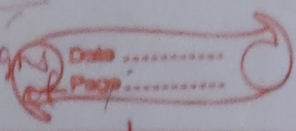
for (struct node*) malloc(sizeof(struct node))

↓ It returns a struct node pointer

↓ This assigns a space of Structure node in heap memory



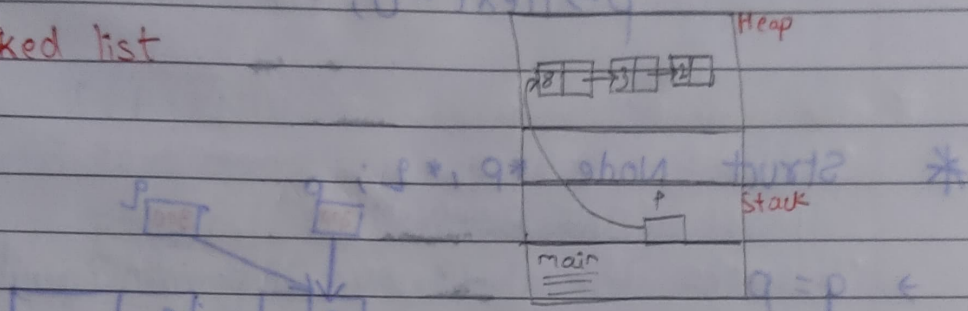
Linked List

Q: Why linked list?

- As array is of fixed size during run time.
- ~~So to get an array of~~

* Array could only be created in heap as well as stack whereas Linked list is only stored in heap memory.

Linked list



* Struct node → Self Referential structure as it has pointer that point to ~~same datatype structure~~

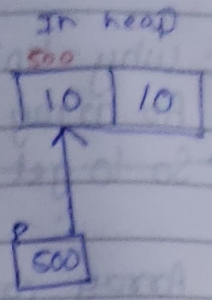
```
int data;
struct Node *next;
};
```

How to create node?

→ In stack : struct node *p

p = new Node

This makes a node space in heap that is pointed by p

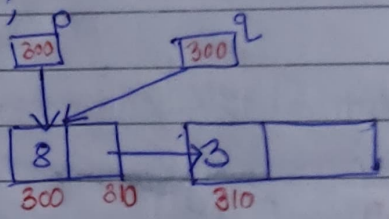


p→data = 10; } To access member of a node

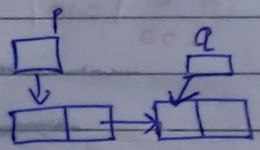
p→next = 0;

* Struct Node *p, *q;

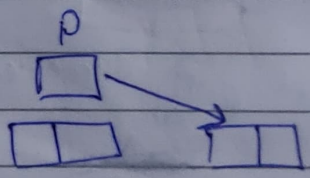
→ q = p



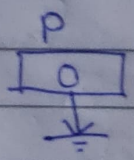
→ q = p→next



→ p = p→next



* If pointer p is null i.e



if (p == Null)

if (p == 0)

if (!p) → i.e (!0) = (1)

also some check conditions

if (p→next == Null) if (p→next != Null)

* Display linked list

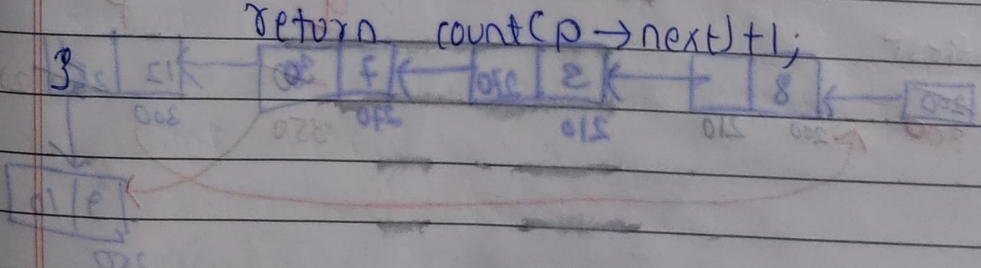
```
while (p != 0)
{   printf ("%d", p->data);
    p = p->next;
}
```

