Date .....

Unit-2

Dynamic Memory Allocation

Memory is allocated at two ~~times~~ times:
(1) Compile Time
(2) Run Time

Memory which is allocated to the program at compile time is called Static Memory Allocation (SMA).

Memory which is allocated to the program at run time is called Dynamic Memory Allocation (DMA).

Memory is not allocated at the compile time but rather memory to be allocated is decided at compile time. Memory actually is allocated at run time.

Ex:-
```
void main()
{
    int a, b;        → 8 bytes decided at compile time
    printf("\n Enter value of a and b");
    scanf("%d %d", &a, &b);
    printf("%d %d \t \t", a, b);
}
void main()
{
    int a[10];       → 40 bytes decided
    int n;
    printf("\n Enter size of array");
    scanf("%d", &n);  → 3 → 28 bytes wasted
```

Spiral

Date .....

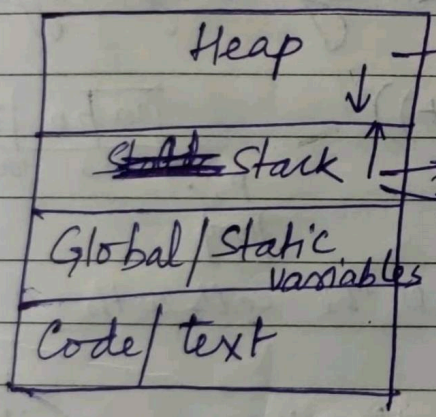
Using static memory allocation, memory is wasted and cannot be reused.

To remove the drawbacks of static memory allocation, we use dynamic memory allocation where memory can be allocated and freed at run time according to program needs.

Dynamic memory allocation functions :-

1. malloc
2. calloc
3. realloc
4. free

Memory Organisation

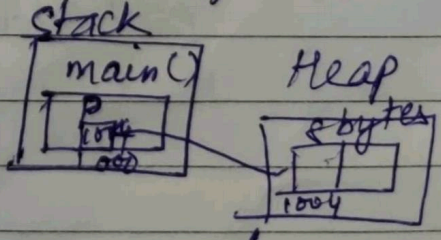


→ For Dynamic Memory Allocation

→ For static memory allocation
→ It will take limited size. Local var & func's will take memory from stack.

* For dynamic memory allocation we will require pointers.

```
void main()
{
  int *p;
```



Always free memory allocated at heap once freed

Date .....

Dynamic Memory Allocation using malloc()

↓

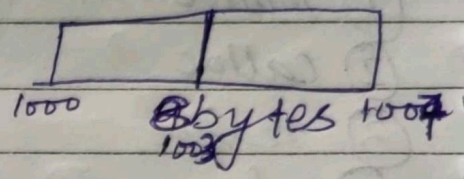
to allocate the memory for structures, linked list.

Syntax: void *malloc(size_t size) → bytes

↓ unsigned value

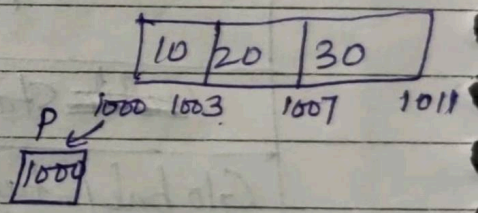
It will return base address of memory block.

malloc returns void pointer because it does not have idea about what type of value will be stored in memory block.



For ex if we want to allocate 12 bytes of memory dynamically in heap area for storing integer type of value

int *ptr = (int*) malloc(3 * sizeof(int))



By default malloc will initialise all the cells with garbage value.

If malloc is not able to allocate memory it will return null else it will return base address of first block on success.

Date .....

Program

```
#include<stdio.h>  #include<stdlib.h>
int main()
{
 int n, *ptr;
printf("\n Enter total no of values");
scanf("%d",&n);
 ptr=(int*)malloc(n*sizeof(int));
  printf("\n Enter values");
  for(int i=0; i<n; i++)
    {
      scanf("%d", ptr+i);
    }
  printf("\n Entered values are");
   for(int i=0; i<n; i++)
    {
      printf("%d", *(ptr+i));
    }
    free(ptr);
}
```

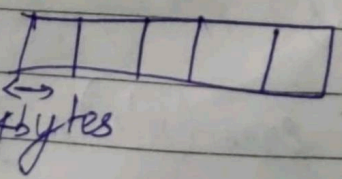
Dynamic memory Allocation using calloc() → contiguous allocation

It is used to allocate memory dynamically for multiple blocks and each block is of same size.

Syntax :- (void*) calloc(size_t n, size_t sizeof each block)

n → no. of blocks

Ex:- calloc(5, sizeof(int))

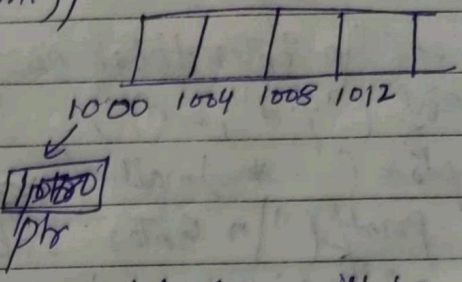


Spiral

Date .....

On success, it will return base address of first memory block and on failure it will return NULL.

```
int *ptr = (int *) calloc(4, sizeof(int))
```



Initially using calloc, all the memory blocks will be initialized with 0.

Program

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
int n, *ptr;
printf("\n Enter the total no. of elements");
    scanf("%d", &n);
ptr = (int *) calloc(n, sizeof(int));
  for(int i = 0; i<n ; i++)
      {
      scanf("%d", (ptr+i));
      }
printf("\n Entered values are");
    for(int i = 0; i<n; i++)
        {
        printf("%d", *(ptr+i));
        }
    // free(ptr); }
```

Spiral

Date .....

Dynamic memory Allocation using realloc()

Using realloc() we can increase/decrease the memory block size.

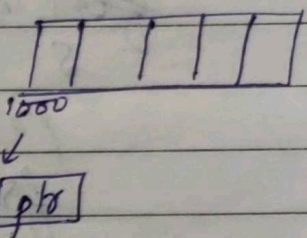
Syntax :- (void*)realloc(void *ptr, size_t size)

↓ new size

It will resize the memory block without loseng the previous content.

Ex :- int *ptr;

ptr = (int*) malloc(sizeof(int))



Now we want to store 6 elements

ptr = (int*) realloc(ptr, 6*sizeof(int));

Program

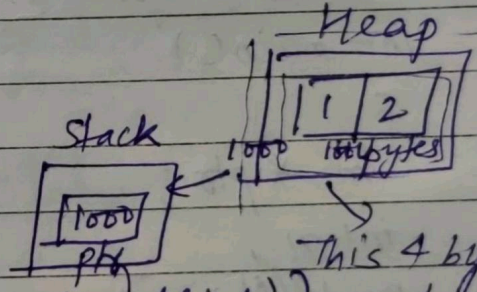
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
 int *ptr;
 int n;
 printf("\n Enter total no. of values");
 scanf("%d", &n);
 printf("\n Enter values");
 for(int i=0; i<n; i++)
 {
  scanf("%d",
ptr = (int*) calloc(n, sizeof(int));
printf("\n Enter values");
for(int i=0; i<n; i++)
```

```
{
scanf("%d", &(ptr+i));
}
int *ptr1; printf("\n Enter updated value of n"); scanf("%d", &n);
ptr1 = (int*) realloc(ptr, n*sizeof(int))
printf("previous address is %d", ptr);
printf("new address is %d", ptr1);
printf("Updated values are");
for(int i=0; i<n; i++)
{
printf("%d\t", *(ptr1+i)); }
free(ptr);
}
```

Deallocating Dynamically allocated memory using free()

Using free() we can dynamically release or free the previously allocated memory.

Syntax: ~~void~~ free(pointer name);




```
Ex: int *ptr;
    ptr = (int*) malloc(2*sizeof(int));
    free(ptr);
    ptr = NULL;
```

Once you have used free function you cannot access the elements by dereferencing.
It means that ptr is holding invalid memory location which is no more existing and may be it is now being referenced by some another pointer. Hence ptr acts as dangling pointer.

Date .....

Program

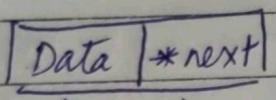
```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int *ptr;
int n;
printf("\n Enter total no of values");
scanf("%d", &n);
ptr = (int *) malloc(n* sizeof(int));
printf("\n Enter elements");
for(int i=0; i<n; i++)
{
scanf("%d", (ptr+i));
}
printf("\n Entered values are");
for(int i=0; i<n; i++)
{
printf("%d\t", *(ptr+i));
}
free(ptr);
printf("\n Entered values are");
for(int i=0; i<n; i++)
{
printf("%d\t", *(ptr+i));
}
}
```

It may show some undefined behaviour because memory has been freed.

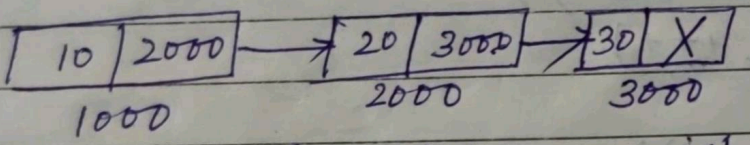
Date .....

Linked List

A linked list is a linear data structure in which the elements are not stored in contiguous memory locations.
The elements are stored in a linear fashion and they are allocated memory dynamically.
Each node of linked list consists of two parts:-



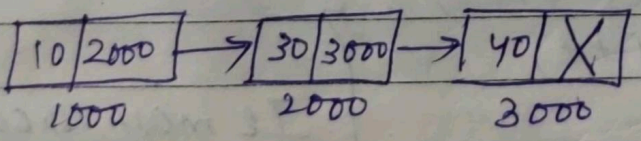
data pointer to next node



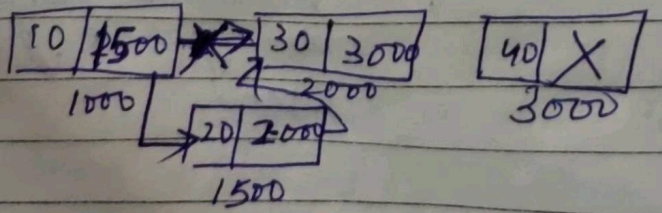
Each node of linked list consist of data field and reference link to the next node in memory.

