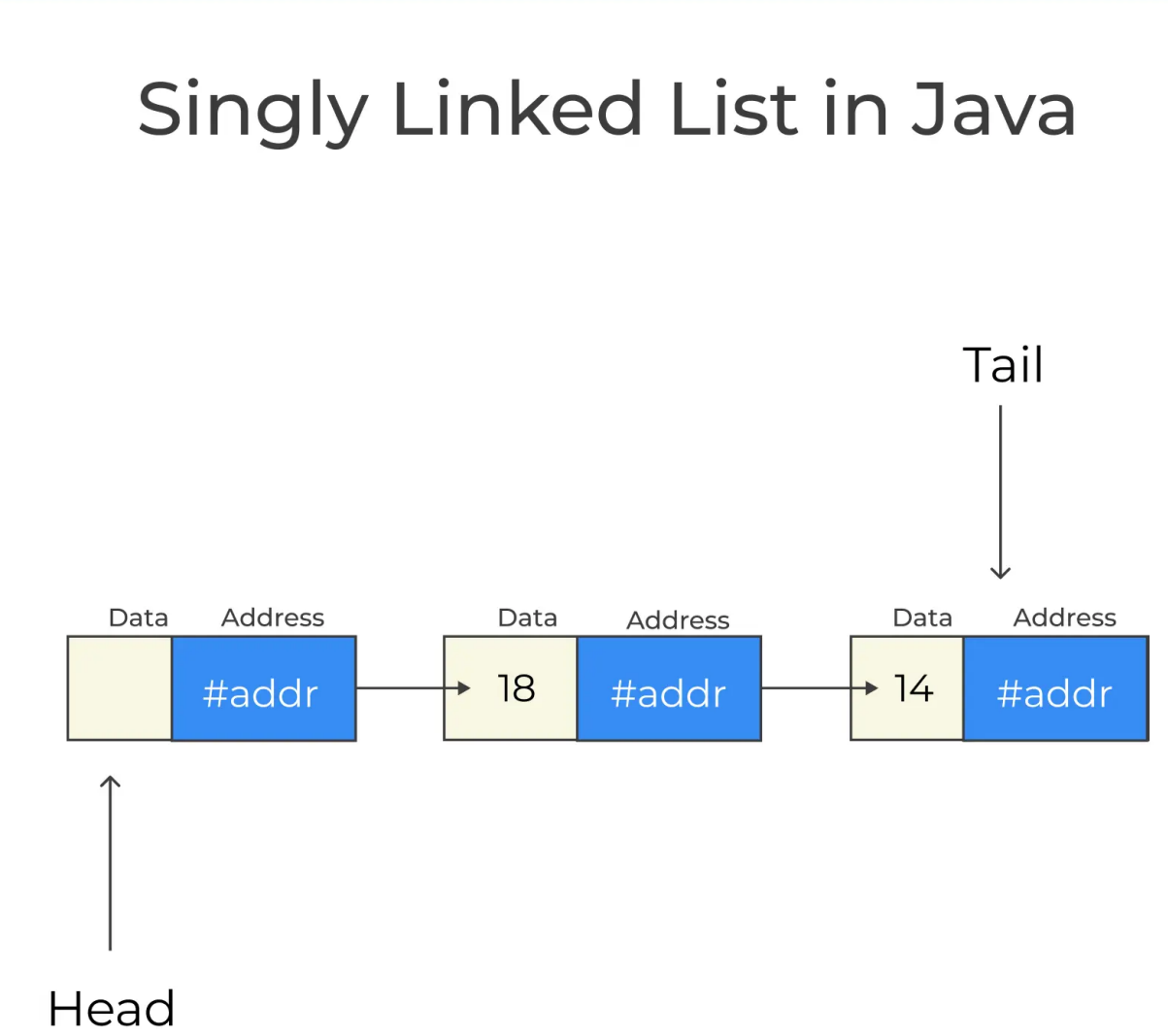
Singly Linked List

Unlike Arrays with are contiguously stored, Linked List is not contiguous and are scattered all over the memory but connected with one another using the next references.

Also, array size can not be dynamically increased however, in Linked List the size can be increased and decreased at any time dynamically.

