Problem

Consider ordered list :

10, 15, 25, 35, 40, 50, 60, 75, 80 ← 45 goes here

→ we have to insert 45

→ we have to make room for 45 ~~in~~ to insert it

→ can be done by

i) shifting element left

ii) shifting element right.

→ This is very long hence insertion (and deletion) is time expensive in ordered list

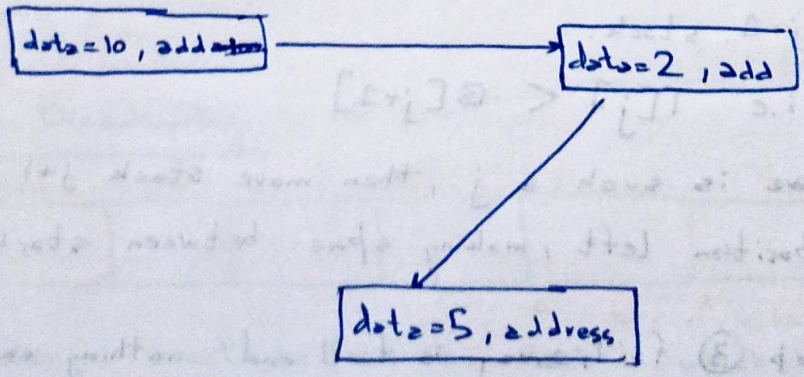
→ We also have fixed size of ~~list~~ array (sequencial list), which wastes memory

→ This can be solved by ~~Linked~~ Linked Representation

Linked Representation

→ Data are linked to other data by something (address)

Eg



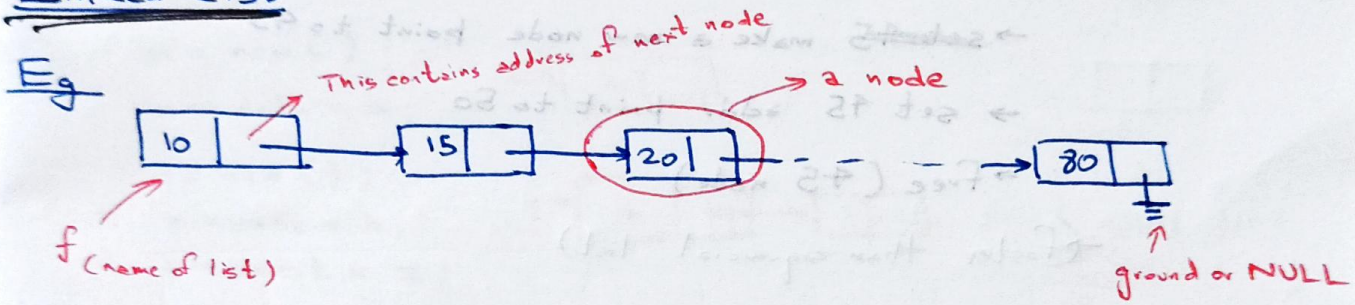
→ elements scattered everywhere in memory

→ more data is required for this implementation,

n element , 2n data (n data, n address)

Linked List

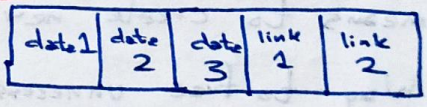
Eg



Code

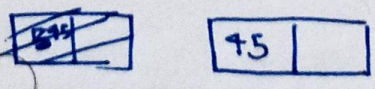
→ in structure, every fields are sequential ordered

```
typedef struct nodetype
{
    int data1;
    int data2;
    int data3;
    struct nodetype *link1, *link2;
} node;
```



Insertion

→ we want to insert 45 (between 40 and 50)

I) create node → 

II) set addr. of new node point to 50

III) set addr. of 40 point to 45

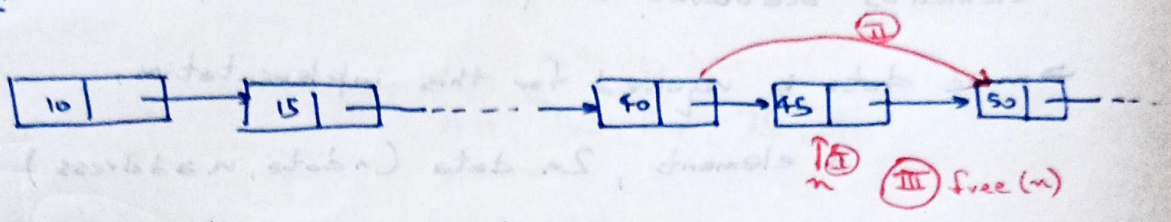
step can not be interchanged.