In linked list, insertions and deletions are much faster as compared to arrays but searching will take time as we have to search sequentially.

For ex:



To insert: 20 in between 10 and 30



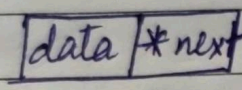
Date .....

Types of linked list

1. Single Linked List
2. Double Linked List
3. Circular linked list

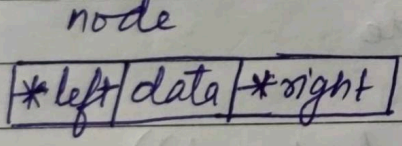
(1) Single linked list

In single linked list, each node will have two fields:



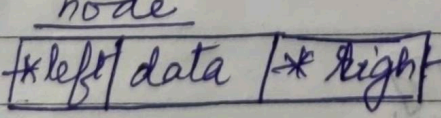
(2) Double Linked list

In double linked list, each node will have three fields:



(3) Circular Linked List

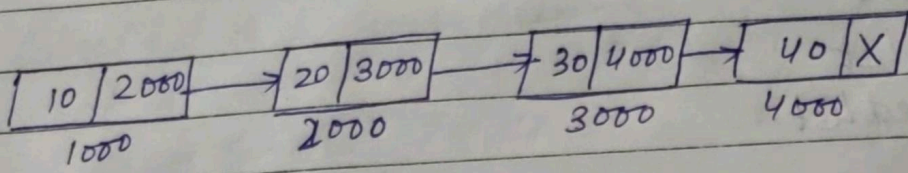
In the circular linked list, each node will have three fields:



Difference between double linked list and circular linked list is that in circular linked list, last node is connected back to first node.

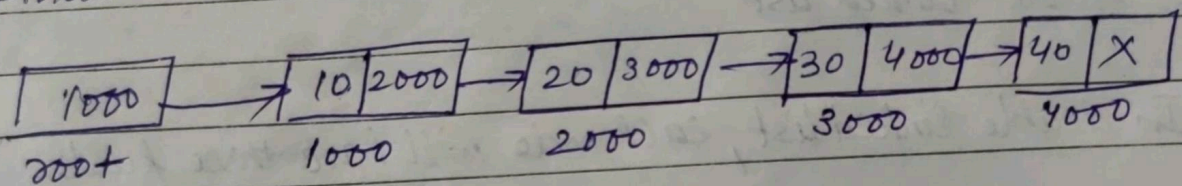
Date .....

Singly linked list creation of node




```
struct node
{
int data;
struct node * next;
};
struct node * root = NULL;
```

→ creation of node



To allocate memory to node

```
root = (struct node *)malloc(sizeof(struct node));
```

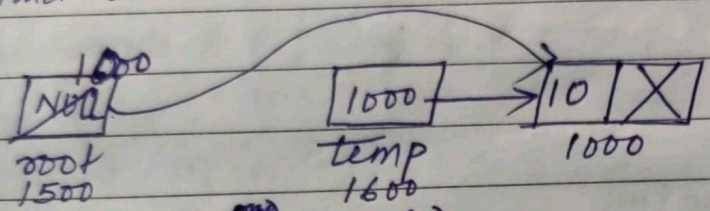
Single linked list operations

1. Insertion at the end
2. Insertion at the beginning
3. Insertion at specified position
4. Deletion at the end
5. Deletion from end
6. Deletion from specified position
7. Find the length of link list
8. Display
9. Reverse linked list

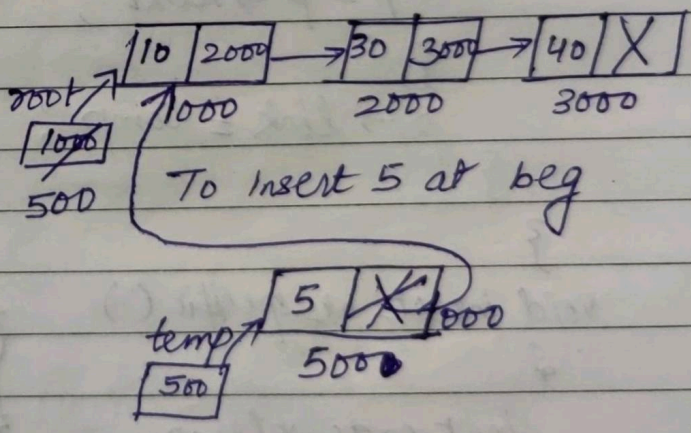
Date .....

① Insertion of node

```
struct node
{
 int data;
struct node *next;
 };
struct node * root = NULL;
```

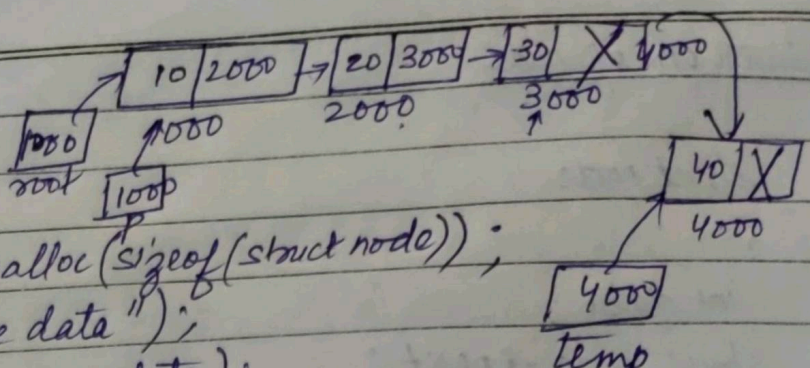



```
void insert_at_beg()
{
 struct node *temp;
temp = (struct node *) malloc(sizeof(struct node));
 printf("\n Enter node data");
 scanf("%d", temp -> data);
 temp -> next = NULL;
   if (root == NULL)
   {
   root = temp;
   }
  else
  {
 temp -> next = root;
 root = temp;
  }
```

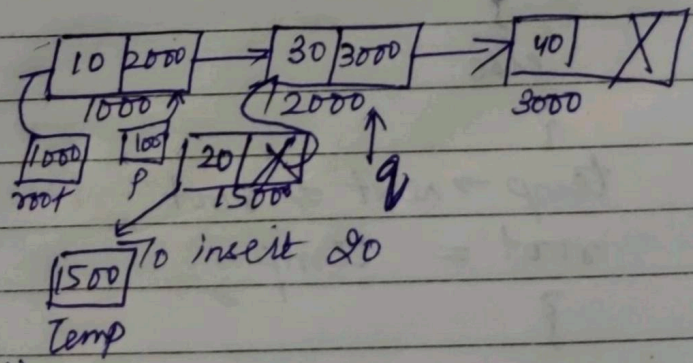


Date .....

```
void insert_at_end()
{
 struct node * temp;
temp = (struct node *) malloc (sizeof (struct node));
 printf ("\n Enter node data");
  scanf ("%d", &temp -> data);
  temp -> next = NULL;
  if (root == = NULL)
    {
    root = temp;
    }
  else
  {
   struct node *p;
     p = root;
    while (p -> next != NULL)
    {
      p = p -> next;
    }
    p -> link = temp;
  }
}
```

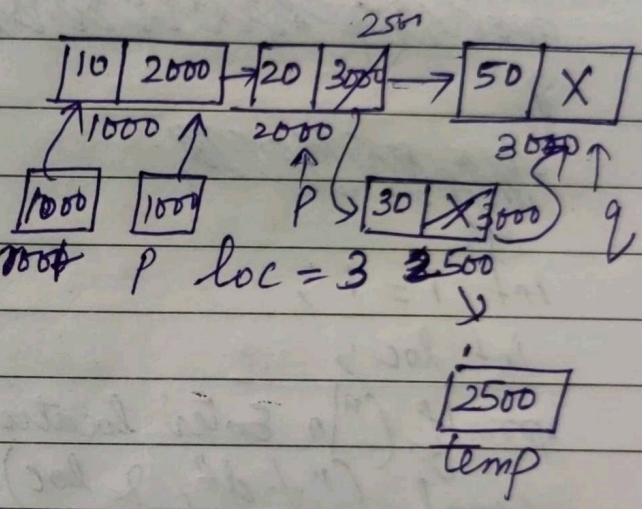



```
void insert_at_specific()
{
  struct node *temp;
    int i=1, loc;
temp = (struct node *)
    malloc (size of (struct node));
  printf ("\n Enter node data");
scanf ("%d", &temp -> data);
 temp -> next = NULL;
```



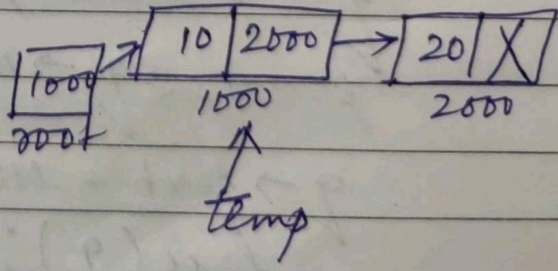
Spiral

```
printf("\n Enter location where you want to insert");
scanf("%d", &loc);
while(i<loc) else if(root==NULL)
{
temp root = temp;
}
else
{
struct node *p *q
p = root;
while(i<loc-1)
{
p = p->next;
i++;
} q = p->next;
q = temp->next
p->next= temp;
}
}
```