* Recursive Display linked list

```
void Display (struct node *p)
{
    if(P != NULL)
    {
        printf ("%d", p->data);
        Display (p->next);
    }
}
```

* Counting of nodes

```
int count(struct Node *p)
{
    if(p==0)
        return 0;
    else
        return count(p->next)+1;
}
```



INT32_MIN
INT32_MAX

* Adding data of nodes

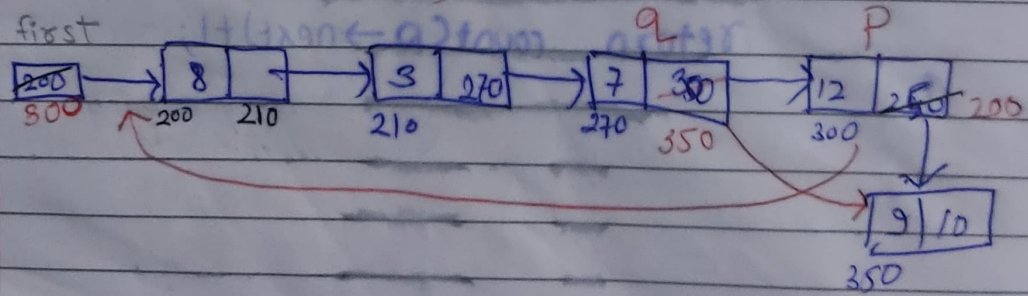
```
int Add (struct Node *p)
{
    int sum = 0;
    while (p)
    {
        sum = sum + p->data
        p = p->next;
    }
    return (sum)
}
```

* Max element in linked list
  - Define minimum as smallest number in integer (INT32_MIN)
  - Iterate through every element (Node) and compare.
  - Also Recurssively.

- Linear Search
- Iterate through each node and return if founded the element.

Improving linear Search

search (12)



```
if(key == p->data)
{ q->next = p->next;
  p->next = first;
  first = p;
}
```

* Insert

Before first node

```
Node *t = new Node;
t-> data = x
t-> next = first
first = t
```

At some position

After position (x)

- iterate till position x ~~that it be p~~
  x->next = node we want to insert (t)
  t->n

After position (x) insert (t) node

- iterate till position x
  t->next = x->next
  x->next = t

- To iterate tell position x is n
  ∴ Time = O(n)

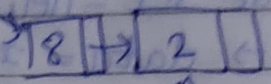
∴ minimum Time O(1)
Max " O(n)

- Create a linked list by inserting at last

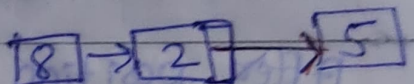
```
struct Node{
    int data;
    struct Node *next;
}*first = NULL, *last = NULL;

void InsertLast(int x)
{
    node *t = new Node;
    t->data = x;
    t->next = Null;
    if (first == Null)
        first = last = t;
    else
    {
        last->next = t;
        last = t;
    }
```

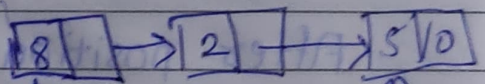
Before insert



While executing insert last



After executing



* Sorted list (Inserting in sorted list)

- Compare value to be inserted with node data
- if value is less than a particular node data then insert
- Cases to be consider
  → if its the first element
  → if it is to be inserted in first position as the element to be inserted is smallest

* Deleting node

- case 1: deleting at first position or first node

```
Node *p = first;
first = first->next;
x = p->data;
delete p;
```

Time → O(1)

ARIHAN INI 32 MIN

Deleting at a particular position
(2 pointer required)

```
Node *p = first;
Node *q = Null;
for (i=0; i<pos-1; i++)
{
    q=p
    p=p->next;
}
q->next = p->next;
x = p->data
delete p
```

