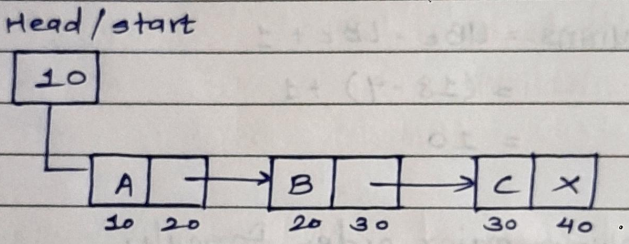
12-10-23

Page No.
Date

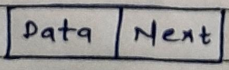
Unit 03:

* Linked list:
- Linked list is a linear data structure, which is a collection of data elements c/a 'nodes.'
- Successive elements in LL are always connected by pointer and the last element always point to 'null.'
- There is a special node called 'head' or 'start' node which always points to 1st ele of the list.



* Basic terms in LL:

1. Node:
- Every ele. in LL is called node.
- Node consists of 2 parts. First part is info./data part & second part is next or pointer part.



- Data part contains actual elements and next part contains the address of next element.

2. NULL pointer:
- When the LL is empty or no node in it, then NULL value is set to pointer. i.e PTR = NULL

3. Empty list:
- When there is no node in a linked list, then it is c/a 'empty linked list.'

* Advantages of LL:

1. Linked list provides a facility to insert and delete nodes dynamically.
* User can increase or decrease the size of list by inserting or deleting a node.

2. Linked list follows an efficient memory allocation of data as in LL, data are not stored in contigious memory location like an array. Here, nodes are stored randomly. That's why, it reduce the wastage of memory space.

3. No need to pre-define the size of linked list.

4. Insertion and deletion is very optimal in linked list. Unlike an array, we need not shift or change the position of an item.

19-10-23

* Limitations of LL:

1. Time complexity - As traversing in LL take more time.

2. Reverse traversing is difficult - Using single LL, it is not possible to traverse in a reverse direction & to do this we need a doubly LL; but use of backward pointers again need an extra space for storage.

3. Implementation - It's imp' is not easy as it include the concept of objects, classes, constructors and structure also.

Viva not in syllabus * Memory allocation: (MA) ~~malloc()~~

We can allocate a memory in 2 ways:

1) Static MA:
- Memory is allocated during compile time, i.e we can't allocate and de-allocate memory during execution.
- Here, stack data structure is used.
- Static MA is less efficient than dynamic MA

2) Dynamic MA:
- Memory is allocated at runtime.
- Allocation and de-allocation can be done using library function of 'stdlib.h.'
- This fuc^n allocates memory from heap and de-allocates whenever not required so that it can be used for other purpose.

Page No.
Date

Library functions ofor memory allocation :

1. Malloc (MA) malloc()
   - It allocate single block of requested memory of specified size.

   Syntax :- void * malloc (size)

   Here, size → block size

   e.g :- ptr = (int*) malloc (5 x sizeof(int))

   for n=5 ∴ 5 x 4 = 20 byte

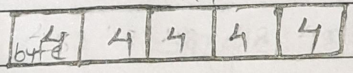
   ↳ 20 byte available space.

2. Calloc (continious allocation) calloc()
   - It performs same operation like malloc but ; difference is that it initializes each block with default value zero.
   - It has 2 parameters as compared to malloc.

   Syntax :- void * calloc (n, size)

   e.g :- ptr = (int*) calloc (n, sizeOf(int));

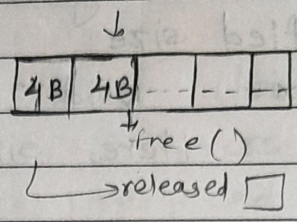
   5 x 4 → 4 bytes for 1

3. Free function ⇒ free()
   - It is used to de-allocate memory space which is allocated by malloc & calloc function.
   - Malloc and calloc function are unable to de-allocate memory by their own.
   - Whenever de-allocation is done by using free function it helps to reduce wastage of memory space by freeing it

- Syntax :- free (int)

e.g :- ptr = (int*) calloc (n, sizeOf (int))



4. Realloc
- To change memory allocation of previously allocated memory, Realloc function is used.
- If memory allocated by calloc and malloc function is insufficient then realloc function is used to reallocate memory dynamically.