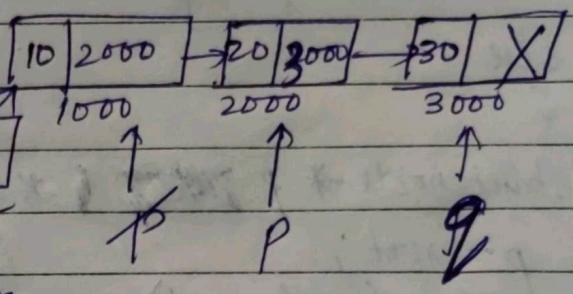
Deletion of node in linked list

```
void delete_from_beg()
{
if(root==NULL)
{
printf("\n No node to delete");
}
else
{ struct node *temp;
```



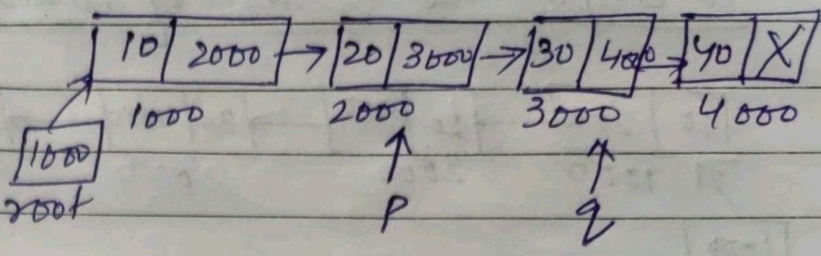
Date .....

```
temp = root;
root = temp->next; temp->next = NULL;
free(temp);
}
}
```



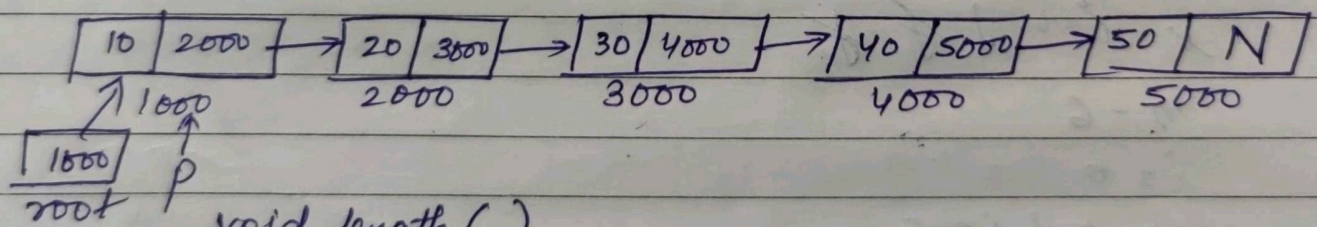

```
void delete_from_end()
{
  struct node *p, *q;
  p = root;
  [illegible]
  [illegible];
  int i = 1;
  int loc;
  printf("\n Enter location");
  scanf("%d", &loc);
  while(i < loc - 1)
  {
    p = p->next;
  }
  q = p->next;
  p->next = q->next;
  q->next = NULL;
  free(q);
  }
}
```

```
void delete_from_specific()
{
struct node *p, *q;
   p = root;
  int i = 1;
  int loc;
printf("\n Enter location");
 scanf("%d", &loc);
  while(i<loc-1)
   {
    p = p->next;
   }
    q = p->next;
p->next = q->next;
   q->next = NULL;
    free(q);
   }
```



To delete from loc: 3

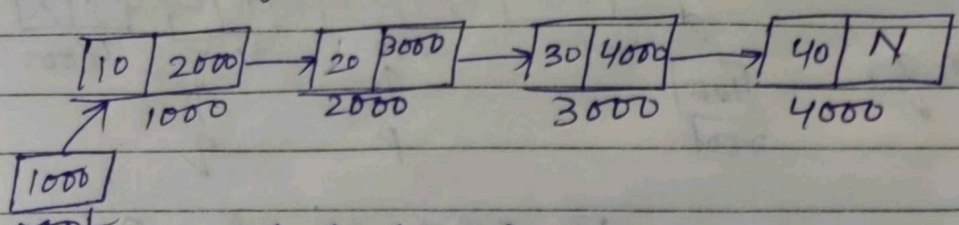
To find length of linked list




```
void length()
{
 struct node *p; int count = 0;
  p = root;
  while(p != NULL)
   { count++;
    p = p->next;
   }
 printf("\n Total no of nodes are %d", count);
}
```

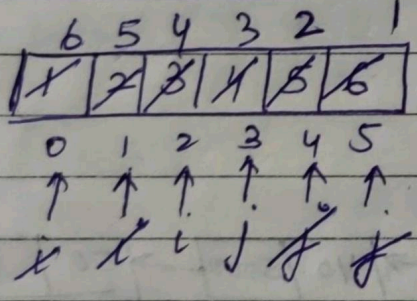
Date .....

To display the elements of linked list




```
void display()
{
struct node *p;
   p = root        printf("\n Elements of linked list are");
while(p != NULL)
{
printf("%d\t", p->data);
   p = p->next;
}
```

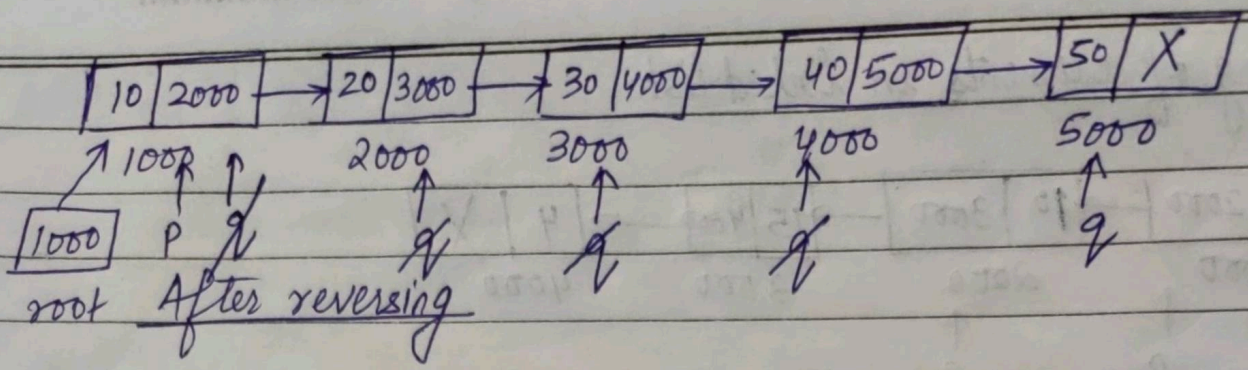
Reverse Linked list

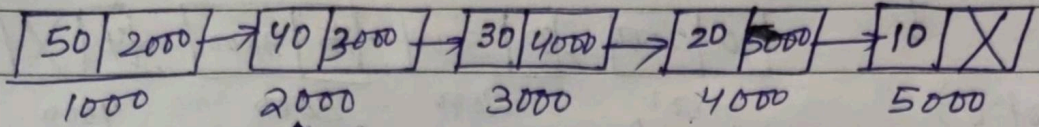



```
n = 6
i = 0
j = n-1  i.e. 5
while(i<j) {
temp = a[i]
a[i] = a[j]
a[j] = temp;
   i++;
   j--;
}
```

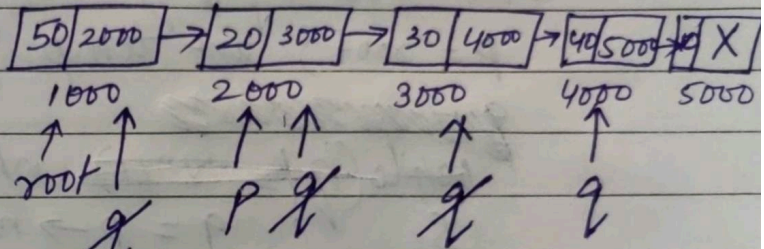
Spiral

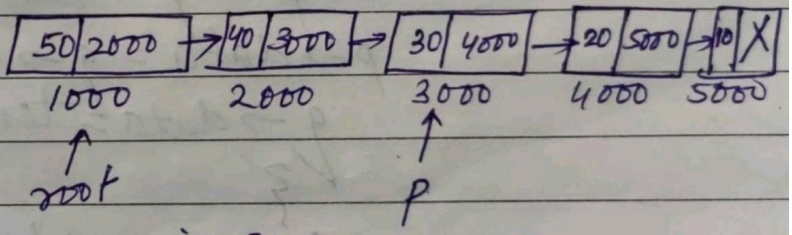
Date .....






```
void reverse()
{
struct node *p, *q;
int i, j, k; int temp;
 i=0;
j = len() - 1;
 p = root
 q = root
 while (i<j)
 { k=0;
 while (k<j)
 {
   q = q -> next;
    k++;
 }
temp = p -> data;
p -> data = q -> data;
 q -> data = temp;
   i++;
   j--;
p = p -> next;
q = root;
 }
```

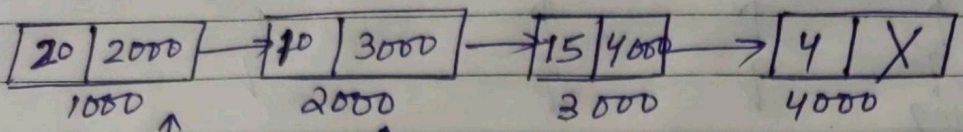


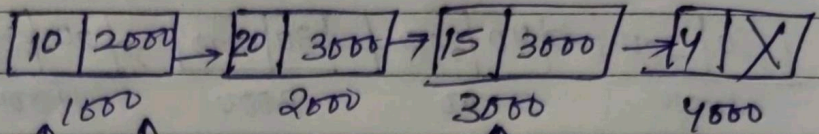


Spiral

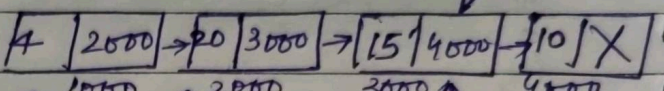
Date .....

