8/04/2023

Linked list

1) Suppose that in our OS we need to make an array of 10 MB size but the memory available is 10 MB but is not contiguous & hence we can't use arrays here but linked list can work here & hence it can work on non-contiguous memory locations.

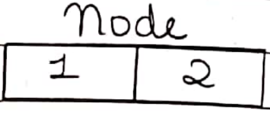
2) At run-time, we can do creation & deletion of node at run-time and hence there would be no wastage of memory which was there in case of arrays.

3) In arrays, insertion of element takes O(n) time complexity but in linked list insertion can be done in O(1) time complexity provided the pointer is at that position only.

4) There is no concept of indexing in case of linked list but concept of address is used here.

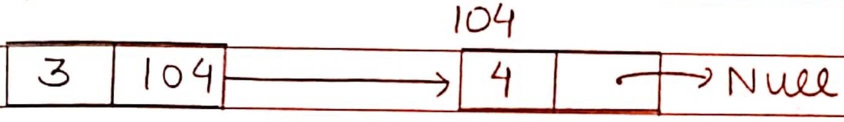
Definition

Linked list can be defined as collection of nodes.



1 → data
2 → address of next node

Simple linked list



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

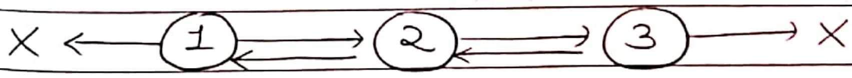
As we create an integer pointer by int *, we create pointer to Node by Node* .

Types of linked list

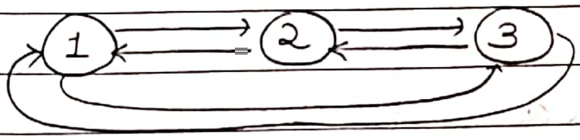
1) Singly linked list

2) Circular linked list

3) Doubly linked list (prev & next pointers)



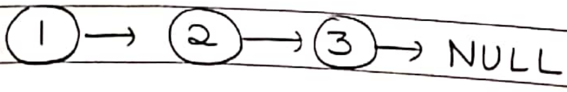
4) Doubly circular linked list (prev & next pointers)



Note→ Linked list is Hindi & this is a magical line but don't tell this to any interviewer. Also linked list is a linear data structure as only one descendant is there.

Creation of Linked List (Better method is also there)

```
class Node {
        public :
            int data;
            Node* next;
            Node() {
                this→data = 0;
                this→next = NULL;
            }
            Node (int data) {
                this→data = data;
                this→next = NULL;
            }
};

main () {
      Node* first = new Node (1);
      Node* second = new Node(2);
      Node* third = new Node (3);

      first → next = second;
      second → next = third;
}
```



Printing a linked list

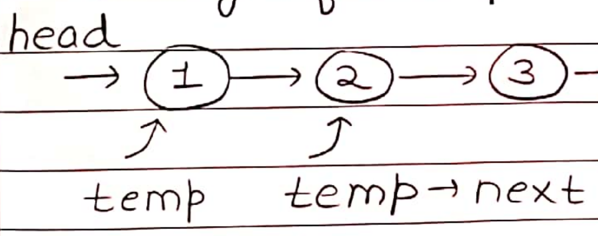
1) Print the data of current node.
2) Move pointer forward.
3) Stop when we reach null.

```
void print (Node* & head){
          //Temperory pointer
          Node* temp = head;
          while (temp != NULL){ // Step-3
               // Step-1
               cout << temp→ data << " ";
//Step-2       temp = temp→next;
      }
}
```

Output

1 2 3

Meaning of temp = temp→ next



temp = temp → next

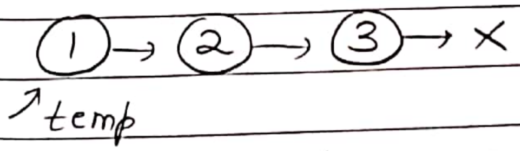
= head → next

Hence we are moving forward in the linked list.