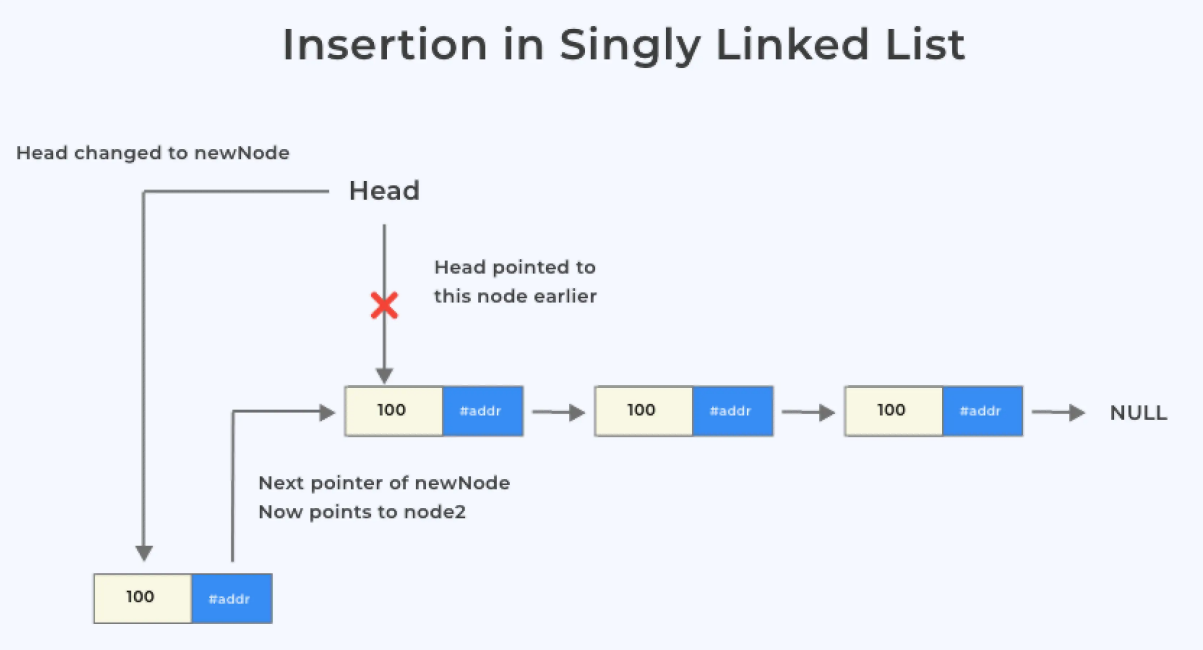


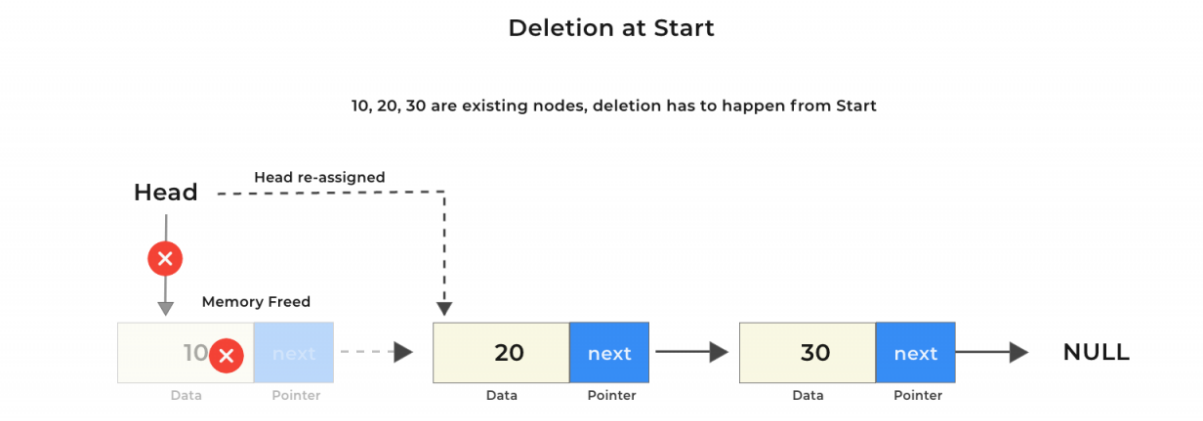

```
class LinkedList {
  Node head; // head

  // Linked list Node
```