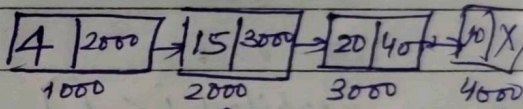
Sorting of elements in linked list

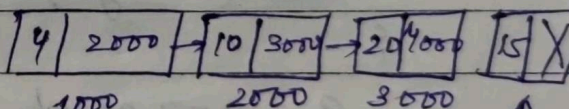


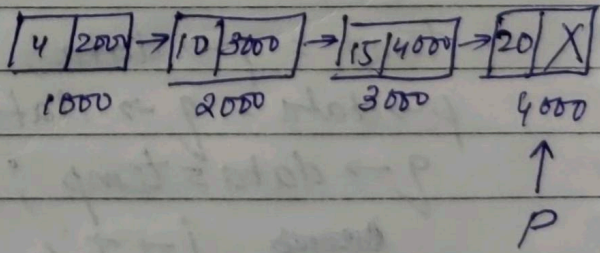



```
void sort()
{
  struct node *p,*q
  p = root; int temp ;
  while(p->next!=NULL)
  {
    q = p->next;
    if(p->data > q->data)
    {
      temp = p->data;
      p->data = q->data;
      q->data = temp;
    }
    q = q->next ;
  }
  p = p->next ;
}
```



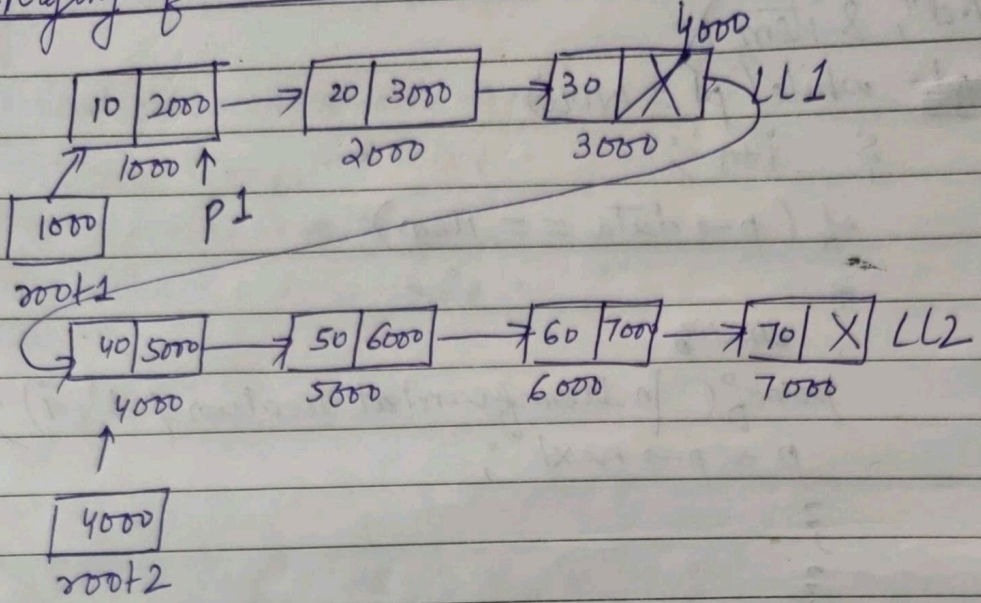






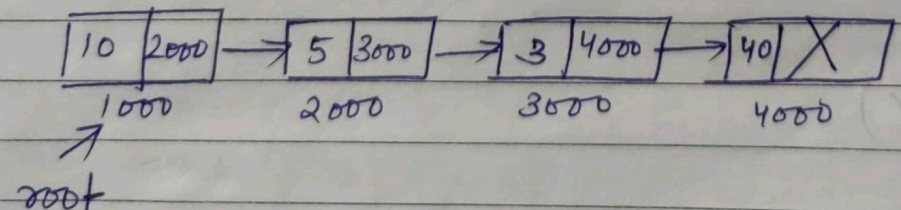
Date .....

Merging of nodes in linked list




```
void merge ()
{
  struct node *p1;
  p1 = root1;
  while(p1->next != NULL)
  {
    p1 = p1->next; }
  p1->next = root2;
}
```

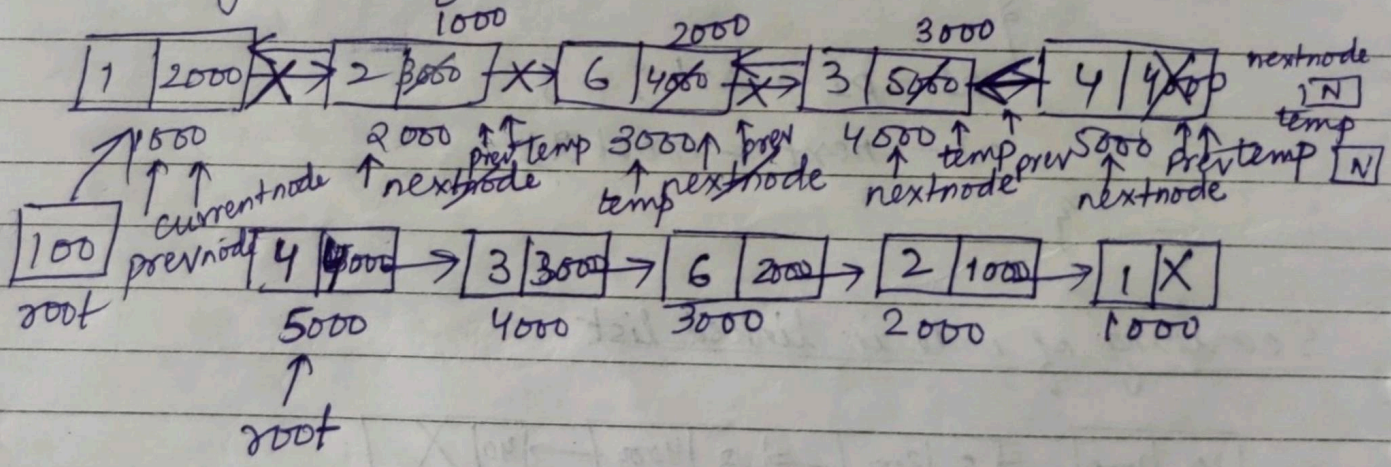
Searching of node in linked list




```
void search
{
  struct node *p; int item;
  p = root; int i = 0;
```

```
printf("\n Enter item you want to search");
scanf("%d", &item);
while(p != NULL)
{ i++;
if(p->data == item)
{
printf("\n Item found at location %d", i); }
p = p->next;
}
}
printf("\n Item not found");
}
```

Reversing nodes of linked list

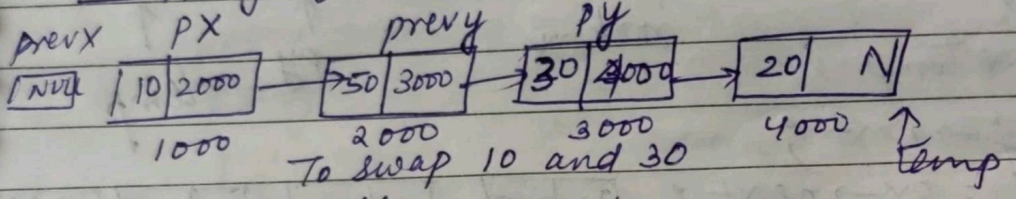



```
void reverse()
{
Struct node *temp;
Temp = root;
Struct node * prevnode = NULL;
Struct node * nextnode;
nextnode = root;
```

Spiral

```
while(nextnode != NULL)
{
  nextnode = nextnode -> next;
  temp -> next = prevnode;
  prevnode = temp;
  temp = nextnode;
  }
  root = prevnode;
  }
```

Swapping of elements in linked list



There are three cases :-

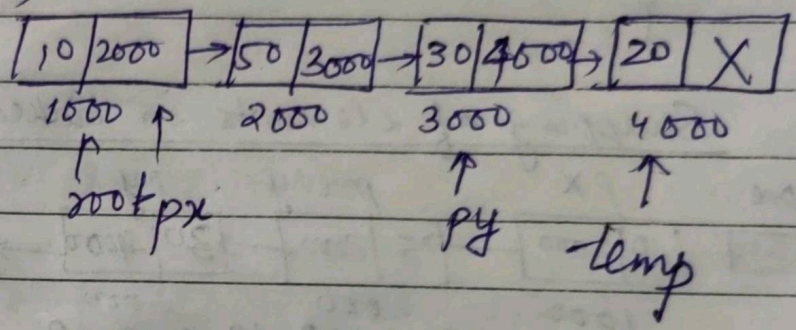
1. Either one of the node is root node.
2. Either one of the node is last node.
3. Both are internal nodes

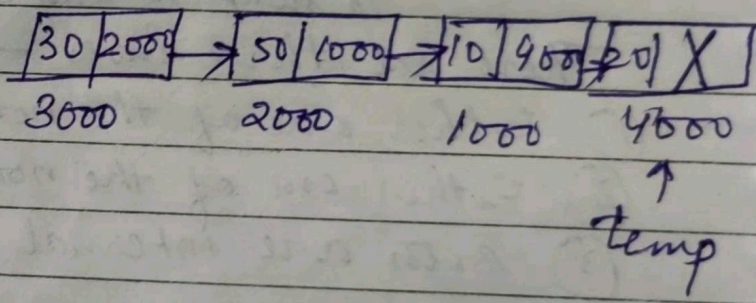
First we have to search elements which need to be swapped.

```
void search()
{
struct node *p,*prev;
 p = root;
 prev = NULL;
while(p != NULL && p->data != item)
 {
    prev = p;
    p = p->next;
 }
  struct node *px = p;
  struct node *prevx = prev;
```

Date .....

```
p = root;
prev = NULL;
while(p != NULL && p->data != item2)
{
    prev = p;
    p = p->next;
}
struct node *py = p;
struct node *prevy = prev;
}
void swap()
{
struct node *temp;
temp = py->next;
py->next = px->next;
px->next = temp;
```



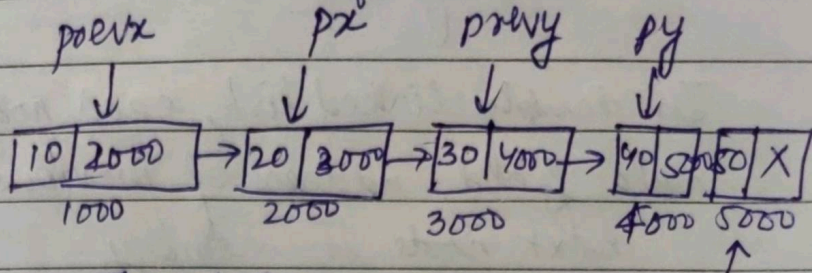


To find root node

```
if(prevx == NULL)
{
    py = root;
prevy->next = px;
}
if(prevy == NULL)
{
    px = root;
prevx->next = py;
}
```

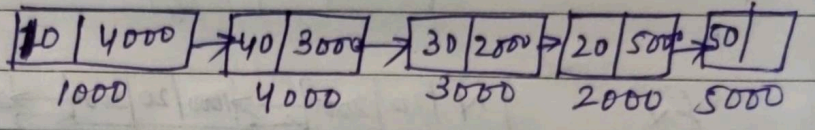
Date .....

