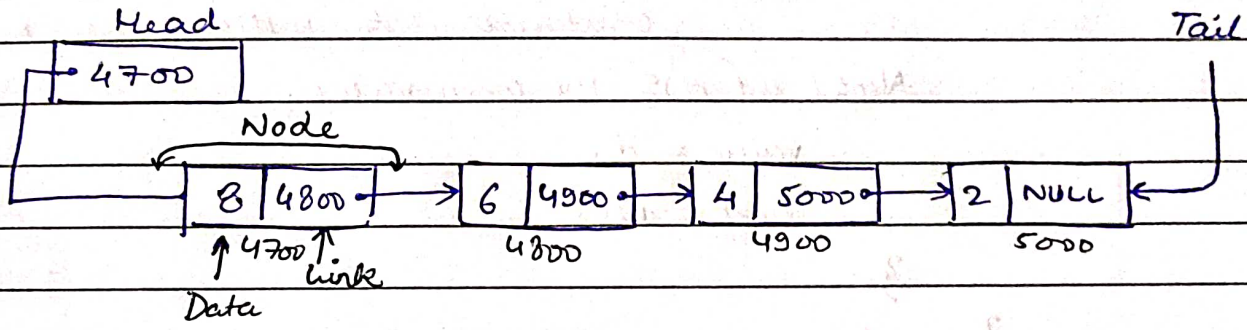
Linked list

1. Linear data structure that includes a series of connected nodes.
2. Nodes are randomly stored in memory.
3. Node consists two parts one the data and other the address.
4. Every link contains connection to another link.

Representation :-



Why linked list over array :-

Limitations of Array :-

1. The size of array should be declared first.
2. Increasing the size of array is complex → side by side
3. All the elements in the array are (contiguously) stored in the memory. Inserting element in array needs shifting of all its (predessors.) → previous

Advantage of linked list over array :-

1. Dynamic memory allocation, elements are stored non-contiguously stored.
2. Linked together with help of (pointers) → only in C language
3. No pre-defined size, size automatically grows as per the program's demand and limits to the available memory space.

Implementation of linked list in Java :-

```
class linked_list{
    Node head;  → head of linked list representing first node

    Static class Node{  → Inner node class
        int value; // Data stored in Node
        Node next; // Reference to next node

        Node(int d){  → Constructure to initialize new node with the given value.
            value = d;
            next = null;
        }
    }

    public static void main (String []args){
        linked_list mylist = new linked_list();  ← object
        mylist.head = new Node(1);  → first node with value 1
        Node s2 = new Node(2);
        Node s3 = new Node(6);
        // linking nodes together.
        mylist.head.next = s2;
        s2.next = s3;
        // traversal of linked list.
        while (mylist.head != null){
            System.out.print(mylist.head.value + " ");
            mylist.head = mylist.head.next;
        }
    }
}
```

Output → 1 → 2 → 6

Advantages of linked list :-

1. Dynamic size :- Dynamically grow or shirk
2. Efficient / Insertion or Deletion :- no need to shift elements
3. Memory efficient :- since size can change accordingly (dynamically)
4. Implementation :- can be used in stack and queues

Disadvantages of linked list :-

1. Higher Memory Usage :- Node occupy more space than array.
2. Traversal Complexity :- Accessing elements require traversal, making random access difficult
3. Limited Reverse Traversal :- Reverse is difficult in singly linked list.

Application :-

1. Sparse Matrix
2. Data storage for various structure, stack, queue, tree
3. Implement Dynamic Memory allocation.
4. undo/redo tasks.
5. Store details like student's details, employee details, product details

Complexity :-

Time Complexity :-

Insertion / deletion = (O(1))

Search = (O(n))

Space Complexity :-

Insertion / deletion / search = (O(n))

Linked list operations :-

1. Traversal :- access each element of linked list
2. Insertion :- insert element at beginning
   - at end
   - at middle
3. Deletion :- delete element at beginning
   - at end
   - at middle
4. Search :- search for a particular element in linked list.
5. Sort :- sort the elements.

Declaration :-

```
class Node{
    int data; //data stored in node
    Node next; // Reference to the next node
    //constructor to initialize a new node
    Node (int data){ //value stored in node.
        this.data = data; // initializing data field with value
        this.next = null; // initializing next field to null as
                          it is last node.
    }
}
```

Traverse :-

```
Node temp = head; // create a temporary variable and mark head
System.out.println("elements are");
while (temp != null){
    System.out.print(temp.data + " ---->");
    temp = temp.next; } // Move to next node in linked list.
```

Insertion :-

1. At the beginning :-

```
Node newNode = newNode(4);
newNode.next = head;
head = newNode;
```

2. At the end :-

```
Node newNode = newNode(4);

Node temp = head;
while (temp.next != null){
    temp = temp.next;
}
temp.next = newNode; // change next of the last node to
                        recently created node.
```