Time spend → O(n) // as time depends on position i.e n

* To check if sorted
  - Iterate through linked list nodes
  - compare with previous nodes data
  - If current node is less than previous node then linked list is not sorted
  - Else at the end node it will turn out to be sorted

* Remove Duplicate from sorted linked list

- Iterate through each node
- If data at one node is similar to previous node
- then delete second duplicate node

```
Node * p = first
Node *q = first->next
while (q != Null)
{
   if (p->data != q->data)
   {     p = q;
         q = q->next;
   }
   else
   {  p->next = q->next;
      delete q;
      q = p->next;
   }
}
```

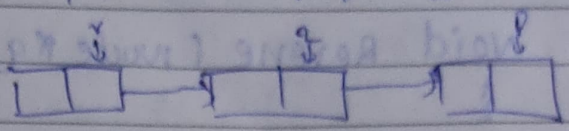
Time : O(n)

* Reverse linked list
- Reversing element
- Reversing links.

- Reversing elements.
  → Iterate and copy element data in an array (auxiliary array)
  → decrement array pointer and copy it in linked list

```
p=first
i=0
while (p!=Null)
{
    A[i] = p->data;
    p = p->next;
    i++;
}
p = first ; i--;
while (p!=Null)
{
    p->data = A[i--];
    p = p->next;
}
```

- Reversing links
- Sliding pointer (3 pointers next to each other)



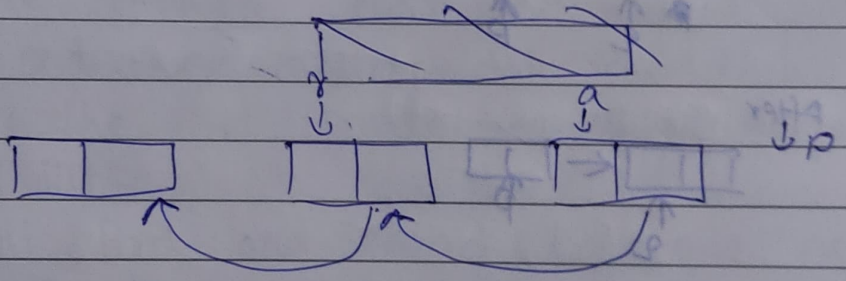
- Iterate through pointer (p) with pointer q and r as sliding pointers following next to it
- After every loop of

```
while (p!=Null)
{
    r=q;
    q=p;
    p=p->next;}
    q->next=r
}
```

execute (to reverse links)

- Make q the last node as first

```
first=q
```



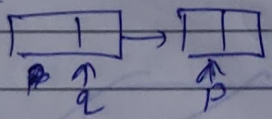
Time : O(n) (to iterate)

- Recursive Reverse of linked list

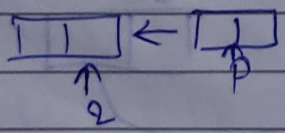
```
void Reverse (node *q, Node *p)
{
    if (P != Null)
    {
        Reverse (p, p->next);
        p->next = q  //while returning
    }
    else
    {
        first = q;
    }
```

- First reach till end of node using recursion then
- Reverse links

Before



After



* Concatenation of 2 Linked list

- Reach the end of first linked list node
- Make the next of last node of first linked list point to the beginning of second linked list.
- Time O(n)

```
p=first
while (p!=0)
{
   p=p->next
}
p->next = second;
second = NULL; //As no need of this pointer
                  anymore as p is sufficient
```

* Merging of 2 linked list in sorted way.