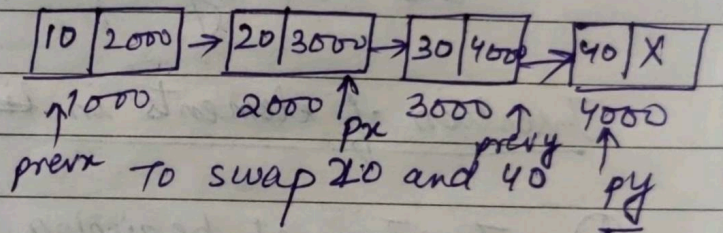
If both nodes are internal node or one of the node is last node.



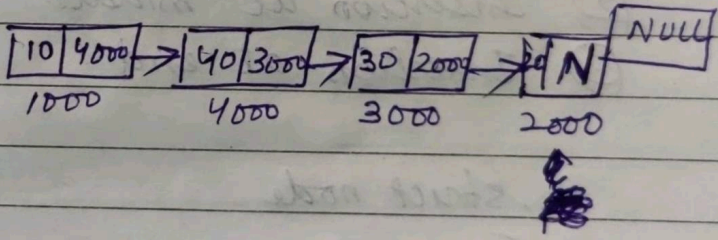

```
void swap( )
{
struct node *temp;
temp = py->next;
py->next = px->next;
px->next = temp;
 if (prevx != NULL && prevy != NULL)
  {
  prevx->next = py;
  prevy->next = px;
  }
```

To swap 20 and 40





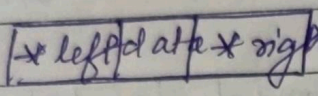
To swap 20 and 40



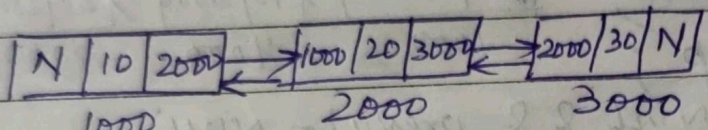
Date .....

Doubly Linked list

In doubly linked list, each node is having three fields i.e data field, address of previous node and address of next node in memory.



Ex:



The first node left pointer will be NULL and the last node right pointer will be NULL.

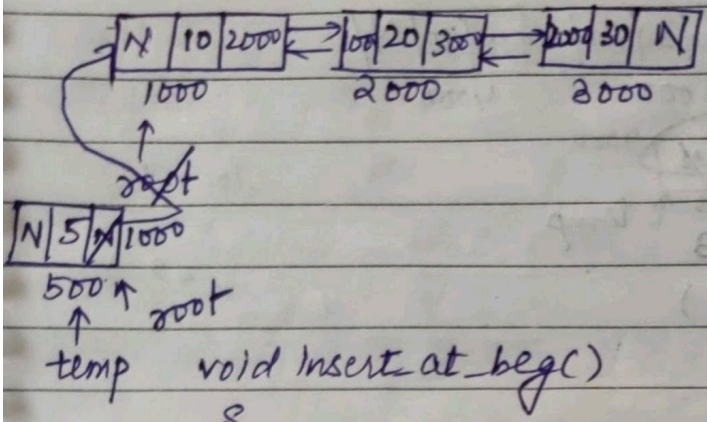
Insertion of elements in ~~link~~ doubly linked list

1. Insertion at beginning
2. Insertion at middle
3. Insertion at end

```
struct node
{
 int data;
struct node * left;
struct node * right;
};
struct node * root;
```

Date .....

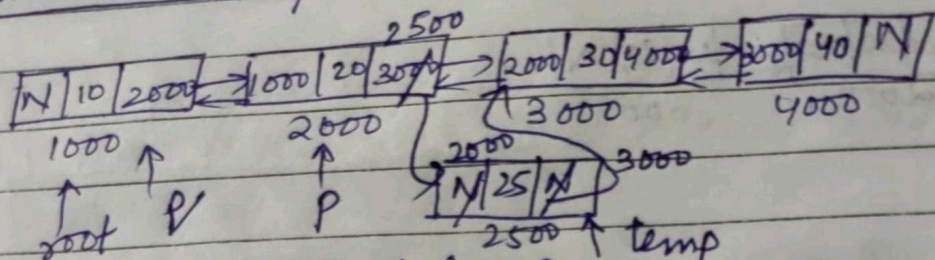
(1) Insertion at beginning




```
void Insert_at_beg()
{
struct node * temp;
temp = (struct node *) malloc (sizeof(struct node));
printf ("\n Enter node data");
scanf ("%d", &temp -> data);
temp -> left = NULL;
temp -> right = NULL;
  if (root == NULL)
    {
     root = temp;
    }
  else
  {
   temp -> right = root
   root -> left = temp;
   root = temp;
  }
```

Date .....

② Insertion at specific position



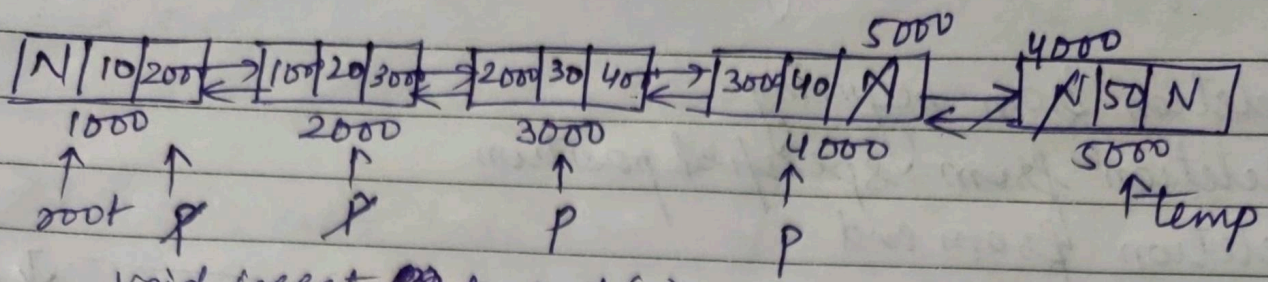
To insert 25 at loc 3

loc = 4
1 < 3
2 < 3

```
void insert_at_specific()
{
  int i=1;
  int loc;
  struct node *temp, *p; p=root;
temp = (struct node *)malloc(sizeof(struct node));
printf("\n Enter node data");
scanf("%d", &temp->data);
temp->left = NULL;
temp->right = NULL; printf("\n Enter location");
if (root == NULL)   scanf("%d", &loc);
{
  root = temp;
}
else
{
  while(i<loc-1)
  {
    p = p->right;
  }
temp->right = p->right;
temp->left = p;
p->right = temp;
}
}
```

Spiral

Insertion at end

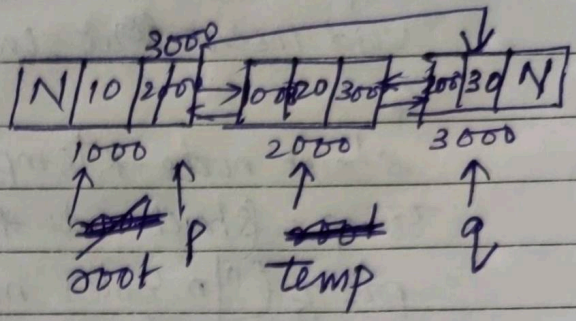



```
void insert_at_end()
{
struct node *temp,*p;
temp=(struct node *)malloc(sizeof(struct node));
printf("\n Enter node data");
scanf("%d",&temp->data);
temp->left=NULL;
temp->right=NULL;
if(root==NULL)
  {
   root=temp;
  }
else
{ p=root;
  while(p->right!=NULL)
   {
    p=p->right;
   }
p->right=temp;
temp->left=p;
}
}
```

Date .....