Syntax :- ptr = realloc (ptr, newsize);

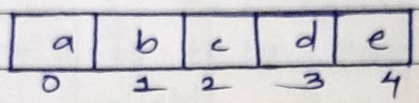
e.g :- ~~(int*)~~ nptr = (realloc (ptr, newsize)
(20, 28)
initial size, new size

14-10-23

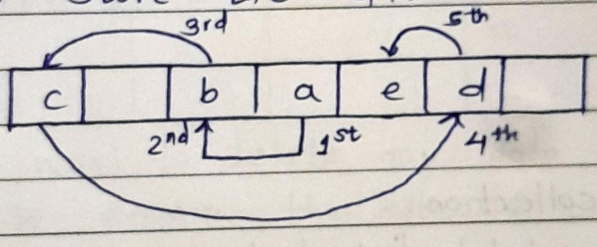
Page No.
Date

* Memory representation using LL:
- List ele. are stored in a memory in an arbitary order.
- LL always use go from one ele. to other ele.
- Consider, memory layout L = {a, b, c, d, e}

Using array representation.



- Array rep^n gives continious memory location to store data; but LL uses an arbitary layout to store the data.



For example, consider a list where each node of a list contain a single char. We can obtain actual string by using LL representation.

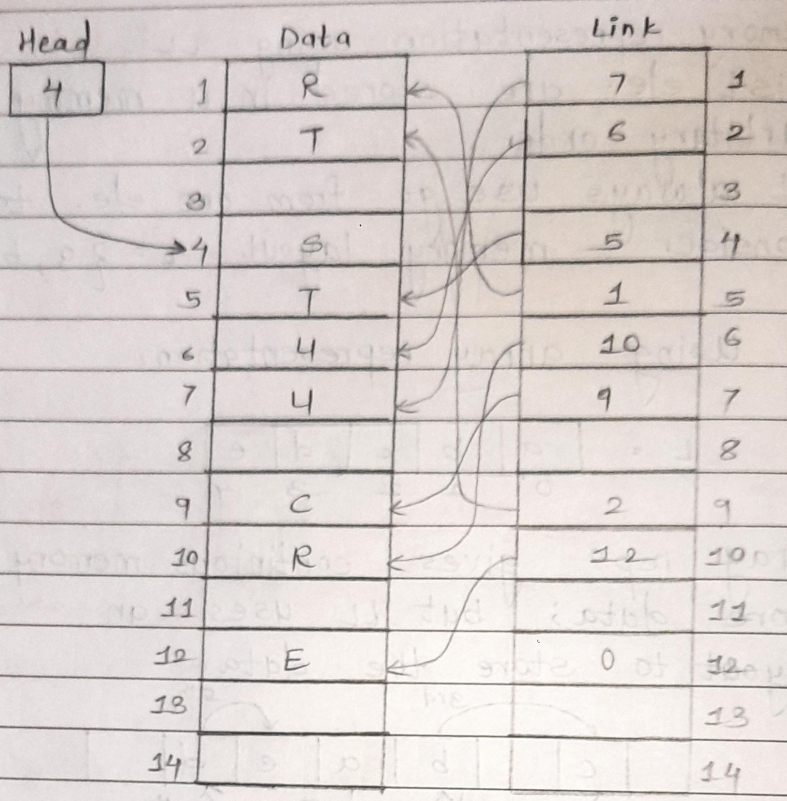
Let, S = "STRUCTURE"

19-10-22

1 5 1 7 9 2 6 10 12

STRUCTURE

Page No.
Date



* Garbage collection:

We can maintain linked list in memory by assuming the possibilities of inserting new node into the list. For that we want some memory space, so that we can insert new node in the list.

Also, if we want to delete any node from list, then that memory space is used for future purpose. So, there is a special space in a memory which consists of unused memory cell which is called as "available space."

Suppose, some memory space is reusable because of node is deleted from the list and we want some space for the future insertion. Then we want to insert deleted node space in a free storage list.