→ if we do III befor II, we lose memory and it creates memory leakage.
(memory is there, but can't be accessed)

(no movement of data, compared to sequential list)

Deletion

Eg:



→ we want to ~~dete~~ delete 45

→ ~~set 45~~ make a new node point to 45

→ set 45 addr point to 50

→ free (45 node)

(faster than sequencial list)

To do these complier need three things.

i) mechanism to define a node [structure]

ii) means to create new node [malloc(), calloc()]

iii) Way to free unnecessary nodes [free()]

Create new node

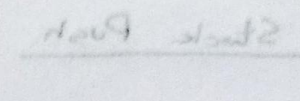
```
node*   new ()                    //assume node structure defined
{
   node* temp ;
   temp = (node *) malloc (sizeof (node)) ;
   return temp;
}
```

Alternate

```
void   new (node **f)
{
  *f = ~~malloc~~ (node*) malloc (sizeof (node));
}
```

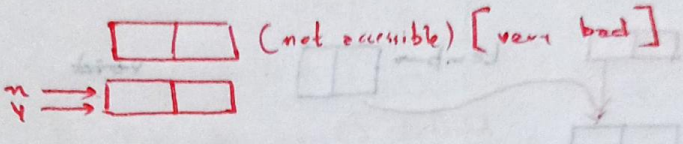
Eg

```
n = new ();
y = new ()
```



(don't do this)

```
n = y ;
```

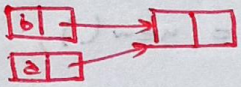


Eg:

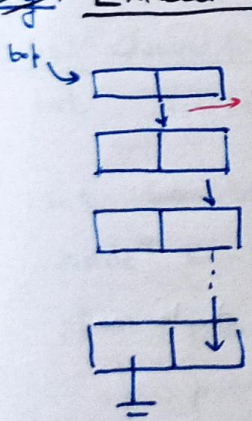
```
n = new ();
n → data = a;
```

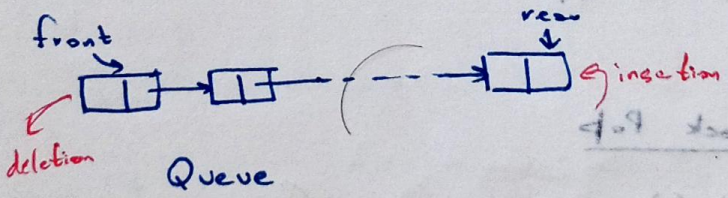

```
y = new ();
y → data = b;
y → next = z;
```


```
n → next = ~~y → next~~ y → next
```



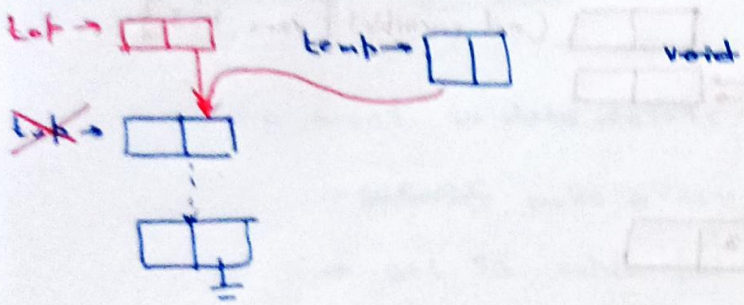
~~Eg~~: Linked Stack and Queue





Stack

Linked Stack Push



void

```
void push (int item)        // consider node with only data and next
{
    node*  temp;
    temp = new();           // consider new as before
    temp -> data = item;    (dynamic allocation, no case of stack full)
    temp -> next = top;
    top = temp;
}
```

Linked Stack Pop

```
int  pop ()                 (even in dynamic allocation, stack can
{ if (top == NULL)           be empty)
    |-- " Empty"
    |   return;
  else
    temp = top;
    item = top -> data;
    top = top -> next;
    free (temp);
    return item;
}
```

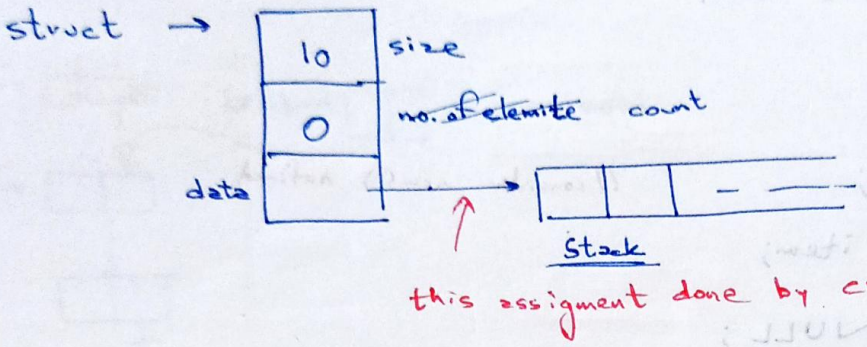
Linked Queue Insert

```
void insertQ(int item)
{
  node *temp;
  temp = new();          // consider new() defined
  temp -> data = item;
  temp -> next = NULL;

  if (front == NULL)
      front = temp;
  else
     rear -> next = temp;

  rear = temp;
}
```

Linked Queue Delete

```
int deleteQ()
{
  int item;
  node* temp;
  if (front == NULL)
     print("Empty")
     return NULL

  temp = front;
  item = front -> data;
  front = front -> next;
  free(temp)
  return item;
}
```

Eg: Integer Stack [Programming Assignment] (Question in middle)



this assignment done by create_stack()

Structure definition

```
struct stack
{
   int size;  → 4 byte
   int count; → 4 byte
   int * data; → 4 byte
};
```

typedef struct stack*, int_stack; → make alias of "struct stack*"