Deletion of node in doubly linked list

1. Deletion from beginning
2. Deletion from specified position
3. Deletion from end



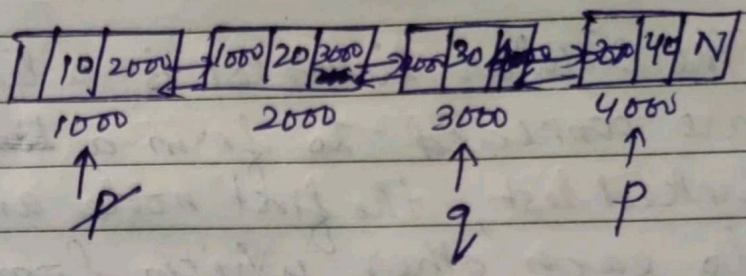

```
void delete_from_beg()
{
 struct node * temp ;
 temp = root ;
 root = root -> right ;
 root -> left = NULL;
 free(temp);
 }
```

```
void delete_from_specific()
{
struct node * temp ,* q;
struct node * p;
  p = root ;
  int i = 1;
  int loc;
  printf("\n Enter location");
   scanf("%d", & loc);
   while (i < loc - 1)
    {
      p = p -> next ;
       i++;
    }
   temp = p -> right;
   q = temp -> right;
   p -> right = temp -> right ;
   q -> left = p;
   temp -> left = NULL;
```

Spiral

Date .....

```
temp -> right = NULL;
free(temp);
}
```




```
void delete_from_end()
{
struct node *p, *q;
p = root;
while(p -> right -> right != NULL)
{
p = p -> right;
}
q = p -> left;
q -> right = NULL;
p -> left = NULL;
free(p);
}
```

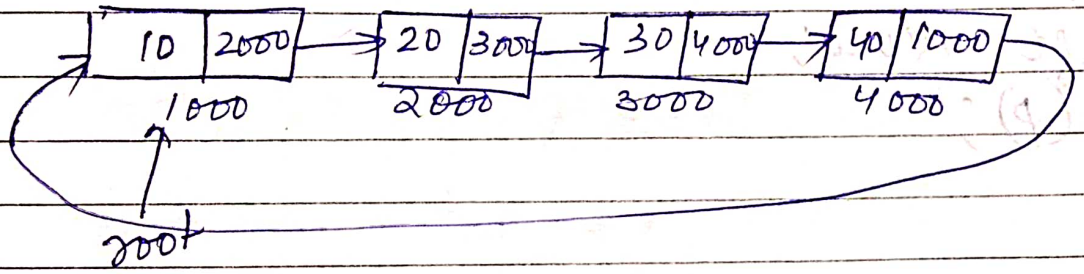
Date .....

Circular linked list

The circular linked list is a linked list where all nodes are connected to form a ~~link~~ circle. In a circular linked list, the first node and last node are connected to each other which forms a circle. There is no NULL at the end.

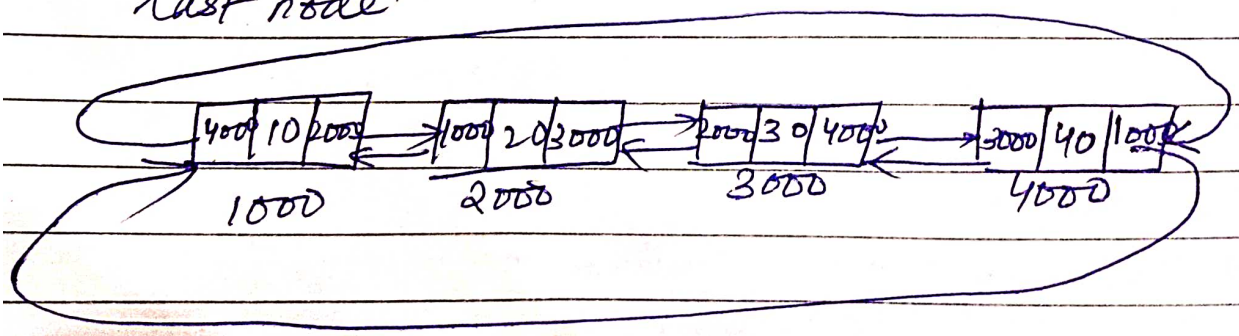
(1) Singly Circular linked list

In singly circular linked list, the last node of the list contains pointer to the first node of the list. There is no NULL at the end.



(2) Doubly Circular linked list

Doubly circular linked list has properties of both doubly linked list and circular linked list and the last node right pointer points to the first node of the list and the first node left pointer points to the last node.



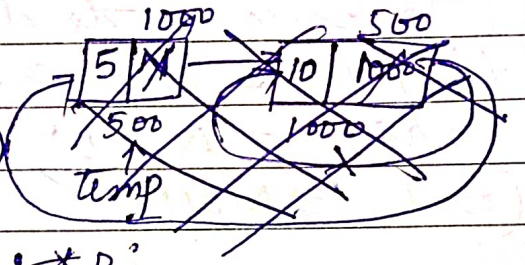
Singly Circular Linked List

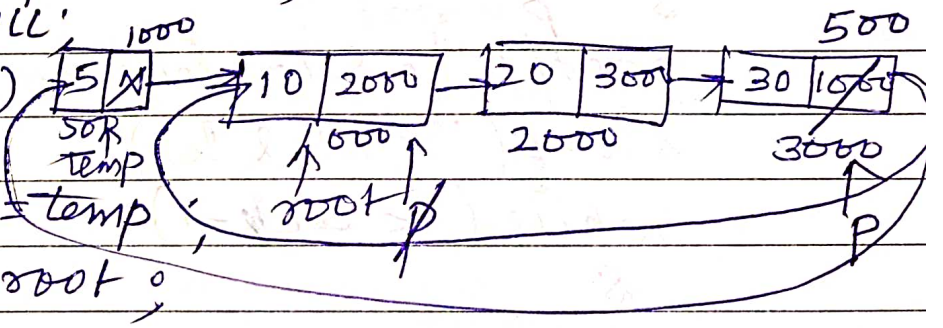
Date .....

1. Insertion at beginning
2. Insertion at specific position
3. Insertion at end

(1) Insertion at beginning

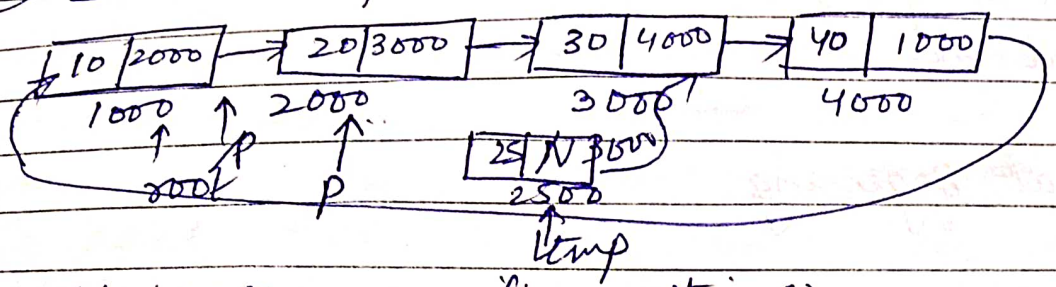
```
struct node
{
 int data;
 struct node *next;
};
struct node *root;
void insert_at_beg()
{
 struct node *temp, *p;
temp = (struct node *) malloc(sizeof(struct node));
printf("\n Enter node data");
scanf("%d", &temp->data);
temp->next = NULL;
if (root == NULL)
{
 root = temp;
 root->next = root;
}
else
{
 p = root;
 while (p->next != root)
 {
  p = p->next;
 }
 temp->next = root;
 p->next = temp;
 root = temp;
}
```





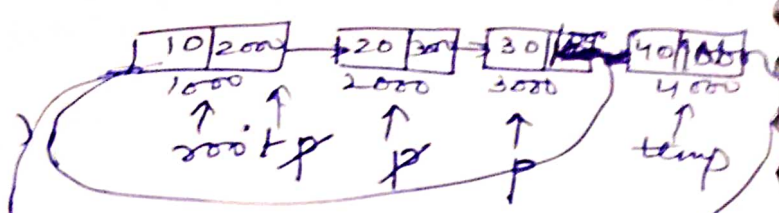
Date .....

(2) Insertion at specified position



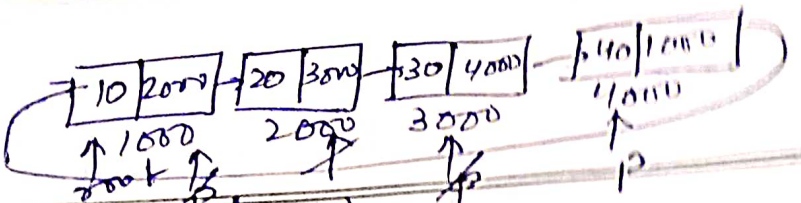

```
void insert_at_specific_position()
{
  struct node *p, *temp; int loc, i=1;
  temp = (struct node *) malloc(sizeof(struct node));
  printf("\n Enter node data");
  scanf("%d", &temp->data);
  temp->next = NULL;
  if (root == NULL)
  {
    root = temp; root->next = root;
  }
  else
  {  p = root;
  printf("\n Enter location");
  scanf("%d", &loc);
  while (i < loc-1)
  {
    p = p->next;
  }
  temp->next = p->next;
  p->next = temp;
}
```

(3) Insertion at end




```
void insert_at_end()
{
  struct node *temp, *p;
temp =(struct node *) malloc(sizeof(struct node));
 printf("\n Enter node data");
 scanf("%d", &temp->data);
 temp->next = NULL;
 if (root == = NULL)
  {
   root = temp;
  root = root->next;
   }
  else
  {
  p = root;
  while (p->next != root)
   {
    p = p->next;
  }
 p->next = temp;
temp->next = root;

}
```

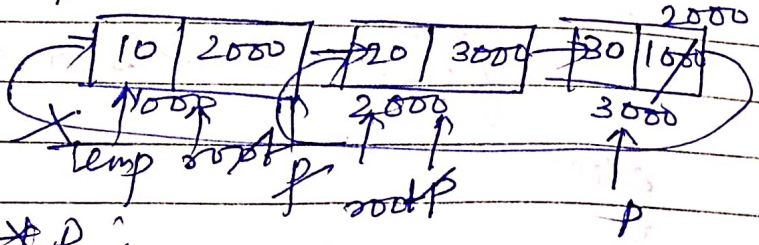



```
void display()
{
struct node *p;
 if (root == NULL)
 {
  printf("\n No nodes to display");
 }
 else
 {
 p = root;
 while(p->next != root)
 {
  printf("%d->", p->data);
  p = p->next;
 }
 printf("%d->", p->data);
 }
```

Date .....