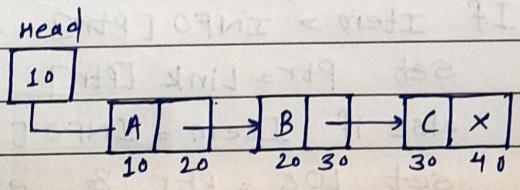
The technique which collects all deleted space into free storage list is called as "Garbage collection." Garbage collection takes place in such a way that, first computer runs through all the list, marking all the cell those are currently used and also marks the free space in the memory.

* Overflow:
- When new data is to be insert into the list; but there is no available space i.e free storage list is empty, then 'overflow' occurs.
- We can handle this situation by adding the space when we're working with data structure.

* Underflow:
- When we want to delete any data from the list and the list is empty, the situation is called as 'underflow.'

* Operations on linked list:

1 Traversing:
- To count the number of elements.
- e.g



∴ PTR = start
∴ PTR = 10
INFO[PTR] = A
PTR = LINK[PTR]
PTR = 20

20-10-23

Page No.
Date

Algorithm for traversing:

step 1 : Set PTR = START /HEAD
step 2 : Repeat step 3 & 4 until PTR ≠ NULL
step 3 : APPLY PROCESS TO INFO[PTR]
step 4 : Set PTR = LINK[PTR]
step 5 : Exit

2. Searching :

- Searching alg. is used to find out the location of the node where item first appear in list.

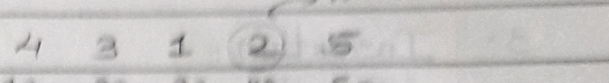
* Unsorted list -

- Let, we've a list ; but the data in the list is not sorted & if we want to search any particular element in the list, then we're to traverse in a list using pointer variable.

Algorithm :

Search (INFO, Next, Ptr)
step 1 : set ptr = start
step 2 : Repeat step 3 until ptr ≠ NULL
step 3 : If Item > INFO [ptr] then
set Ptr = Link [Ptr]
else if Item = INFO [Ptr] then
set LOC = Ptr & exit
else
set LOC = NULL & exit
step 4 : Exit.

e.g Unsorted list :



1. set ptr = 10
2.

* Sorted list -

Algorithm :

Search (INFO, Next, Ptr)

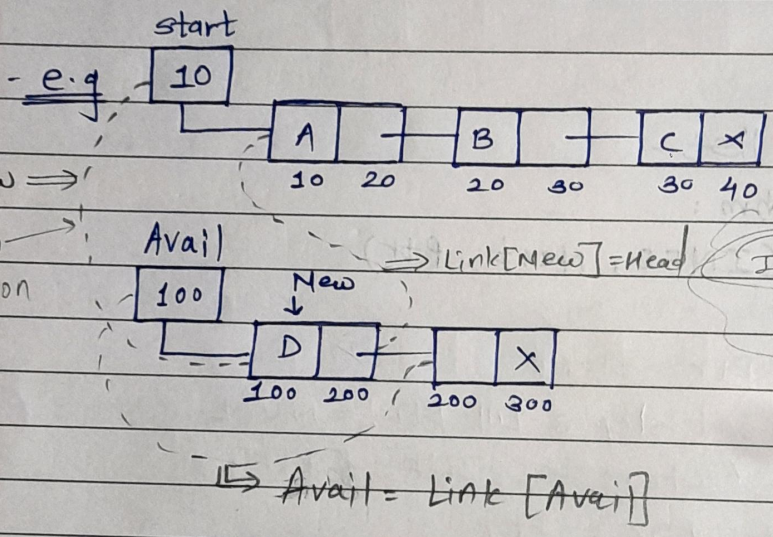
Step 1: Set Ptr = start
Step 2: Repeat step 3 till Ptr != NULL
Step 3: if (Item == INFO [Ptr]
set LOC = Ptr
else
Ptr = Link [Ptr]
step 4: Exit

21-10-23

Page No.
Date

3. Insertion:

- 'Insertion' refers to the operation of adding new element in the existing linked list.
- Insertion of new element in the list can be done in 3 ways :- (1) In the begginning of list
  (2) At the given pos"
  (3) At the last position / insertion in sorted list



- To insert the new element into the existing LL, following 3 things should be done:
  1) Allocating the new node for new element from the available list.
  2) Assigning data to new element.
  3) Adjust the pointers.