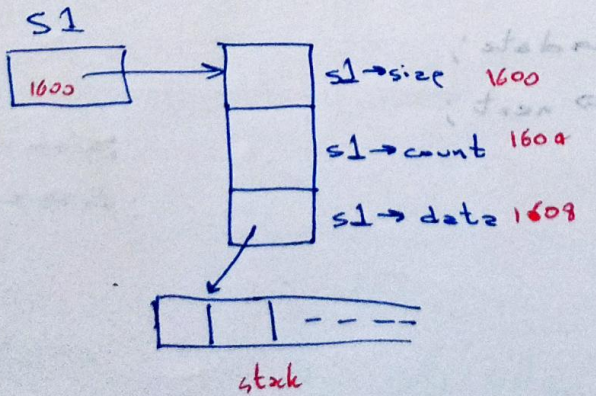
→ now we can make a stack by the following:

→ int_stack s1;

→ type : struct stack*

→ size = ~~12 byte~~ 4 byte → it's just a pointer

~~creat~~



(data stored contiguously in structure)

create_stack()

This is a function → we have to first assign a data to int_stack s1 using malloc

```
int_stack create_stack (int size)
{
    int_stack s1;  → s1 = (int_stack*) malloc (sizeof (struct stack));
    s1 → size = size;
    s1 → count = 0;
    s1 → data = (int*) malloc (sizeof (int) * size);
                → insert null check by malloc
    return s1;
}
```

Eg: Integer Stack (Again)

→ The previous Eg. had extra complication, due to typedef, so let's try again

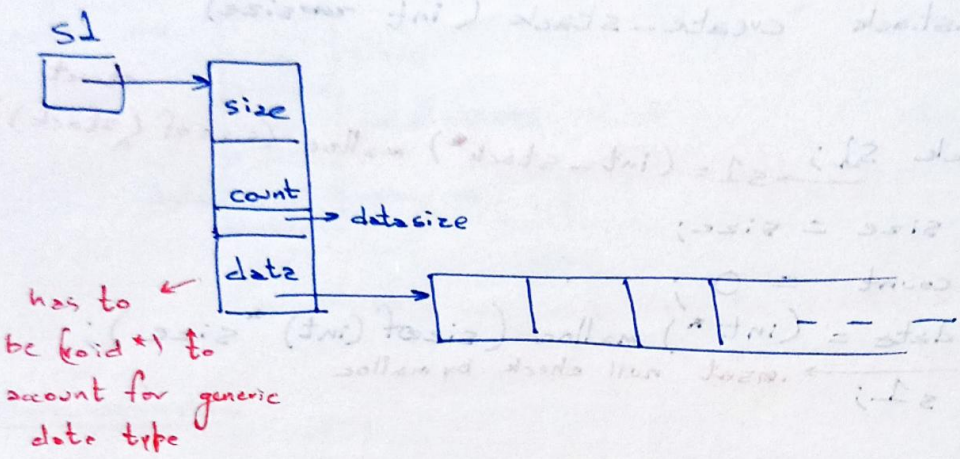
→ push(), pop(), is_free(), is_empty() are done using data and count.

→ to free the data, do this

- → free (s1 → data);
- → free (s1);

→ order fixed, otherwise segmentation fault.

Eg: Generic Stack [Programming Exercise]



→ as we have (void*) data, we can't do the following

→ s1 → data[1]

→ s1 → data[1] = 100

as data is dereferenced [void*]

→ we have to use function memcpy() to solve this → check man pages

~~create_gstack()~~

~~→ typedef of struct stack*~~

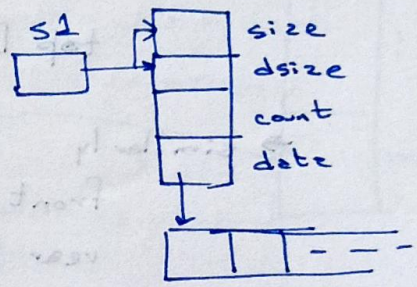
~~gstack~~

stack definition

→ typedef struct stack* gstack; // consider struct stack defined

create gstack()

```
gstack create_gstack (int size, int dsize)
{
    gstack s1;
    s1 = (gstack) malloc (sizeof (struct stack));

    s1 -> size = size;
    s1 -> dsize = dsize;
    s1 -> count = 0

    s1 -> data = malloc (dsize * size);
    return s1;
}
```



push()

```
int push_gstack (gstack s, void *dp)
{
    int b = s -> count * s -> dsize;
    memcpy (&s -> data [b], dp, dsize);
    s -> count --;
}
```

→ to print dp, we can ~~use~~ make three print function and use pointer to functions, using dsize.

Multiple Queue and Stacks in a linked list

Suppose I want to build n stacks and m queues

∴ we need n tops, m rear, m front

← most probably array containing pointers

→ So we can use an array:

top [i] = top of i^{th} stack, $1 \leq i \leq n$

→ similarly

front [i] = front of the i^{th} queue.

rear [i] = rear of the i^{th} queue

$1 \leq i \leq m$

→ The initial conditions could be:

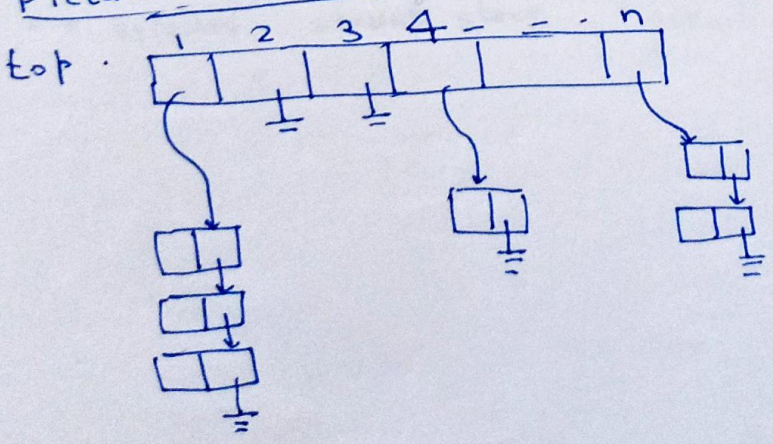
→ top [i] = NULL $1 \leq i \leq n$

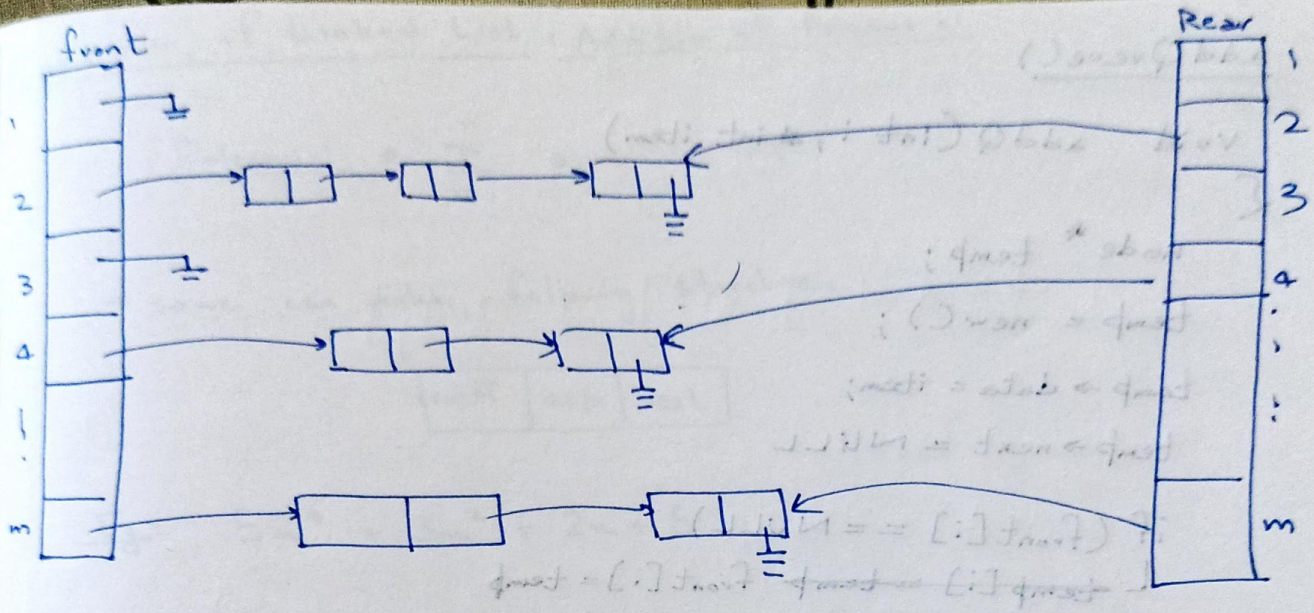
→ front [i] = NULL $1 \leq i \leq m$

∴ if (top [i] == NULL)
↳ i^{th} stack empty

∴ if (front [i] == NULL)
↳ i^{th} queue is empty

Pictorial Representation





push()

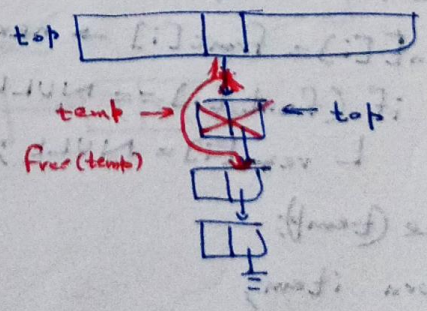
```
void push (int i, int stack)
{
    node * temp;          // assume node declared as before
    temp = new ();        // new() as before
    temp -> data = item;
    temp -> next = top[i]  // top array is an array of pointers to node
    top[i] = temp;
}
```

→ top array defination

```
node* top[100];          //global parameter
```

pop()

```
int pop (int i)
{ node * temp; int item;
  if (top[i] == NULL)
    [ stack[i] is empty
      return NULL;
  temp = top[i]
  item = top -> data;
  ~~temp[i] =~~ top[i] = top[i] -> next
} free (temp); return item;
```



addQueue()

```
void addQ(int i, int item)
{
    node * temp;
    temp = new();
    temp -> data = item;
    temp -> next = NULL

    if (front[i] == NULL)
        front[i] = temp
    else
        rear[i] -> next = temp

    rear[i] = temp
}
```

delete Queue

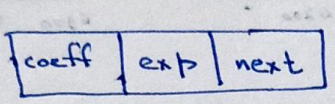
```
int deleteQ(int i)
{
    node * temp;
    if (front[i] == NULL)
        print queue empty
        return NULL;
    temp = front[i];
    item = temp -> data;
    front[i] = front[i] -> next;
        if (front[i] == NULL)
            rear[i] = NULL;
    free(temp);
    return item;
}
```

→ we can say that linked implementation better for this wrt sequential implementation