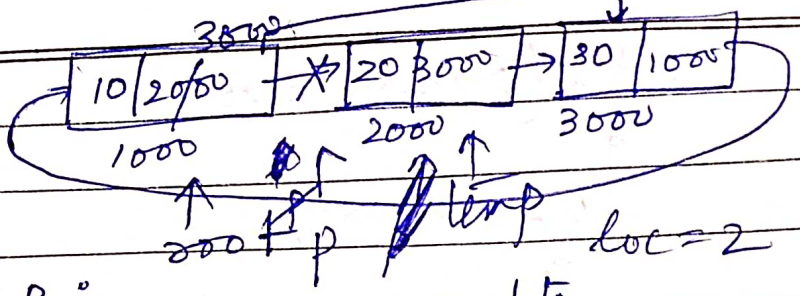
Deletion of node in singly circular linked list

1. Deletion from beginning
2. Deletion from specific position
3. Deletion from end



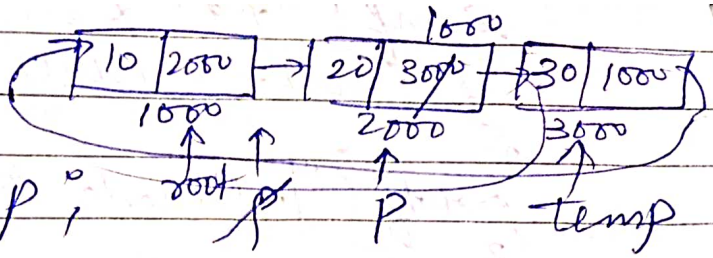

```
void del_from_beg()
{
  struct node *temp, *p;
  if (root == NULL)
  {
    printf("\n No nodes to delete");
  }
  else
  {
    p = root;
    while (p->next != root)
    {
      p = p->next;
    }
    temp = root;
    root = root->next;
    p->next = root;
    temp->next = NULL;
    free(temp);
  }
}
```

Date .....



(2) void del_from_specific()

```
{
  int i, loc; i = 1;
  struct node *temp, *p;
  printf("\n Enter location from where you want to delete");
  scanf("%d", &loc);
  if (root == NULL)
  {
  printf("\n No nodes to delete");
  }
  else
  {  p = root;
  while(i< loc-1)
  {
   p= p->next;
  }
  temp = p->next;
  p->next = temp->next;
  temp->next = NULL;
  free(temp);
  }
}
```

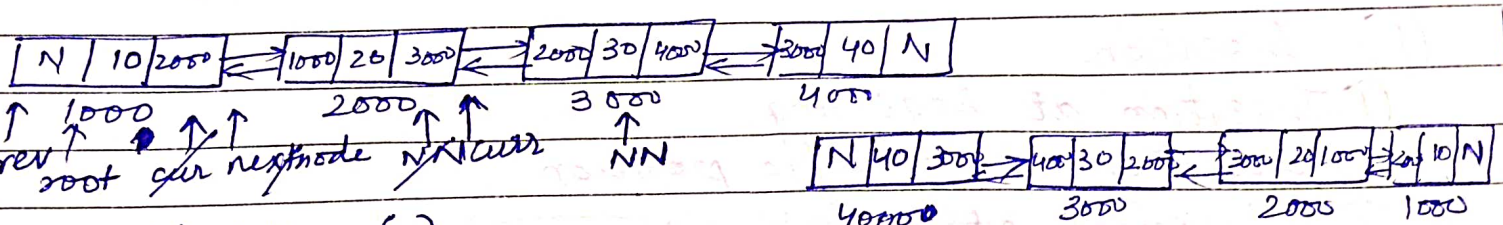



```
void del_from_end()
{
  struct node *temp, *p;
  int i;
  i = 1;
    if (root == NULL)
    {
    printf ("\n No nodes to delete");
    }
    else
    {
    p = root;
    while (i< len()-1)
     {
       p = p-> next ;
     }
  temp = p->next;
  p->next = temp ->next;
temp->next = NULL;
    free(temp);
}
```

Spiral

Date .....

Reversing a ~~circular~~ doubly linked list

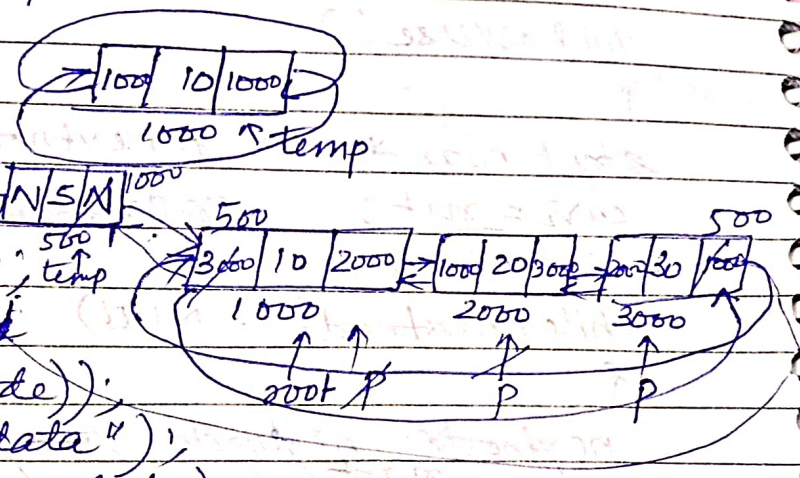



```
void reverse()
{
struct node *curr, *nextnode, *prev;
curr = root;                    prev=NULL;
nextnode=root;
while(nextnode != NULL)
{
 nextnode = nextnode->right;
curr->right = prev;             curr->left = curr->right
 prev=curr;  curr=nextnode;
 } nextnode = nextnode->right;
curr->right = prev;
curr->left = Nextnode;
 prev = curr;
  curr = nextnode;
 }
root = prev;
}
```

Date .....

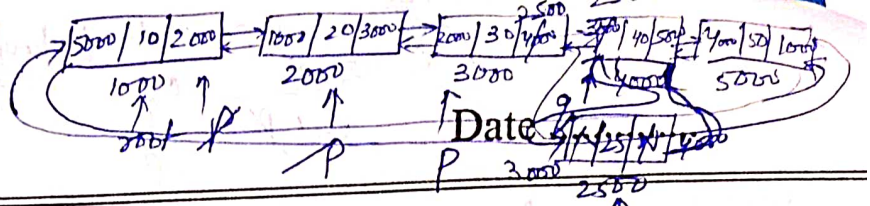
Circular Doubly Linked List

(1) Insertion
(1) Insertion at beginning
(2) Insertion at specific position
(3) Insertion at end




```
void insert_at_beg()
{
  struct node *temp, *p;
  temp = (struct node *)malloc
        (sizeof(struct node));
  printf("\n Enter node data");
  scanf("%d", &temp->data);
  temp->left = NULL;
  temp->right = NULL;
  if (root == NULL)
    {
      root = temp;
      root->right = root;
      root->left = root;
    }
  else
  {
    p = root;
    while (p->right != root)
    {
      p = p->right;
    }
  temp->right = root;
  p->right = temp;
  temp->left = p;
  root->left = temp;
  root = temp;
  }
}
```

Spiral

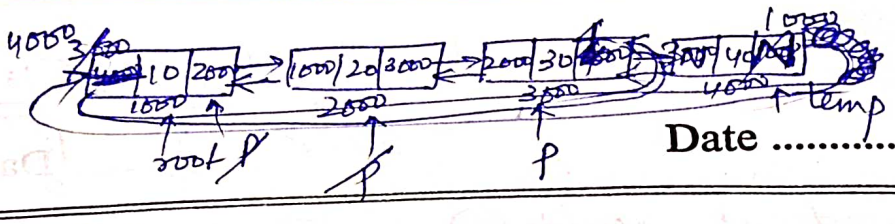



```
void insert_at_specific()
{
struct node *p, *temp, *q;
int i, loc;
i = 1;
temp = (struct node *) malloc (sizeof (struct node));
printf ("\n Enter node data ");
scanf ("%d", &temp -> data);
temp -> right = NULL;
temp -> left = NULL;
printf ("\n Enter location");
scanf ("%d", &loc);
if (root == NULL)
{
 root = temp;
 root -> right = root;
 root -> left = root;
}
else
{ p = root;
while (i < loc - 1)
{
 p = p -> right;
 i++;
}
q = p -> right;
temp -> right = q;
q -> left = temp;
p -> right = temp;
temp -> left = p;
}
}
```

loc = 4

temp

Spiral

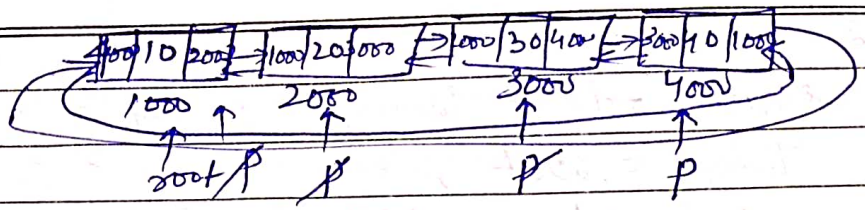


Date .....

```
void insert_at_end()
{
struct node *temp, *p;
temp = (struct node *)malloc(sizeof(struct node));
printf("\n Enter node data");
scanf("%d", &temp->data);
temp->left = NULL;
temp->right = NULL;
if (root == NULL)
{
root = temp;
root->right = root;
root->left = root;
}
else
{
p = root;
while (p->right != root)
{
p = p->right;
}
temp->right = root;  root->left = temp;
temp->left = p;
p->right = temp;
}
}
```

Date .....