3. specified position in middle :-

```
Node newNode = new Node(4);
Node temp = head;
for (i = 2; i < position; i++){
    if (temp.next != null){
        temp = temp.next;
    }
} // Change next pointers to include the new node in between
newNode.next = temp.next;
temp.next = newNode;
```

Deletion :-

1. Delete in beginning :-

```
head = head.next;
```

2. Delete from end :-

```
Node temp = head;
while (temp.next.next != null){ //traverse to second last
    temp = temp.next;          element.
}
temp.next = null; //change next pointer to null
```

3. Delete from middle :-

```
Node temp = head;
for (int i=2; i<position; i++){ //Traverse to the element
    if (temp.next != null){ before the element to be deleted
        temp = temp.next;
    }
} //Change next pointer to exclude the node from list:
temp.next = temp.next.next;
```

Search :-

```
boolean searchNode (Node head, int key){
    Node current = head;
    while (current != null){
        if (current.data == key){
            return true;
        }
        current = current.next;
    } return false; }
```

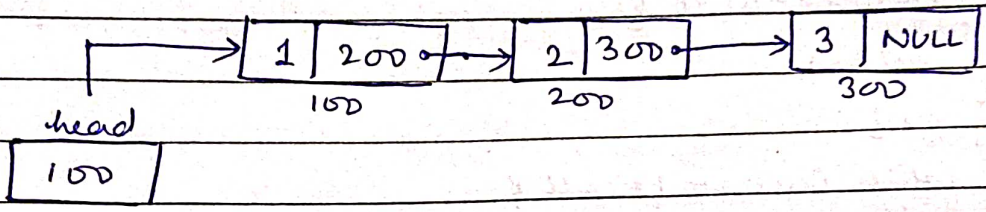
Sorting :-

```
void sortLinkedList (Node head){
    Node current = head, index;
    int temp;
    if (head == null){
        return;
    } else {
        while (current != null){
            index = current.next;
            while (index != null){
                if (current.data > index.data){
                    temp = current.data;
                    current.data = index.data;
                    index.data = temp;
                }
                index = index.next;
            }
            current = current.next;
        }
    }
}
```

Types of Linked Lists :-

1. Singly linked list :-

→ Contains two parts data and address (of successor node)

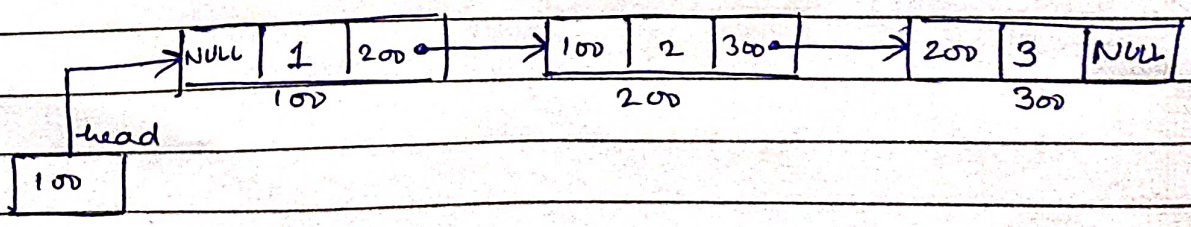



```
class Node {
    int data;
    Node next;

    Node (int data){
        this.data = data;
        this.next = null;
    }
}
```

2. Doubly linked list

→ Contains three parts, previous node address, data, next node address.

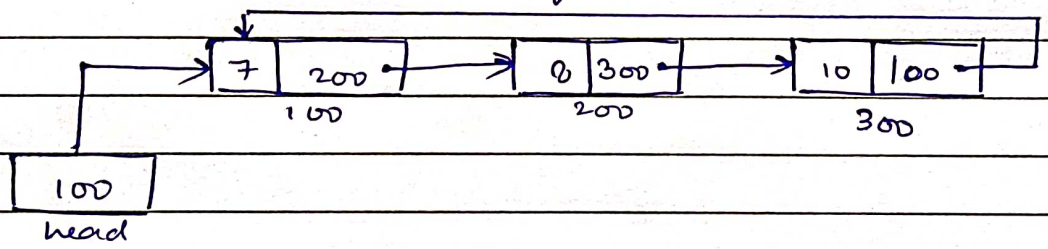


```
class Node{
    int data;
    Node next;
    Node prev;

    Node (int data){
        this.data = data;
        this.next = null;
        this.prev = null;
    }
}
```

Circular linked list :-

→ Same as singly linked list but last node is not connected to null instead its connected to the first node.



Doubly Circular linked list :-

→ Features both circular and doubly linked list.