- Iterate through first and second linked list and the smallest data should be made the first node of third (merged) linked list
- Until first and second both are not NULL
  - → if first is less than second then
    - merge the link of third node to current node of first
    - make third's node the current node
    - Increment current Node
    - Make last->next = NULL
  - → Similarly if second is less than first then

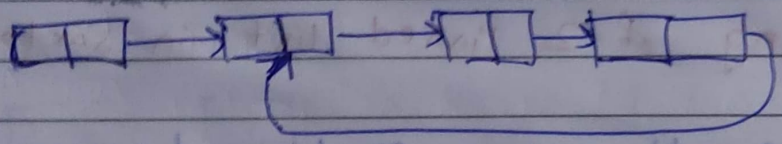
```
if (first != Null)
    last->next = first
else
    last->next = second
```

Time $\Theta(M+N)$

- M → no of first elements
- N → no of second elements (number of nodes pointed by second pointer)

* Check for loop in linked list

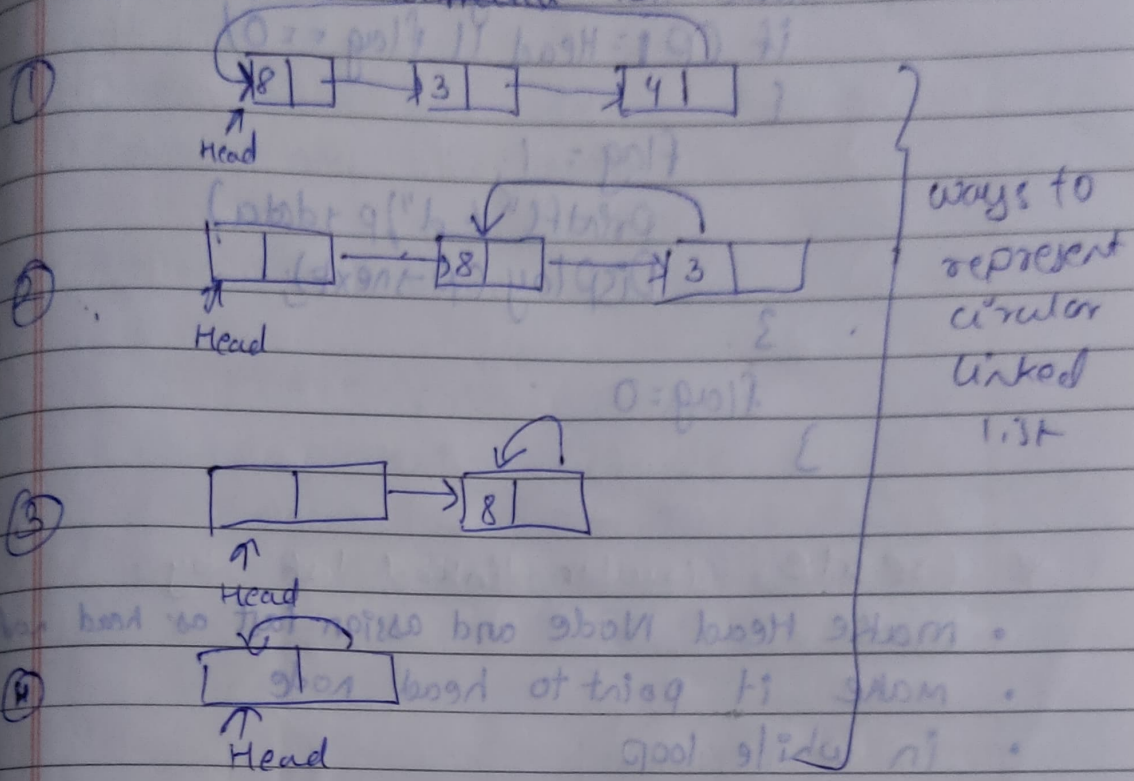
- move two pointer with one at slow speed and other at greater speed
- If at any point they meet each other then there's a loop



```
isloop (Node *f)
{ Node *p, *q;
  p = q = f;
  do
  {
    p = p->next;
    q = q->next;
    q = q!=NULL ? q->next : NULL;
    if(q!=Null)  q=q->next  else  q=Null
  } while(p && q && p!=q);
```

return p=q ? true : false;

* Circular linked list
- last node is connected to first node.