26-10-23

Page No.
Date

Algorithm:

Insert_beg (Info, New, Item, Link, Avail, start)

Step 1: if start = NULL then exit
Step 2: if Avail = NULL then overflow occurs and exit
Step 3: Set Avail = New
~~Step 4~~: Set Avail = Link [Avail]
Step 4: Set Info [New] = Item
Step 5: Link [New] = Head
Step 6: Head = New
Step 7: Exit

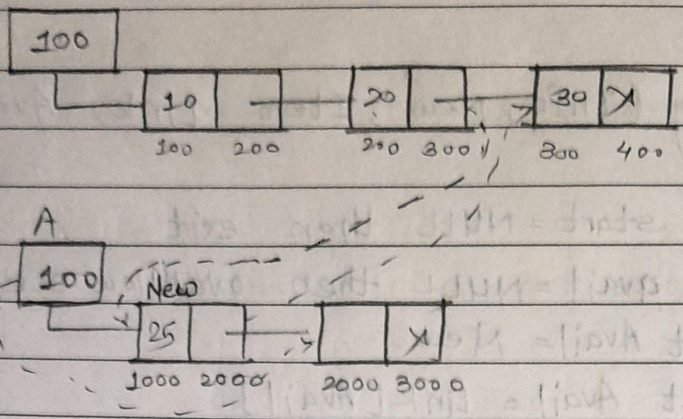
Pseudocode:

```
Insert_beg (struct node * Head, int data)

    {
        struct node * new
        new = (struct node) malloc (sizeOf (struct node))
        new -> data = Item
        new -> Next = Head
        Head = new
    }
    Return Head
```

27-10-23

Page No.
Date

Q.



Existing item = 20 ; New item = 25

Algo

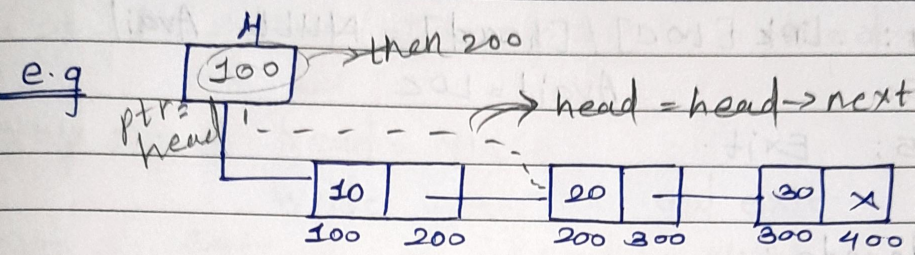
Insert_LOC (Head, Link, Info)

step 1 : if Avail = NULL
write overflow and exit
step 2: set New = Avail and
Avail = Link[Avail]
Step 3: Set Info[New] = Item
Step 4: if LOC = NULL then
set Link [New] = Head and
Head = New
else
Link [New] = Link[LOC]
Link [LOC] = New
Step 5: Exit.

4. Deletion:

- Deletion refers to the operation of removing a node from the linked list.
- Deletion can be done in 3 ways :-
  (i) Delete a node from the 1st posⁿ.
  (ii) Delete a node from the given locⁿ.
  (iii) Delete a last node from the list.

e.g



Pseudocode :

```
del_1st (struct node* head)
    {
      struct node* ptr;
      if (head == NULL)
        {
          print("List is empty");
        }
      else
        {
          ptr = head;          // 100
          head = head -> next; // = 200
          free(ptr);           // 100
        }
      return head;
```

Link(head) = Avail
100 = Avail
Avail = LOC
when avail

28-10-22

Page No.
Date

Algorithm:

Algo_Del_1st (Info, ptr, Link, head)

step 1: if head = NULL
        write underflow and exit.
step 2: Set ptr = head
step 3: Set head = Link[head]
step 4: Link[LOC]/[head] = ~~NULL~~ Avail
~~step 5:~~ Avail = LOC
step 5: Exit.