Application of Linked List : Addition of Polynomial

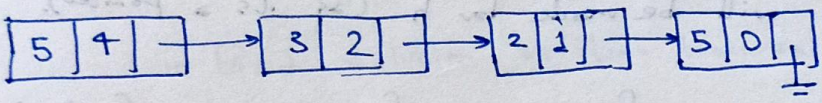
Polynomial → $\sum_{i=0}^{n} a_i x^i = 0$

→ so we can make following structure,



Eg: $5x^4 + 3x^2 + 2x + 5$

→

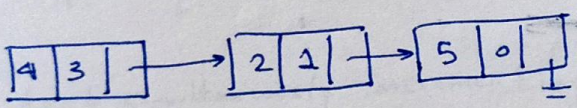


→ if degree is same, we add their coefficient

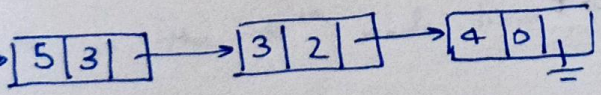
~~→ otherwise no~~

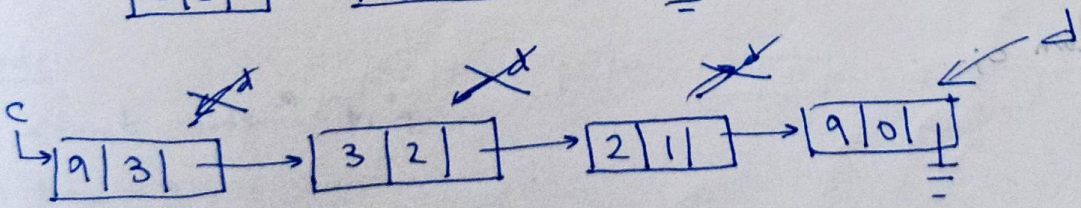
Eg of algo

P: $4x^3 + 2x + 5$



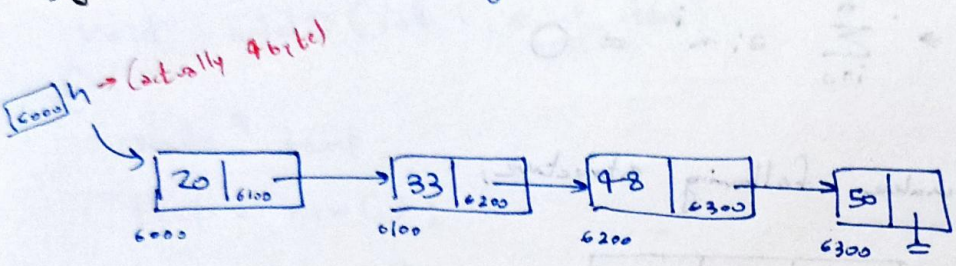
Q: $5x^3 + 3x^2 + 4$





Counting no. of nodes

Consider following linked list



→ when we do

```
struct node * h;
```

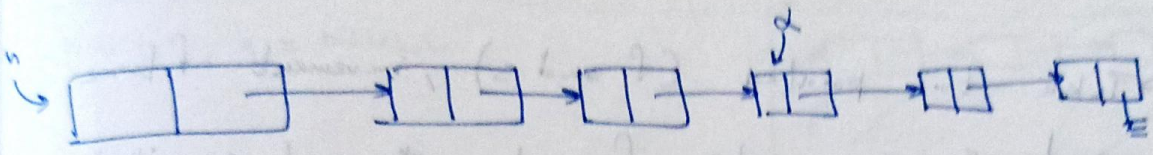
4 byte will be made for h (as it's a pointer).

→ for counting no. of nodes, go from start to finish, incrementing a count every time a node is parsed.

code

```
int count (struct node * s)        ← This will be call by value
{
    int c = 0;
    while (s != NULL)
    {
        c++;
        s = s -> next
    }
    return c;
}
```

Find the k^{th} node of a linked list from the last



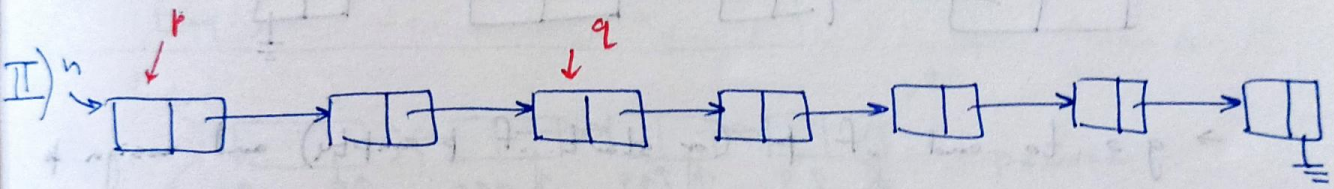
suppose $k = 3$ → output will by α

Possible Solutions

I) find total nodes (n), find $(\text{total nodes} - k)$ and then traverse again to $(n-k)^{th}$ node from start $O(2n)$

→ Bad solution as two traversal required.

II)



suppose $k = 3$

let p & q be other pointe. (difference b/w p & q = k)

→ keep p simultaneously incrementing p & q, till q reaches last node.