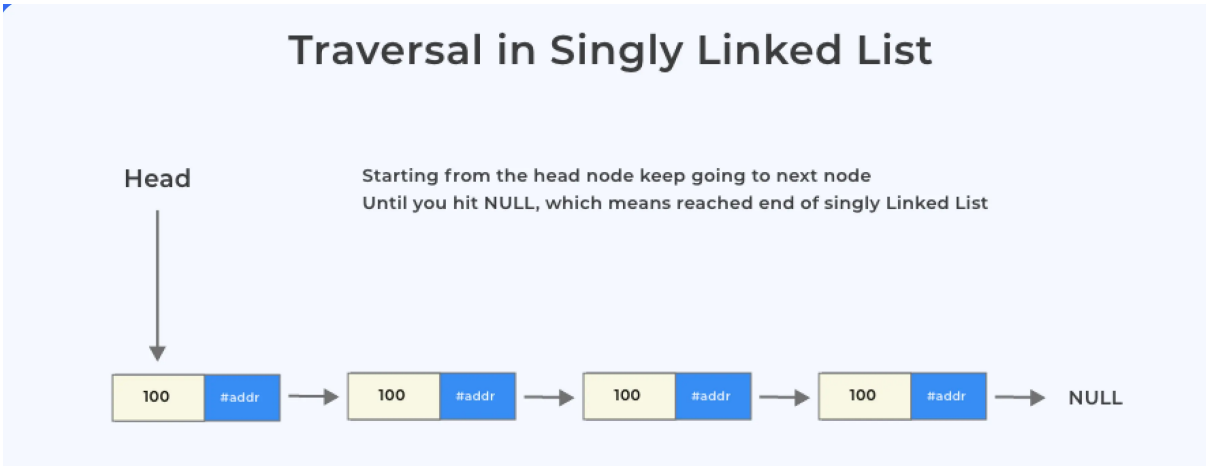
```
  class Node {
    int data;
    Node next;

    // constructor to initialize
    Node(int d) {
      data = d;
      next = null;
    }
  }
}
```

Operation on singly linked list

- Deletion
- Insertion
- Traversals







Insertion in Linked List

How do you add an element to a Singly Linked List ?

Adding data in a Singly Linked List is practically a bit easier than the other data structures. As in Singly Linked List, we does not need to traverse the whole data structure for entering a node, we just need to update the address of the last node which is currently pointing to null, or some other node, with the new node that we want to insert. There are three different ways in which we can insert a node in a linked list -:

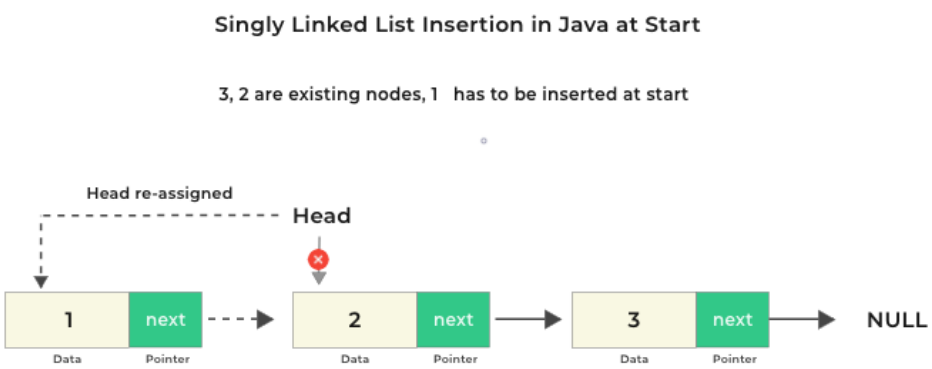
1. **Insertion at beginning**
2. **Insertion at end**
3. **Insertion at nth position**

Inserting an element in the beginning of the Singly Linked List

- To insert a node in the beggining of a linked list, we first have to check the Head's Reference to the first node of a linked list.
- If the head is equal to null, then the list is already empty else the list already has an element whose reference is stored by the head.
- To insert an element, in the beginning, we will have to replace the address stored by the head with the address of the new element we wish to insert.
- The address space of the previously stored element will now be stored in the pointer reference of the inserted element.

Algorithm

- IF HEAD == NULLEXIT
- ELSE
- NEW_NODE = ADDNODE()
- NEW_NODE -> DATA = ITEM
- PTR = HEAD
- NEW -> NEXT = PTR
- HEAD = NEW_NODE
- EXIT

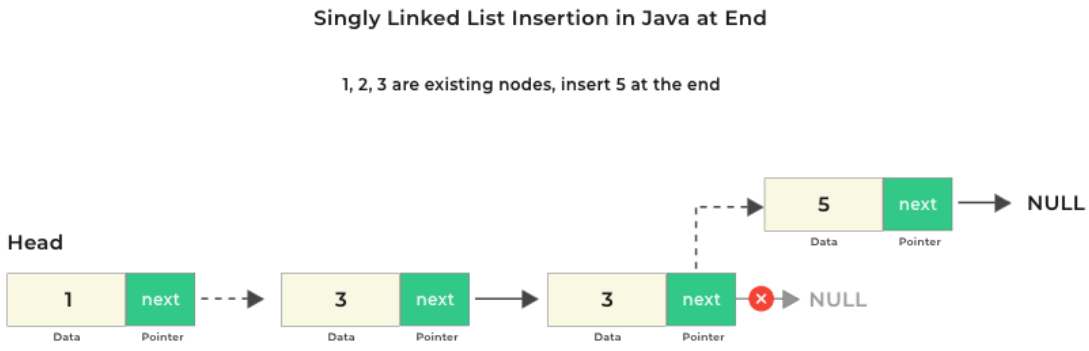


Inserting an element in the End of the Singly Linked List

- First, Traverse the list from head to last position.
- After traversing the list add the new node and allocate memory space to it.
- Point the next pointer of the new node to the null.
- Point the next pointer of current node to the new node.