Pseudocode:

```
del_last (struct node * head)

{
  struct node* ~~head~~ ptr , * prev
  if (head == NULL)
     {
       print ("List is empty") }
     else if (head->next = NULL)
        {
          free(head)  -> if LL has only 1 element
          head = NULL
        }
  else {
        ptr = head
        while (ptr->next != NULL)
          {
            prev -> ptr
            ptr = ptr -> next
          }
```

```
    prev -> next = NULL
      free (ptr)
  }
   return head
}
```

* Algorithm:

Algo_Del_last (Info, ptr, link, head

Step 1 : if head = NULL
write underflow and exit.

Step 2 : ~~Set~~ if link [head] = NULL
link [head] = Avail
head = NULL

Step 3 : set ptr = head

Step 4 : Repeat step 5 until link[ptr] != NULL

Step 5 : set prev = ptr
set ptr = link [ptr]

Step 6 : set link[prev] = NULL
link [ptr] = Avail
Avail = LOC

Step 7 : Exit

02-11-23

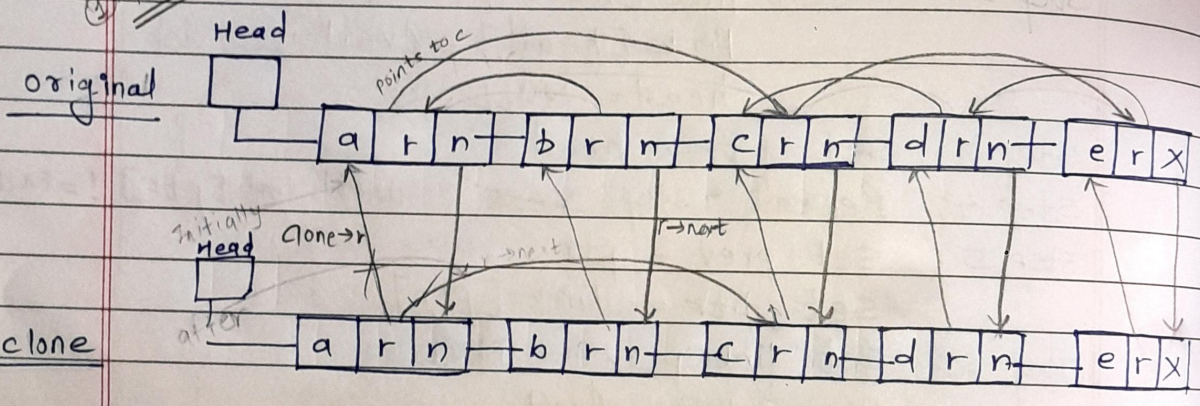
Page No.
Date

→ S.complexity: No. of random pointers → node
→ T.complexity: O(n)

* Copy / clone single linked list:

- Here, we've to deal with two pointers - one is 'next' pointer which points to the next element in the list and 'random' pointer which points to any node in the list. (cloning list)
- For cloning the LL, it takes O(n) time. The pointer points to the next node is c/a 'next pointer' & the pointer points to the random element is c/a 'arbit' pointer i.e it can point to any arbitary node in the list.

(1) For a.



clone→r = clone→r→r→next.

If we copy/ clone the list, then how to manipulate the pointers. Assign the addr to the random pointer of copied node. Initially random pointer of copied node doesn't any addr; bcoz we haven't assign any addr to random pointer.
To assign the addr we've to follow some steps:-

i) The next pointer of original list will point to the next node of copied list. o.arn → c.arn

ii) Now, we'll assign address for all random pointers of copied linked list by providing the addr of node available in the original LL, i.e random pointer of a of clone list holds the addr of node a of original list.
Random pointer of b of clone list holds the addr of node b of original list.

o.arn → c.arn

iii) Now, to provide the link to random pointer, we use the stmt. ⇒ clone→r = clone→r→r→next

For each node in copied LL will hold the addr of

for e.g ① Consider, node a from copied LL which hold the address of node a in original LL.