→ p is the k^{th} node from last. $O(n)$

Code

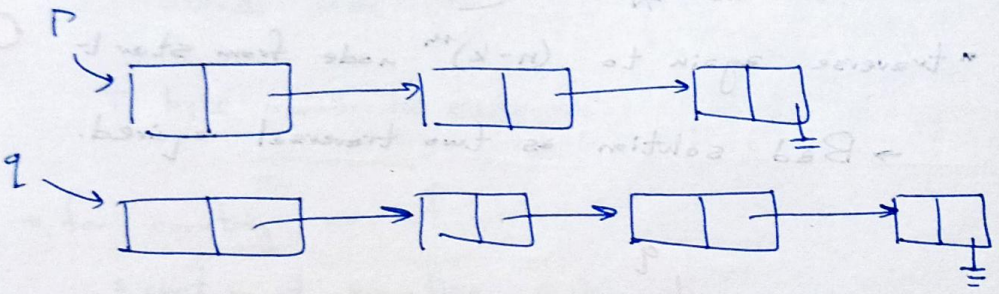
```
struct node *p, *q
p = q = h
for (int i = 1; i <= k; i++)
    q = q->next
while (q->next != NULL)
{
    q = q->next
    p = p->next
}
```

Finding middle of the node in the linked list

→ Take two pointed (f and s), increvement f twice and s once, when f reaches the end, s will be in middle.

→ some slight nuances, like out of bounds for f

Merge two linked lists (one after another) (p after q)



→ go to end of p (or start if p empty) and assign it

p → next = q

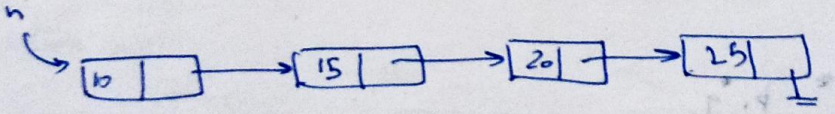
3 case

i) p = null, q ≠ null

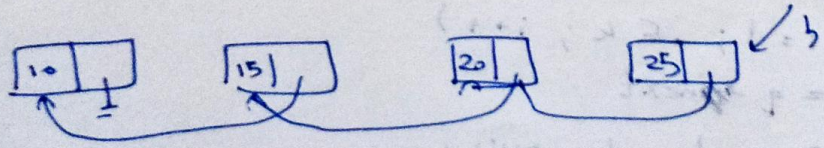
ii) q = null, p ≠ null

iii) both ≠ null

Reversing a linked list element



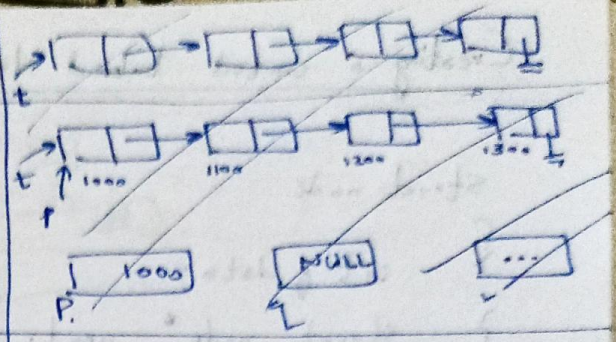
make



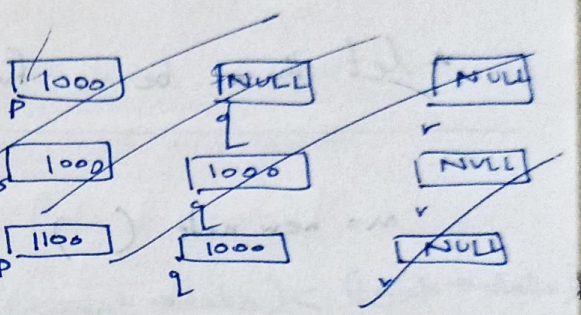
→ address will be same

node *p, *q, *r;

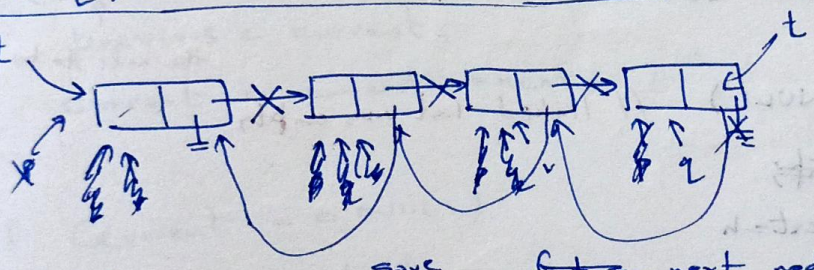
p = t , q = NULL



while (p != NULL)
{
    r = q
    q = p
    p = p → next
    q → next = r
}
t = q;



Eg



p → NULL

→ select a node, save a future next node for future, make changes to that node accordingly.

→ In end, do

t = q

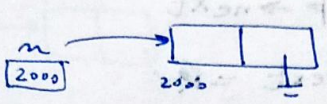
so as to update t

Creating a sorted linked list [insertion sort]

```
struct node
{
    int data;
} struct node* next;
```