Algorithm

- IF HEAD == NULLEXIT
- ELSE
- NEW_NODE = ADDNODE()
- NEW_NODE -> DATA = ITEM
- PTR = NULL
- NEW -> NEXT = NULL
- PREV.PTR = NEW_NODE
- EXIT



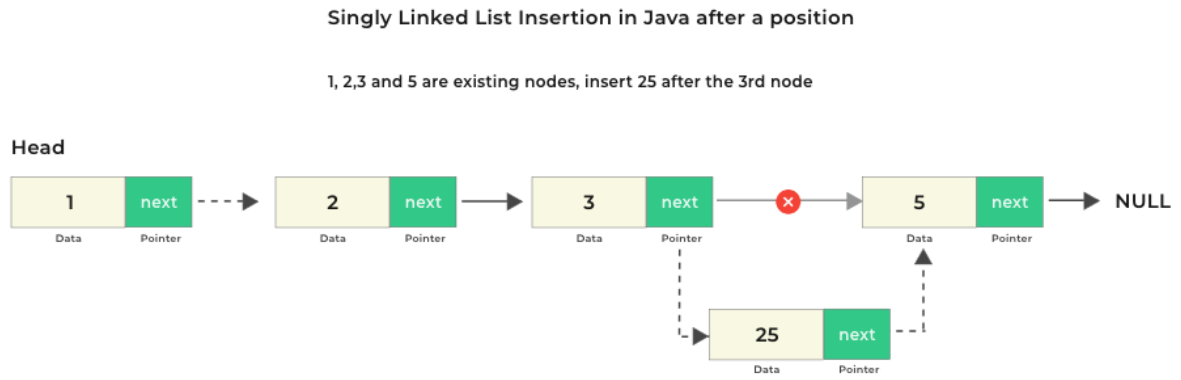
Inserting an element in the nth position of the Singly Linked List

- First, Traverse the list from head to n-1 position
- After traversing the list add the new node and allocate memory space to it.

- Point the next pointer of the new node to the next of current node.
- Point the next pointer of current node to the new node.

Algorithm

- IF HEAD == NULL
  EXIT
- ELSE
- NEW_NODE = ADDNODE()
- NEW_NODE -> DATA = ITEM
- SET TEMP = HEAD
- SET I = 0
- REPEAT
- TEMP = TEMP → NEXT
- IF TEMP = NULL
- EXIT
- PTR → NEXT = TEMP → NEXT
- TEMP → NEXT = PTR
- SET PTR = NEW_NODE
- EXIT




```
package linklist;

public class LinkList_Insert {
```

```
  Node head;

  // Node Class
  class Node {
    int data;
    Node next;

    Node(int x) {
      data = x;
      next = null;
    }
  }

  public Node insertBeginning(int data) {
    Node newNode = new Node(data);
    newNode.next = head;
    head = newNode;
    System.out.println(data + "  inserted at the Begninning");
    return head;
  }

  public void insertEnd(int data) {
    Node newNode = new Node(data);

    if (head == null) {
      head = newNode;
      System.out.println(newNode.data + " inserted at the End");
      return;
    }

    Node temp = head;

    while (temp.next != null)
      temp = temp.next;

    temp.next = newNode;
    System.out.println(newNode.data + " inserted at the End");

  }

  public void insertAfter(int n, int data) {
    int size = calcSize(head);
    int pos=n;

    // Can only insert after 1st position
    // Can't insert if position to insert is greater than size of Linked List
    if (n < 1 || n > size) {
      System.out.println("Can't insert\n");


    } else {
      Node newNode = new Node(data);
      // required to traverse
      Node temp = head;

      // traverse to the nth node
      while (--n > 0)
        temp = temp.next;
```