o.arn → c.arn

a→r = a→r→r→next (n)

a → points to; r → points to next of clone

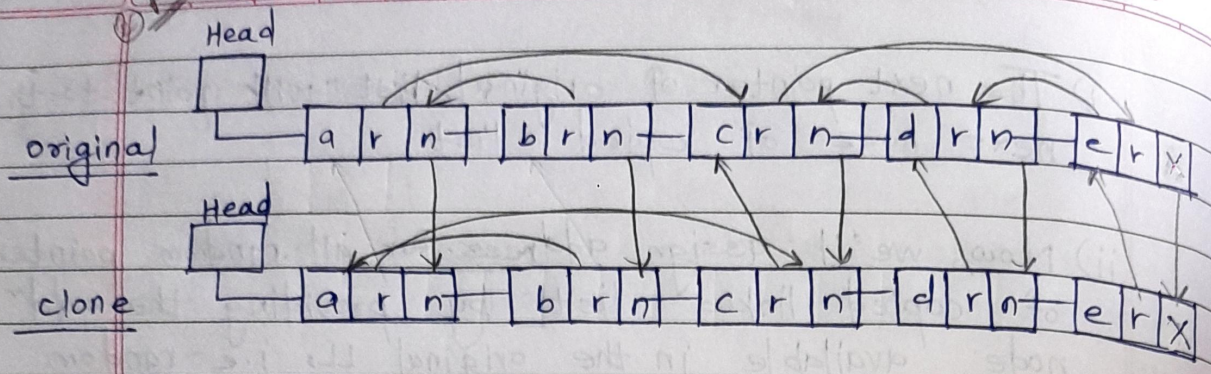
Random pointer of node a in original LL hold the addr of c, then the next of c hold the addr of node c from clone LL. ∴ clone of a hold addr of clone of c

② Consider, node b from copied LL which hold the addr of node b in original LL.

b→r = b→r→a→next

Points to next of a ⇒ ∴ Node b of copied LL point to a of copied LL

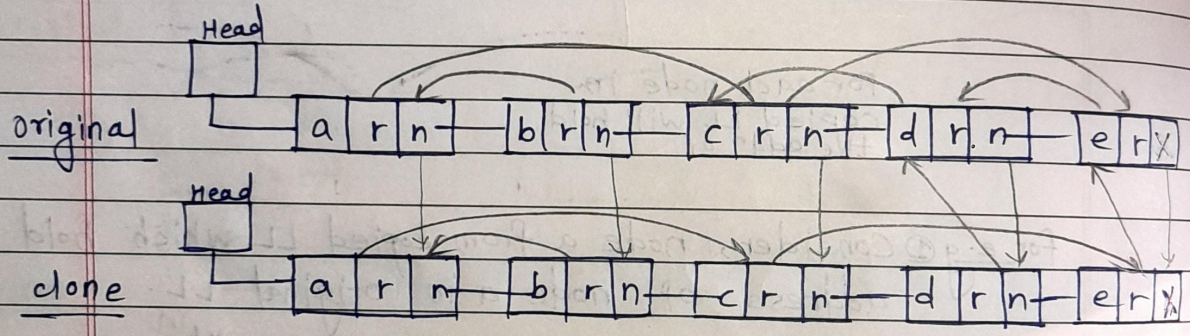
Random pointer of node b in original lined list holds the addr of a, then the next of a hold the addr of node a from clone LL.



③ Consider, node c from copied LL which hold the address of node c in original LL.