* let there be a function newnode()

```
n = new node ( )
```

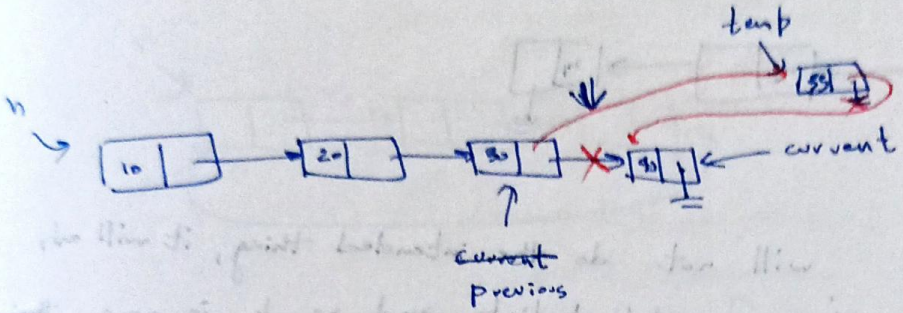



```
struct node*  insert (struct node* h, struct node* ~~new~~ temp)
```

the node to be inserted

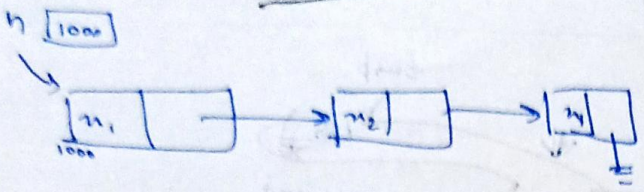
```
{
    if (h == NULL)   // linked list was empty
    {   ~~h = temp;~~
        temp -> next = h
        h = temp;
        return h;
    }
    else if ( h -> data  >=  temp -> data )   // temp should be added in start
    { temp -> next = h
      h = temp;
      return h;
    }
else
{
```

struct node* current , * previous




```
current = h.
previous = NULL;

while (current != NULL && (current->data) < (temp->data))
{
    previous = current;
    current = current -> next;
}

If (current == NULL)
{  previous -> next = temp;
   temp -> next = NULL;
}
else
{
   temp -> next = current;
   previous -> next = temp;
}
} //end of else
//end of function
```

~~Create~~ ~~Delete~~ free linked list



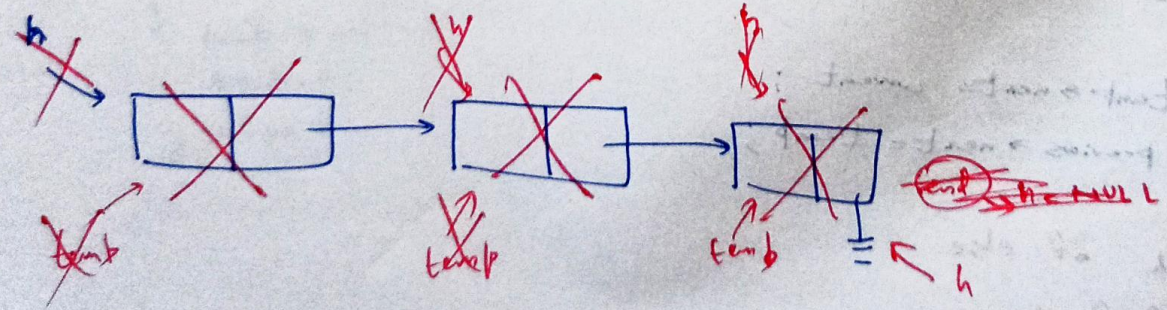
→ free (h) will not do the intended thing, it will only free h, not the linked list, and as h is gone, ~~there~~ there will be no way to reaccess the linked list

[memory Leak]

→ you have to ~~remove~~ free individual nodes first

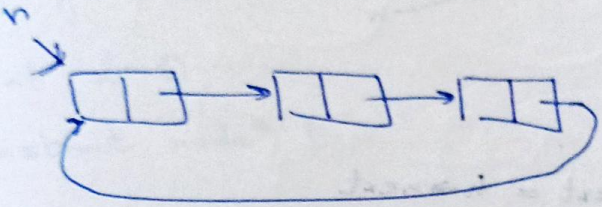
code

```
struct node * temp.
while (h != NULL)
{
   temp = h
    h = h -> next;
   free (temp);
}
```

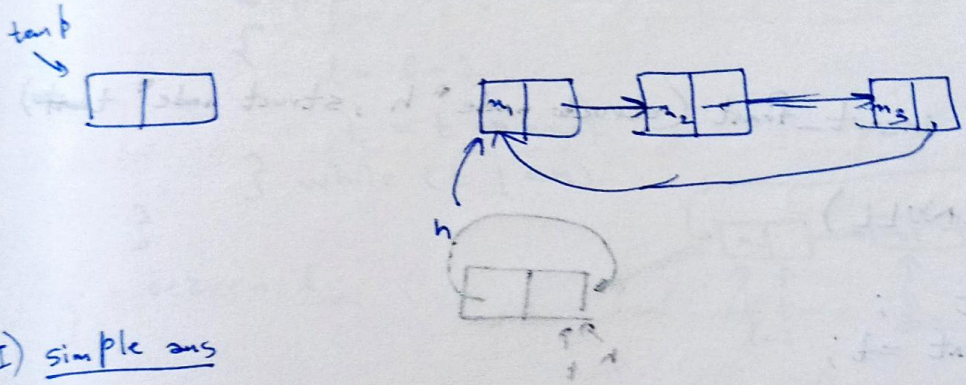


h = NULL

Circular Linked List



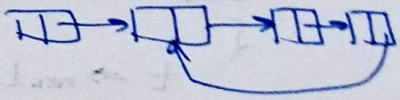
inserting a new node at the front of a circular linked list