* Display Circular linked list

```
void display(Node *p)
{
    do
    {
        print("%d", p->data);
        p = p->next;
    } while (p != Head);
}
Display(Head);
```

* Display Circular linked list using recursion

```
void Display (Node * p)
{
    static int flag = 0
    if (p != Head || flag == 0)
    {
        flag = 1;
        Printf("%d", p->data)
        Display (p->next);
    }
    flag = 0
}
```

* Create circular linked list by
  - make Head Node and assign last as head node
  - make it point to head node
  - in while loop
    - → Assign address in heap to a node
    - → Enter data
    - → then make the node point to the node pointed by last node
    - → Make last node point to current Node
    - → make current node last node

* Inserting in circular linked list : Similar as normal linked list

* Inserting after last node or before head node
  - Iterate till last node then make the last node point to the new node

- make new node the head node based on case if the new node is to be inserted before head or after last condition

* Deleting from circular linked list

cases

- Deleting Head node
- Deleting any other node.

→ Deleting any other node:

```
p = Head;
for (i=0; i<pos-2; i++)
    p = p->next;
q = p->next;
p->next = q->next;
x = q->next;
delete q;
```

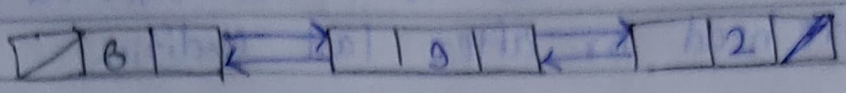
Reasons:
- less than sign because index starts from zero whereas index starts from 1
- -1 because once the loop is run it will take pointer to second node
- -2 because we want to access node previous to node that needs to be deleted

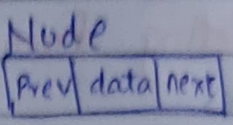
→ Deleting head node

```
while (p->next != Head)
    p = p->next;
p->next = Head->next;
x = Head->data;
delete Head;
Head = p->next;
```

Time O(n)

Doubly linked list






```
Struct Node
{
   Struct  Node *prev;
   int  data;
   Struct Node *next;
};
```

creating →

```
Struct Node *t
t = new Node;
t→prev = Null;
t→data = 10;
t→next = Null;
```

* Inserting in Doubly linked list

- Cases
  - Before 1st Node
  - Any other position

- Before 1st node

```
Node *t = new Node,
t→data = x
t→prev = Null;
t→next = first
First→pre = t
first = t
```

- Any other position
  - After some position

```
Node *p = first
for (i=0; i<pos-1; i++)        → Time Depends on this O(n)
        p = p->next;
Node *t = NewNode;
    t->data = x
t->next = p->next;
t->pre = p
if (p->next != Null)
        p->next->prev = t;
p->next = t;
```

* Deleting a node
  - Cases
    1) Delete 1st Node
    2) Delete any other node.

- Delete 1st Node

```
Node *p = first;
first = first->next;          → Time O(1)
x = p->data;
delete p;
if (first)
        first->prev = Null;
```

- Deleting any other node.