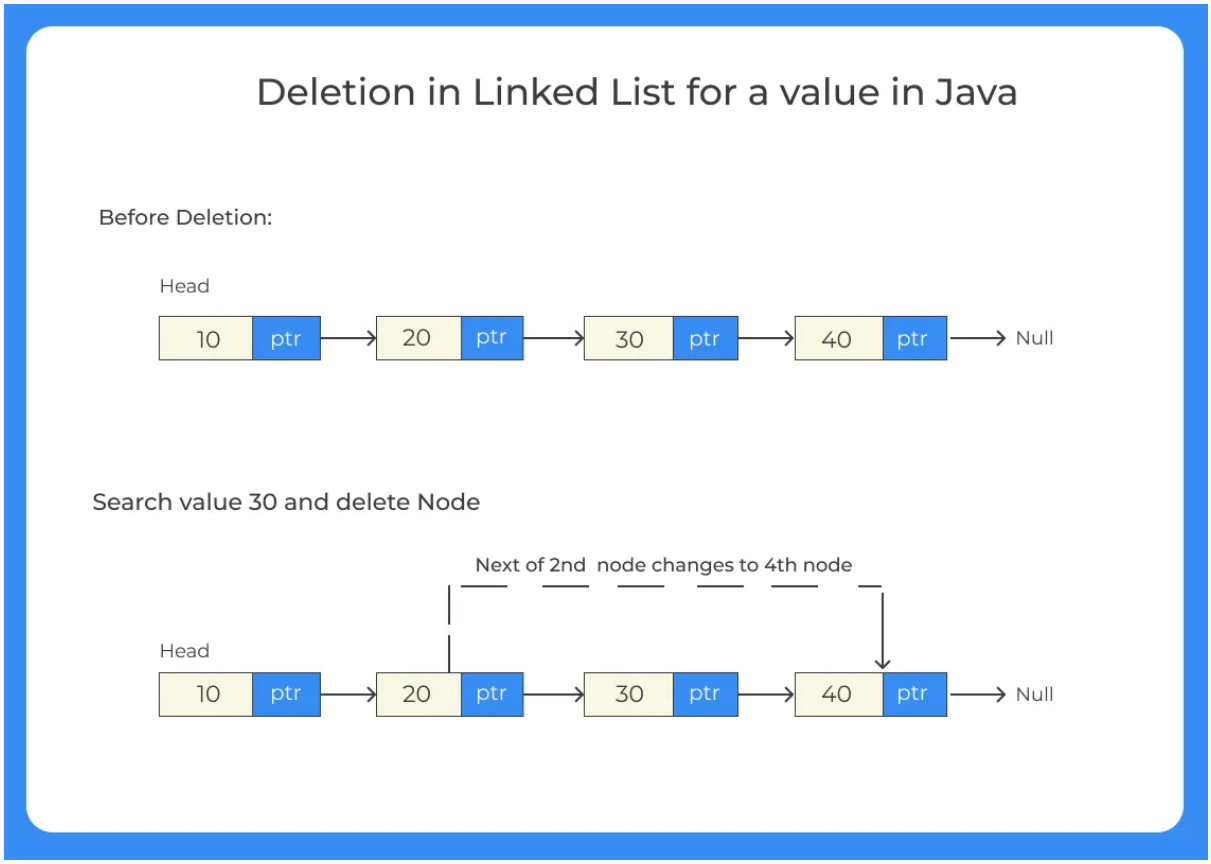
```
      newNode.next = temp.next;
      temp.next = newNode;
      System.out.println(data + " inserted at the "+pos+" position ");
    }
  }

  public void display() {
    System.out.println();
    Node node = head;
    // as linked list will end when Node reaches Null
    while (node != null) {
      System.out.print(node.data + " ");
      node = node.next;
    }
    System.out.println();
  }

  public int calcSize(Node node) {
    int size = 0;
    while (node != null) {
      node = node.next;
      size++;
    }
    return size;
  }

  public static void main(String args[]) {
    LinkList_Insert listObj = new LinkList_Insert();

    listObj.insertBeginning(15);
    listObj.insertBeginning(10);
    listObj.insertBeginning(5);

    listObj.display();

    listObj.insertEnd(20);
    listObj.insertEnd(25);
    listObj.insertEnd(30);
    listObj.insertEnd(35);

    listObj.display();

    listObj.insertAfter(3, 100);

    listObj.display();
  }
}
```

Singly Linked List Deletion in Java

In the singly linked list, we can delete the node in the following ways or we can say they ways of deleting nodes .

When we delete the node in the linked list then there are three ways to delete the node .

- **Deletion for position**
  - At Start
  - At End
  - For a given position in middle
- **Deletion for a Value**




```
package linklist;

public class LinkList_Delete_Position {
  Node head;

  // Node Class
  class Node {
    int data;
    Node next;

    Node(int x) // parameterized constructor
    {
      data = x;
```

```
      next = null;
    }
  }

  public Node insert(int data) {
    // Creating newNode memory & assigning data value
    Node newNode = new Node(data);
    // assigning this newNode's next as current head node
    newNode.next = head;

    // re-assigning head to this newNode
    head = newNode;

    return head;
  }

  public void display() {
    Node node = head;
    // as linked list will end when Node reaches Null
    while (node != null) {
      System.out.print(node.data + " ");
      node = node.next;
    }
    System.out.println();
  }

  void deletepos(int pos) {
    Node temp = head;
    Node previous = null;

    int size = calcSize(head);

    if (pos < 1 || pos > size) {
      System.out.println("Enter valid position");

      return;
    }

    // if the head node itself needs to be deleted
    if (pos == 1) {
      // changing head to next in the list
      head = temp.next;
      System.out.println("Deleted Item: " + temp.data);
      return;
    }

    // run until we find the value to be deleted in the list
    while (--pos > 0) {
      // store previous link node as we need to change its next val
      previous = temp;
      temp = temp.next;
    }

    // (pos-1)th node's next assigned to (pos+1)nth node
    previous.next = temp.next;
    System.out.println("Deleted Item: " + temp.data);

  }
```