Why we created temp ?

It is a good practice not to change the head and we created temp which is pointing to head & hence we move temp forward leaving head at same place.

Ex→



temp→ next → data = 2

temp→ next → next → data = 3

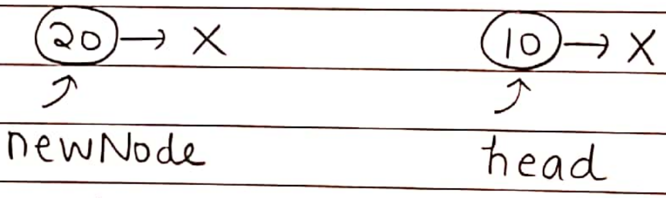
Betler way of creating linked list

(i) Insertion at head

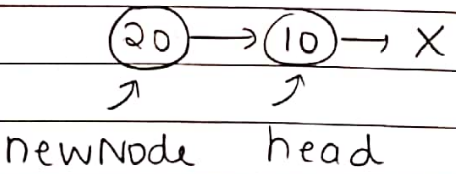
Initial scenario → 10 → X

Inserting 20 at the head

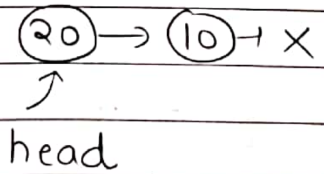
1) Creating node with value 20



2) newNode to be connected to head



3) Update head



```
void insert (Node* & head, int data){
        //Step-1
        Node * newNode = new Node(data);
      //Step-2
      newNode → next = head;
    //Step-3
    head = newNode;
}
```

Suppose now we inserted 20, 30, 40 & 50

using the function, then the linked list would become.

50 → 40 → 30 → 20 → 10 → X

Printing linked list will give the output as 50 40 30 20 10

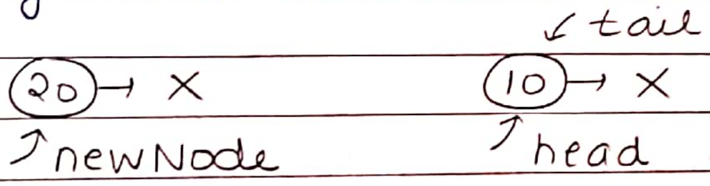
Note → head ⇒ starting point of linked list
tail ⇒ ending point of linked list

point = node here.

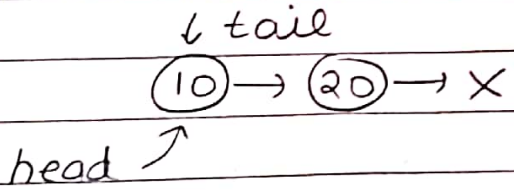
(ii) Insertion at tail

Initial scenario → 10 → X
Inserting 20 at tail

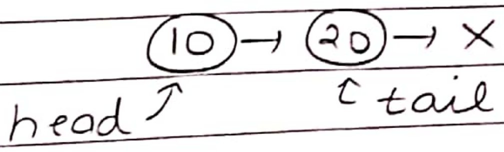
1) Creating node with value 20



2) newNode to be connected to the tail



3) Update tail



void insert (Node * &head, Node * &tail,
int data) {

```
    // Step-1
    Node* newNode = new Node(data);
    //Step-2
    tail→next = newNode;
    //Step-3
    tail = newNode;
}
```

Note→ If initially head & tail was initialized to NULL, then we need to handle this explicitly as when we are creating the new node, initialize the head & tail with that node & this would be when we are creating the 1st node.

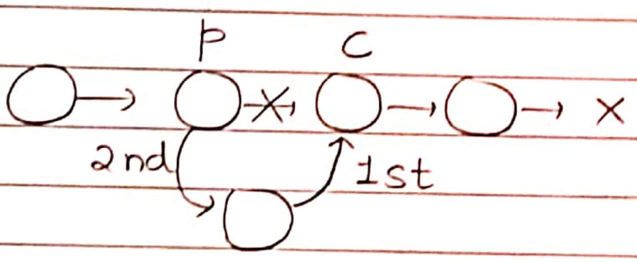
```
if (head == NULL) {
        head = newNode;        } 1st node creation
        tail = newNode;
}
```

Better way & easy to understand for handling empty linked list case.

```
if (head == NULL) {
      Node* newNode = new Node (Data);
      head = newNode;
      tail = newNode;
      return; //As node has been created
}
```

(iii) Inserting at specified position.