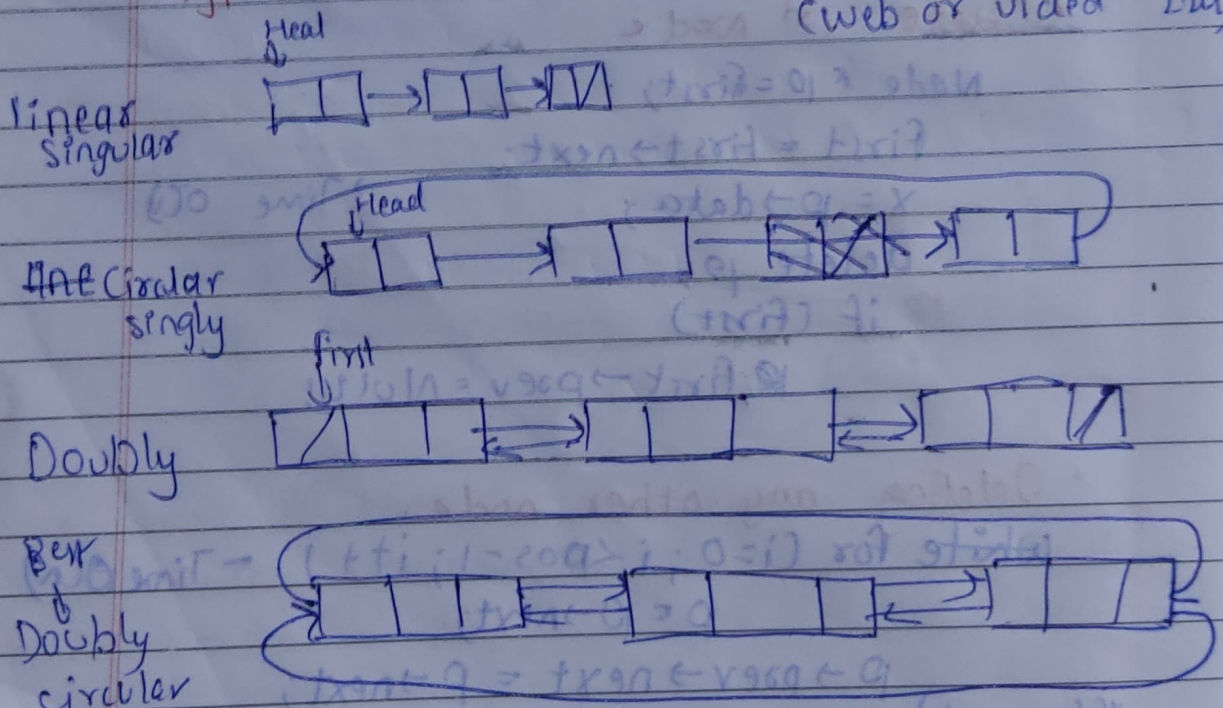
```
while for (i=0; i<pos-1; i++)   - Time O(n)
        p = p->next;
p->prev->next = p->next;
if(p->next) p->next->prev = p->prev;
delete p
```

* Reverse

```
p = first;
while(p)
{
  temp = p->next;
  p->next = p->prev;
  p->prev = temp;
  p = p->prev;   // as p change i.e moved forward
  if (p->next == NULL)  // if p is last node
      first = p          // make it first
}
```

condition (P != NULL) is given becz at last lonode p = p->prev which will make the p Null and will make the first Null, so to avoid this p != NULL condition is given

Types of linked list (Comparision) (web or video 2hr)



Doubly & Doubly Circular has ~~only~~ same space (or contains same space)

* Difference between Linked list & Vs Array (web or video 225)
(Depens on situation which is better)

* Find Middle node.

- In 2 scans.
  → Find length.
  → Reach node [length/2]

- In 1 scan
  → take two pointer p & q.
  - make q jump two node & p jump 1 node
  - when q reaches Null p is already at middle

```
p = q;
while(q)
{
    q = q->next
    if(q) q = q->next
    if(q) p = p->next
}
printf("%d", p->data)
```

- Using 1 scan an stack.
- Insert all linked list data in stak
- the pop half the stack size the 1st element in stack will loe the addoess of middle element

* Find Intersection of two linked list

- take pointer and start pushing the address of ~~stack~~ node of 1st linked list in stack.
- Do similar with second linked list
- start poping out address
  - maintain its copy
- Once the address become uncommon the last poped address copy is the intersecting point

```
* p = first

while (p) push (&stk1, p)
  p = second;
while (p) push (&stk1, p)
  while (stackTop(stk1) == stackTop(stk2))
    { p = pop (&stk1);
      pop (&stk2);
    }
```