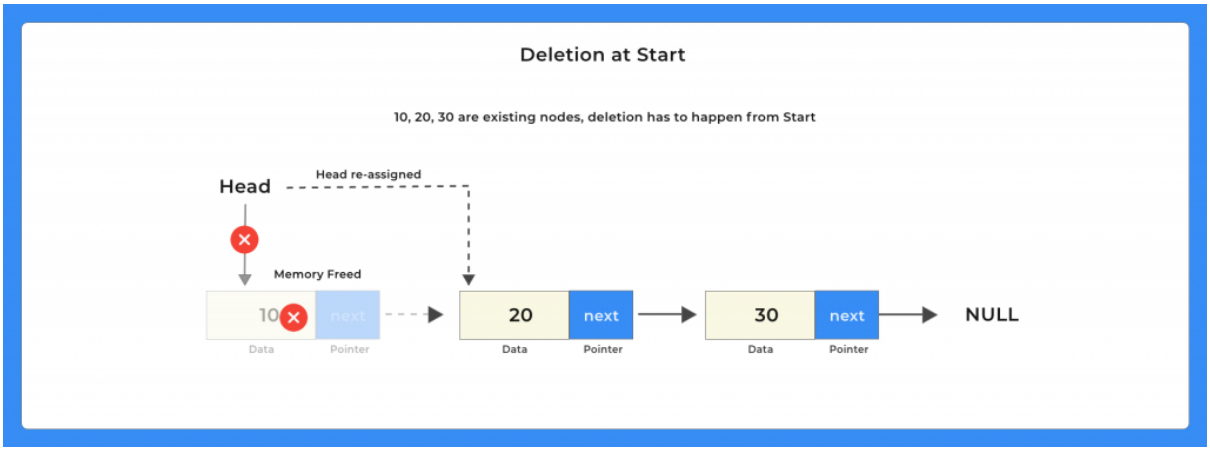
```
  public int calcSize(Node node) {
    int size = 0;
    // traverse to the last node each time incrementing the size
    while (node != null) {
      node = node.next;
      size++;
    }
    return size;
  }

  public static void main(String args[]) {
    LinkList_Delete_Position ll = new LinkList_Delete_Position();

    ll.insert(60);
    ll.insert(50);
    ll.insert(40);
    ll.insert(30);
    ll.insert(20);
    ll.insert(10);

    ll.display();

    ll.deletepos(1);
    ll.display();

    ll.deletepos(3);
    ll.display();

    ll.deletepos(4);
    ll.display();

    ll.deletepos(-2); // not valid as pos negative
    ll.deletepos(10); // not valid as breaches size of Linked List

  }
}
```

```
package linklist;

public class LinkList_Delete_Value {
  Node head;

  // Node Class
  class Node {
    int data;
    Node next;

    Node(int x) // parameterized constructor
    {
      data = x;
      next = null;
    }
  }
```