I) simple ans

```
temp -> next = h
h = temp
```

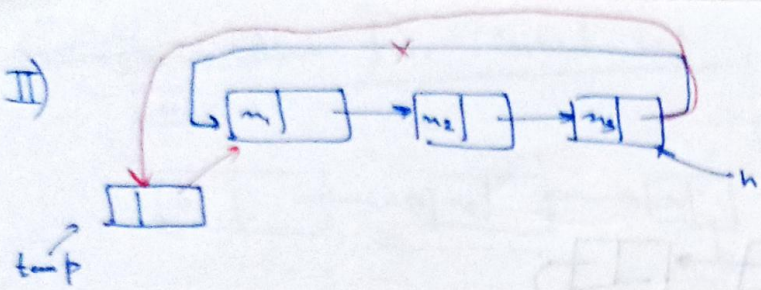
problem : Linked List will look like this and it is not a circular



solution : traverse to the end and make changes

problem to solution : Insertion became O(n).

II)



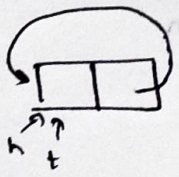
temp → next = h → next

h → next = temp

→ This solution ~~~~ saves time Complexity too, [O(1)]

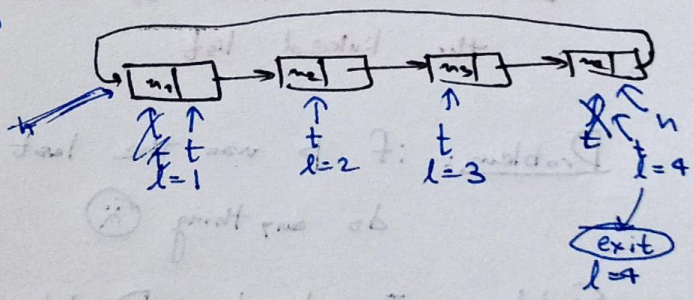
Code

```
struct node* insert_front (struct node* h , struct node* t)
{
    if (h==NULL)
    {
        h = t ;
        t → next = t;
    }
    else
    {
        t → next = h → next ;
        h → next = t ;
    }
}
```

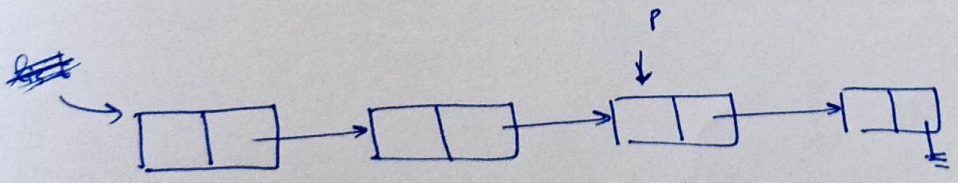


Length of circular Linked List

```
int length (struct node* h)
{ int l=0;
  struct node* t;

  if (h != NULL)
  { t= h; l=l+1; t=t->next;
    while (t != h) do
    {
      l= l+1;
      t = t -> next
    } while (t != h)
  }
  return l;
}
```



Problem with Single Linked List

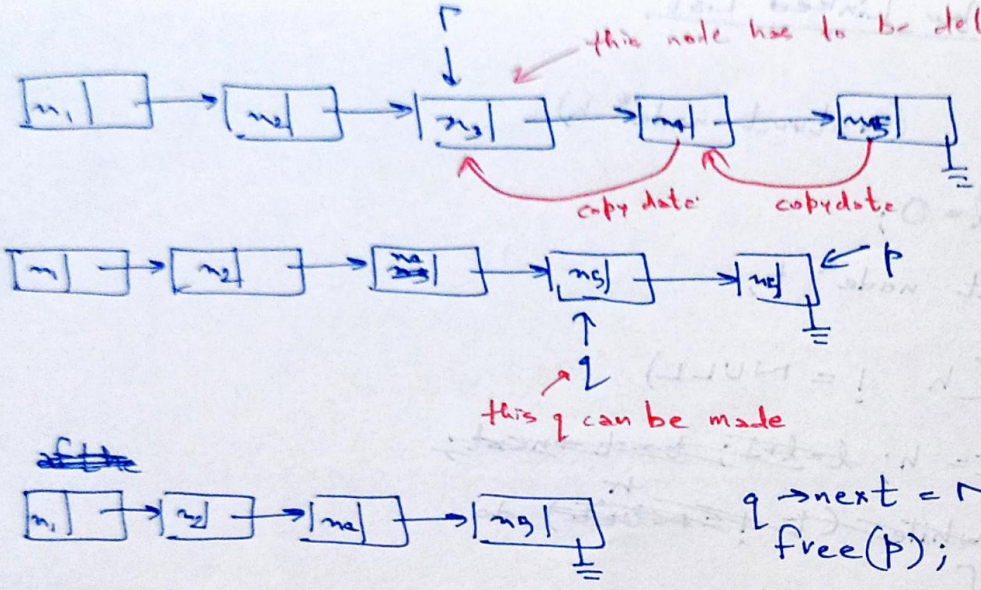


→ it is not possible to delete a node using one pointer. (no start given)

→ In general, there is no way to go back in linked list without making new pointers.

Work around

rather than deleting the node, we copy next node's contents to the present node and continue forward, and in end, we delete last node.




```
q->next = NULL;
free(p);
```

★ this was done as there was no way to access start of the linked list.

Problem: if p was the last node of linked list, we can't do anything ☹

Solution: Incorporate Double Linked List