1) Check for empty linked list case.
2) If non-empty then
   (i) Traverse to that position.
   (ii) Create a node



   (iii) newNode → next = c ;
   (iv) p → next = new Node ;

What if we do p → next = new Node first and then update new Node → next to curr but by this, we will be losing the track of further linked list.

Some edge cases
1) pos = = 0 → insert At Head
2) pos = = len → insert At Tail

} Reusing func^n.

Length of linked list

```
int findLength (Node* &head){
            int len = 1;
            Node * temp = head;
            while (temp → next != NULL){
                    temp = temp → next;
                    len++;
        }
        return len;
}
```

Deletion operation in Linked list

(i) Deleting head of linked list

10 → 20 → 30 → X   temp = head

1) head = head → next
2) temp → next = null ;
3) delete temp

20 → 30 → X

(ii) Deleting tail in linked list

↓ prev
10 → 20 → 30 → X   temp = tail

1) Find prev
2) prev → next = null
3) tail = prev
4) delete temp

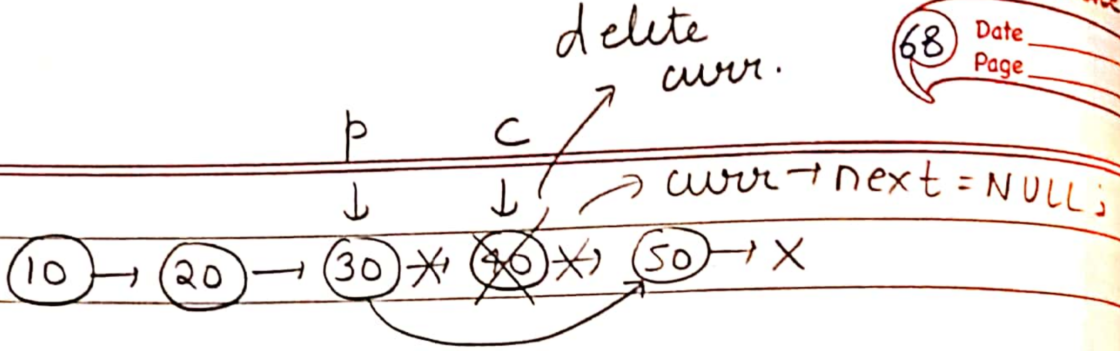
10 → 20 → X

```
void deleteOp(int pos, Node* & head,
                    Node* & tail) {

      // Empty linked list
      if (head == NULL) {
            return ;
      }
     // Delete head
     if (pos == 1)
```

```
    {
      Node* temp = head;
      head = head → next; //S-1
     temp → next = NULL; //S-2
     delete temp; //S-3
  }


   // Deleting tail
   int len = findLength(head);
   if (pos == len){
        // Find prev (S-1)
        int i = 1;
        Node* prev = head;

        while (i < pos-1) {
             prev = prev → next;
             i++;
        }
      //S-2
       prev → next = NULL;
     //S-3
       tail = prev;
    //S-4
      delete temp;

  }
```

(iii) Deleting middle node (specified position)

1) Find prev & curr.
2) Do prev → next = curr → next;
3) curr → next = NULL;
4) delete curr.



prev→next = curr→next.

Updated linked list

```
int i=1;
Node * prev = head;
//Finding prev
while (i< pos -1) {
        prev = prev → next;
        i++;
}
//Finding curr
Node * curr = prev → next;
//Step-2
prev→ next = curr → next;
//Step-3
curr → next = NULL;
//Step-4
delete curr;
}
```