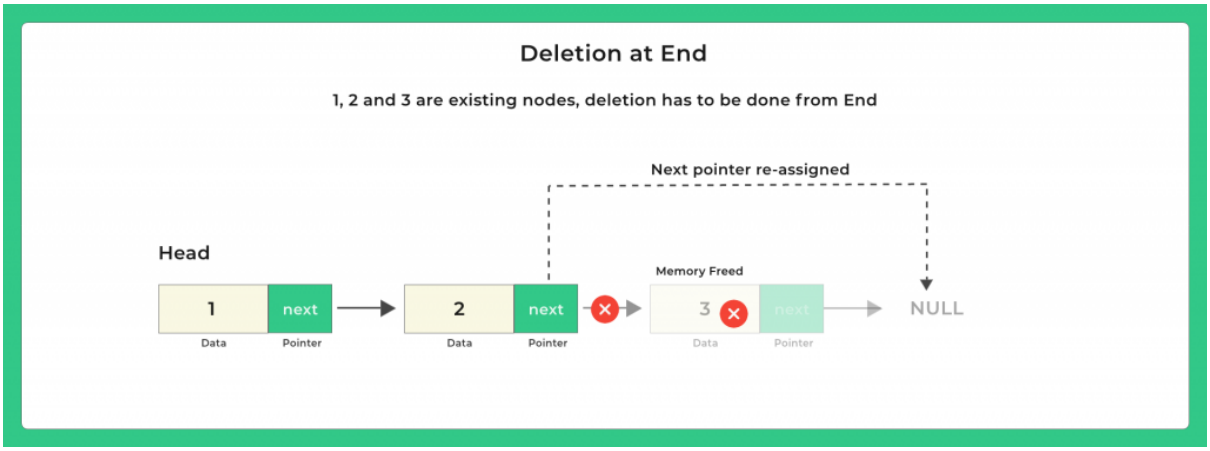
```
  public Node insertStart(int data) {
    // Creating newNode memory & assigning data value
    Node newNode = new Node(data);
    // assigning this newNode's next as current head node
    newNode.next = head;

    // re-assigning head to this newNode
    head = newNode;

    return head;
  }

  public void display() {
    Node node = head;
    // as linked list will end when Node reaches Null
    while (node != null) {
      System.out.print(node.data + " ");
      node = node.next;
    }
    System.out.println();
  }

  void deleteVal(int value) {
    Node temp = head;
    Node previous = null;

    if (temp == null) {
      System.out.println("Can't delete Linked List empty");
      return;
    }

    // Case when there is only 1 node in the list
    if (temp.next == null) {
      head = null;
      System.out.println("Deleted: " + value);
      return;
    }

    // if the head node itself needs to be deleted
    if (temp.data == value) {
      head = temp.next; // changing head to next in the list
      System.out.println("Deleted: " + value);
      return;
    }

    // traverse until we find the value to be deleted or lld ends
    while (temp != null && temp.data != value) {
      // store previous link node as we need to change its next val
      previous = temp;
      temp = temp.next;
    }

    // if value is not present then
    // temp will have traversed to last node NULL
    if (temp == null) {
      System.out.println("Value not found");
      return;
    }
```

```
    // for node to be deleted : temp lets call it nth node
    // assign (n-1)th node's next to (n+1)th node
    previous.next = temp.next;
    System.out.println("Deleted: " + value);
  }

  public static void main(String args[]) {
    LinkList_Delete_Value lld = new LinkList_Delete_Value();

    lld.insertStart(6);
    lld.insertStart(5);
    lld.insertStart(4);
    lld.insertStart(3);
    lld.insertStart(2);
    lld.insertStart(1);

    lld.display();

    lld.deleteVal(10);
    lld.deleteVal(5);
    lld.deleteVal(2);

    lld.display();

    lld.deleteVal(1);
    lld.display();

  }

}
```

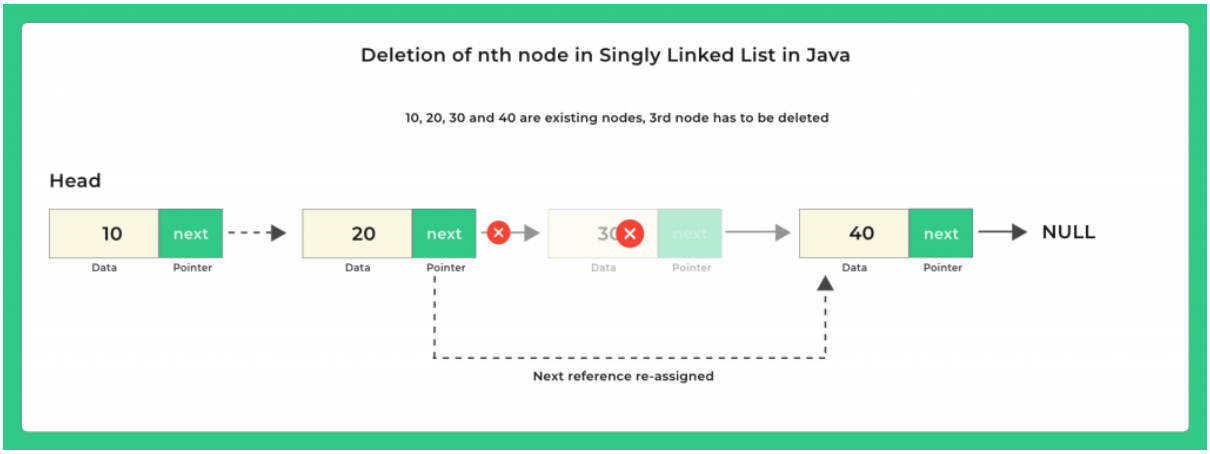
Deletion from the beginning in a Linked List