$$c \rightarrow r = c \rightarrow r \rightarrow e \rightarrow next$$

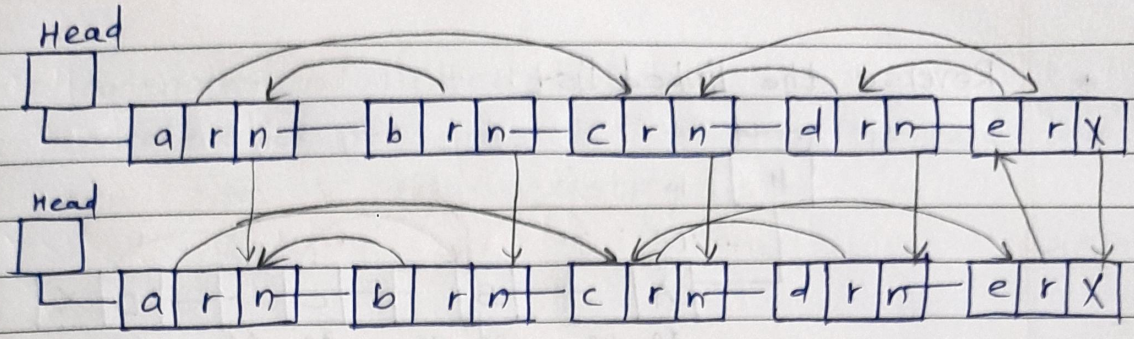
Random pointer of node c in original LL hold the address of e, then the next of e hold the address of node e from clone LL.



④ Consider, node d from copied LL which hold the address of node d in original LL.

$$d \rightarrow r = d \rightarrow r \rightarrow c \rightarrow next$$

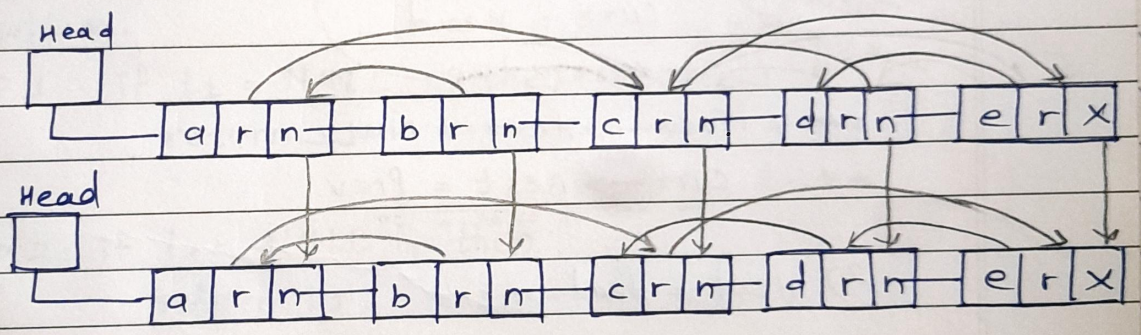
Random pointer of node d in original LL hold the addr of c, then the next of c hold the address of node c from clone LL.



⑤ Consider, node e from copied LL which hold the address of node e in original LL.

$e \rightarrow r = e \rightarrow r \rightarrow d \rightarrow next$

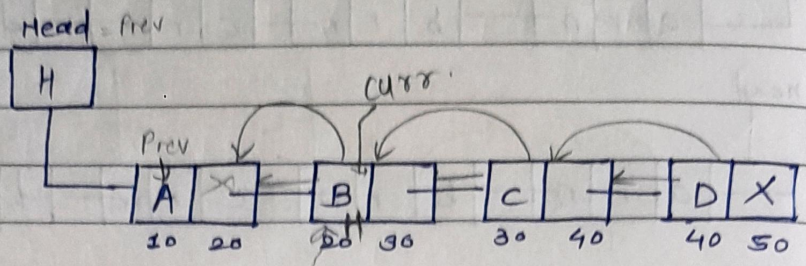
Random pointer of node e in original LL hold the address of d, then the next of d hold the address of node d from clone LL.



3-11-23

Page No.
Date

* Reverse the linked list:



1) Set Prev = Head

Prev = 10

Head = Head → next

∴ H = 10 → next

H = 20

This will work together

curr = Head

∴ curr = 20

** A → Prev → next = NULL

** curr → next = Prev

20 → n => 30 = 10

2) Head = Head → next

= 20 → next

∴ H = 30

Prev = curr

10 = 20

curr = Head → ∴ curr = 30

40 20

curr → next = prev

Head = Head → next

30 → next

40

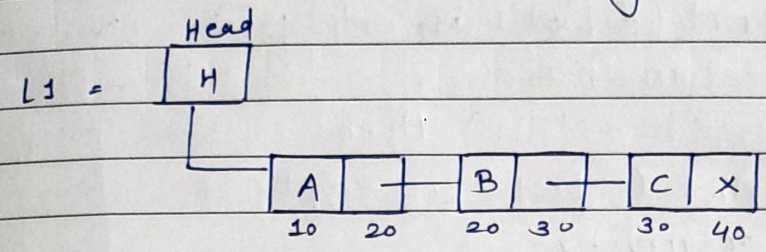
∴ Head = 40

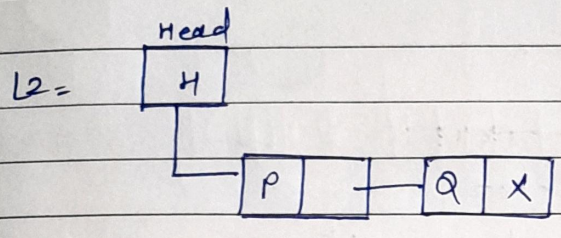
04-11-23

Page No.
Date

* Concatenation of linked list:

- "Concatenation of sigle LL" means append the new list to the existing list.





* Algorithm:

Step 1 : if L1 = NULL , then
return L2

Step 2 : if L2 = NULL , then
return L1 .

Step 3 : Set ptr = head (L1)

Step 4 : do traversing in list L1 by using
APPLY PROCESS TO Info [ptr]

Step 5 : Set ptr = head (L2)

* Pseudo code:

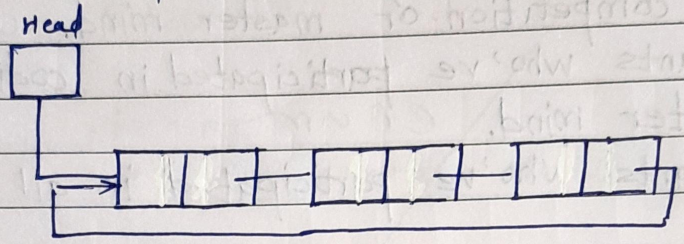
```
struct node * concat (node *ptr, *h1, *h2)
        if (h1 = NULL) then
        print ("List 1 is empty")
        return h2
      elseif (h2 = NULL) then
      print ("List 2 is empty")
       return h1.

       ptr = h1
      while (ptr -> next != NULL)
        {
            ptr = ptr -> next
        }
      ptr -> next = h2
    }
    return h1;
}
```

Page No.
Date

* Types of linked list: → single / simple / linear LL
  → ~~Circular LL~~
  → Doubly LL
  → Header LL

1. Circular linked list:

- In a single list, we can move from starting node to any other node in one direction only; but in circular linked list, the link field of last node always point to the head node.
- Here, null pointer is not present.



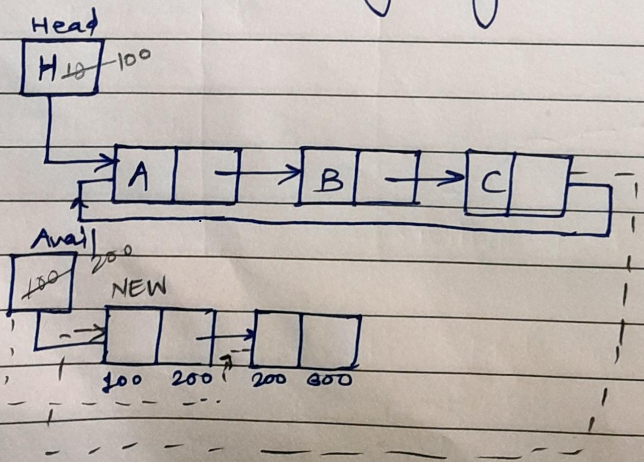
* Operations of circular LL :-

① Traversing: Terminating condⁿ (until): ptr → next != start
(same as single LL)

② Searching:
(same as single LL)

③ Insertion:

i) Insertion in the beginning



Page No.
Date

Algorithm :

Insert_beg

step 1 : ~~Set~~ if start = NULL then exit

step 2 : if avail = NULL then overflow occurs and exit.

step 3 : Set avail = NEW

set avail = link[avai[]

step 4: Set link[New] = start

start = NEW

step 5: link[Last] = NEW