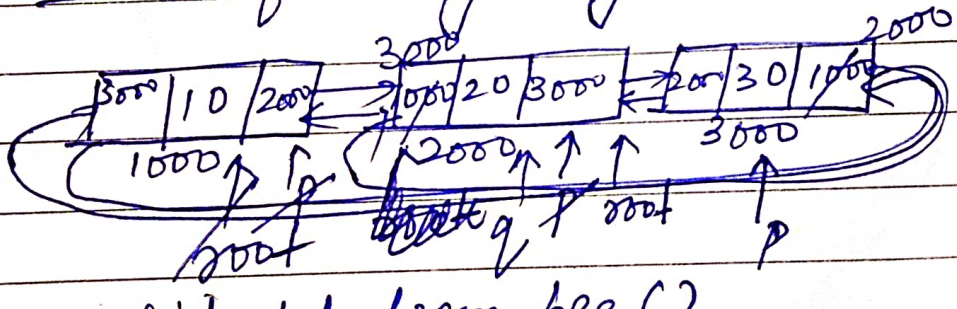
```
void display()
{
struct node *p;
p = root;
if (root == NULL)
{
printf("\n No nodes to display");
}
else
{
p = root;
while(p->next != root)
{
printf("%d->", p->data);
p = p->next;
}
printf("%d->", p->data);
}
```



Deletion in Doubly Circular linked list

1. Deletion from beginning
2. Deletion from specified position
3. Deletion from end.

1. Deletion from beginning



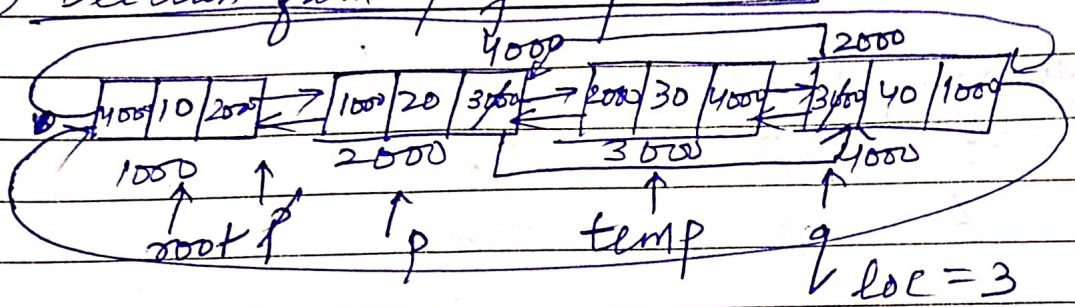

```
void del_from_beg()
```

Spiral

Date .....

```
{
struct node *temp, *p, *q;
temp = root;
if (root == NULL)
{
printf("\n No nodes to delete");
}
else
{
p = root;
while (p->right != root)
{
p = p->right;
}
q = temp->right;
p->right = q;
q->left = p;
root = root->right;
temp->right = NULL;
temp->left = NULL;
free(temp);
}
}
```

(2) Deletion from specified position

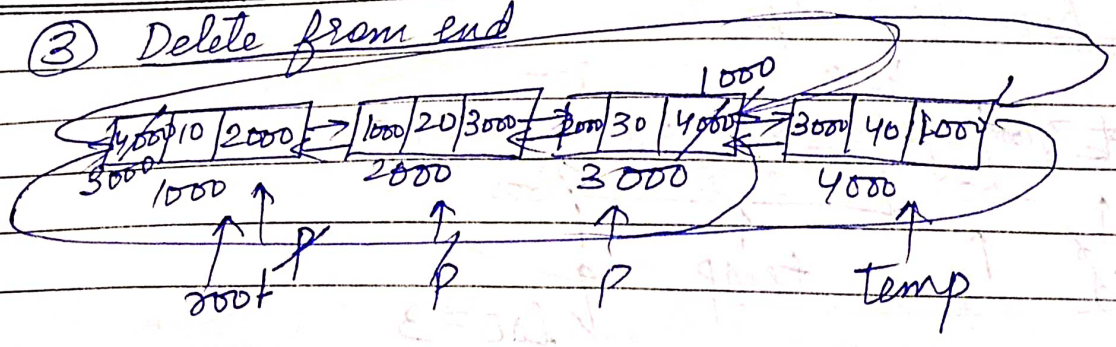



```
void del_from_middle()
{
struct node *root, *p, *q;
int i, loc;
i=1;
printf("\n Enter location from where you want to delete");
scanf("%d", &loc);
if (root==NULL)
{
printf("\n No nodes to delete");
}
else
{
p=root;
while(i<loc-1)
{
p=p->right;
}
temp=p->right;
q=temp->right;
p->right=q;
q->left=p;
temp->right=NULL;
temp->left=NULL;
free(temp); } }
```

Spread

Date .....

③ Delete from end

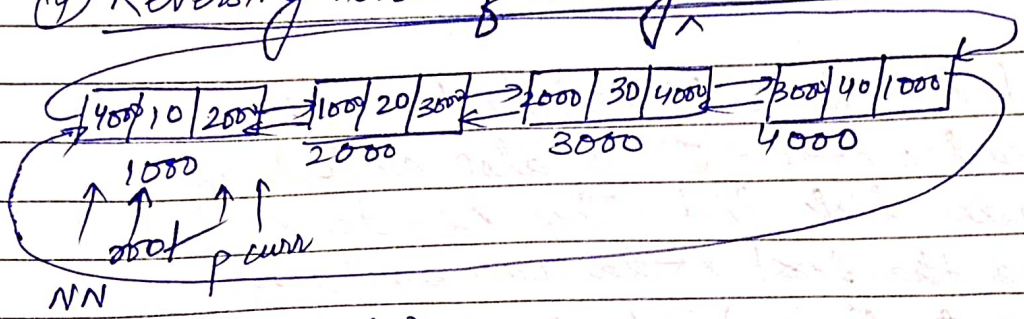



```
void del_from_end()
{
struct node *temp, *p; int i; i=1;
if (root == NULL)
{
printf("\n No nodes to delete");
}
else
{
p=root;
while (i< len()-1)
{
p=p->right;
}
temp = p->right;
p->right = temp->right;
root->left = p;
temp->right = NULL;
temp->left = NULL;
free(temp);
}
}
```

Date .....

circular

(4) Reversing nodes of doubly linked list




```
void reverse ()
{
struct node *p, *prev, *curr, *nextnode;
prev = NULL;
p = root;
curr = root;
nextnode = root;
while (p->right != root)
{
  p = p->right;
}
prev = p;
while (nextnode->right != NULL)
{
  nextnode = nextnode->right;
  curr->right = prev;
curr->left = nextnode;
  prev = curr;
  curr = nextnode;
}
nextnode = nextnode->right;
curr->right = prev;
curr->left = nextnode;
  prev = curr;
curr = nextnode;
root = prev; }
```

ode

Speraf

Date .....

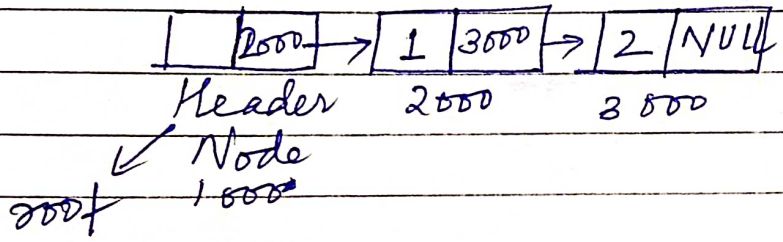
Header linked list

A header linked list is a linked list which contains a header node. It is also called as Grounded header link list. This header node is added before the first node and contains information about linked list such as count of elements etc.

There are two types of header linked list:

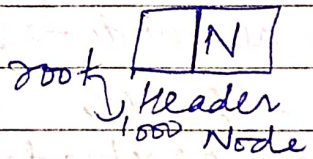
(1) Grounded Header list

The last node next field always contains NULL.



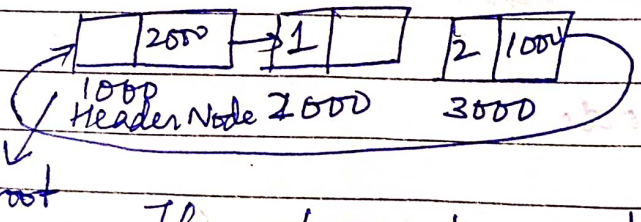
Here root will point to header node.

If root → next = NULL. It means linked list will be empty



(2) Circular Header list

Here the last node next field points to the header node.



If root → next = root. It means that linked

Circular header

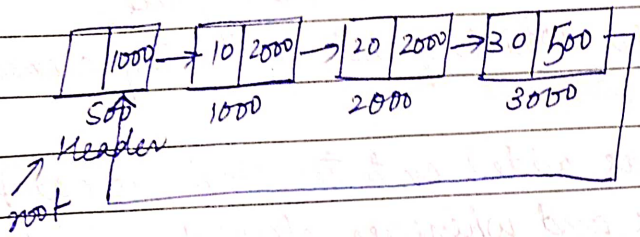
Spiral

Date .....

list will be empty

* Header list will always be circular header link list unless it is mentioned that it is grounded header link list.

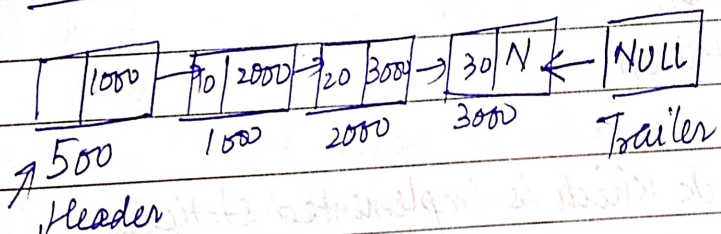
Header node always act as sentinel indicating end of list.



If header is added in front of circular linked list we are aware that how many times we have traversed the list.

Special Type of Header list

List with Header & Trailer node



Difference between Header linked list & normal linked list

(1) In normal list, root always points to first node but in header list, first node is always after header node.

(2) In normal list, location of first node is pointed by root but in header list, first node is pointed by header

Sprint