Deletion from end in Singly Linked List



Delete n^{th} node in Linked List




```
package linklist;

public class LinkList_Delete_Position {
  Node head;

  // Node Class
  class Node {
    int data;
    Node next;

    Node(int x) // parameterized constructor
    {
      data = x;
      next = null;
    }
  }

  public Node insert(int data) {
    // Creating newNode memory & assigning data value
    Node newNode = new Node(data);
    // assigning this newNode's next as current head node
    newNode.next = head;
```

```
    // re-assigning head to this newNode
    head = newNode;

    return head;
  }

  public void display() {
    Node node = head;
    System.out.println();
    if (node == null) {
      System.out.println("List is empty");
      return;
    }

    // as linked list will end when Node reaches Null
    while (node != null) {
      System.out.print(node.data + " ");
      node = node.next;
    }
    System.out.println();
  }

  void deletepos(int pos) {
    Node temp = head;
    Node previous = null;

    int size = calcSize(head);

    if (pos < 1 || pos > size) {
      System.out.println("Enter valid position");

      return;
    }

    // if the head node itself needs to be deleted
    if (pos == 1) {
      // changing head to next in the list
      head = temp.next;
      System.out.println("Deleted Item: " + temp.data);
      return;
    }

    // run until we find the value to be deleted in the list
    while (--pos > 0) {
      // store previous link node as we need to change its next val
      previous = temp;
      temp = temp.next;
    }

    // (pos-1)th node's next assigned to (pos+1)nth node
    previous.next = temp.next;
    System.out.println("Deleted Item: " + temp.data);

  }

  public int calcSize(Node node) {
    int size = 0;
    // traverse to the last node each time incrementing the size
```

```
    while (node != null) {
      node = node.next;
      size++;
    }
    return size;
  }

  public void deleteStart() {
    if (head == null) {
      System.out.println("List is empty, not possible to delete");
      return;
    }

    System.out.println("Deleted: " + head.data + " From Beginning ");
    // move head to next node
    head = head.next;
  }

  public void deleteLast() {
    if (head == null) {
      System.out.println("List is empty, not possible to delete");
      return;
    }

    // if LL has only one node
    if (head.next == null) {
      System.out.println("Deleted: " + head.data + " From the End");
      head = head.next;
    }
    Node previous = null;
    Node temp = head;
    // else traverse to the last node
    while (temp.next != null) {
      // store previous link node as we need to change its next val
      previous = temp;
      temp = temp.next;
    }
    // Current assign 2nd last node's next to Null
    System.out.println("Deleted: " + temp.data + " From the End");
    previous.next = null;
    // 2nd last now becomes the last node

  }

  public void deleteNthNode(int n) {
    int len = calcSize(head);

    // Can only insert after 1st position
    // Can't insert if position to insert is greater than size of Linked List
    if (n < 1 || n > len) {
      System.out.println("Can't delete\n");

    } else {
      if (n == 1) {
        // head has to be deleted
        System.out.println("Deleted: " + head.data);
        head = head.next;
        return;
```

```
      }
      // required to traverse
      Node temp = head;
      Node previous = null;

      // traverse to the nth node
      while (--n > 0) {
        previous = temp;
        temp = temp.next;
      }
      // assigned next node of the previous node to nth node's next
      previous.next = temp.next;
      System.out.println("Deleted: " + temp.data);
    }
  }

  public static void main(String args[]) {
    LinkList_Delete_Position ll = new LinkList_Delete_Position();

    ll.insert(60);
    ll.insert(50);
    ll.insert(40);
    ll.insert(30);
    ll.insert(20);
    ll.insert(10);

    ll.display();

    ll.deletepos(1);
    ll.display();

    ll.deletepos(3);
    ll.display();

    ll.deletepos(4);
    ll.display();

    ll.deletepos(-2); // not valid as pos negative
    ll.deletepos(10); // not valid as breaches size of Linked List

    ll.display();
    ll.deleteStart();
    ll.display();
    ll.deleteStart();
    ll.display();
    ll.deleteStart();
    ll.display();

    ll.insert(60);
    ll.insert(50);
    ll.insert(40);
    ll.insert(30);
    ll.insert(20);
    ll.insert(10);
    ll.display();
    ll.deleteLast();
    ll.display();
    ll.deleteLast();
```

```
    ll.display();
    ll.deleteLast();
    ll.display();
    ll.deleteLast();
    ll.display();
    ll.deleteLast();
    ll.display();
    ll.deleteStart();
    ll.display();



    ll.insert(35);
    ll.insert(34);
    ll.insert(33);
    ll.insert(32);
    ll.insert(31);
    ll.insert(30);

    ll.display();

    ll.deleteNthNode(3);
    ll.display();
    ll.deleteNthNode(4);
    ll.display();

  }
}
```