Data Structure with C

14/08/17

Advantages of Array:

1. Hold more than one value
2. Faster in execution
3. Random Access of elements

Limitations:

1. Homogeneous
2. Size can't be altered, continuous allocation
3. Bound Check
4. Memory loss within the block i.e. internal fragmentation.
5. Insertion & deletion of element takes more time
6. Deletion of element permenantly is not possible

DATA STRUCTURE

- way of organisation of data in the memory, it will also define the logical relationship b/w the data elements in a data structure.
- provides the way of accessing data elements.

* Classification of Data Structure

1. Primitive D.S.
   - int
   - char
   - float...
2. Non-primitive D.S.
   - Linear D.S
     - stack
     - Queue
     - Linkdlist
   - Non-Linear D.S
     - Tree(s)
     - Graph
   - FILE

Static v/s Dynamic Memory Allocation.

1. Static memory access faster Dynamic memory allocation
2. Memory Allocated during compile time in static
3. Memory size can't be alter in static but can do so in dynamic
4. Better memory utilization in dynamic

<alloc.h>
<stdlib.h>
<process.h>
<malloc.h>

void * malloc (Block Size)
- allocates one block at a time.
- default value - garbage

void * calloc (No. of Block, Size of each block)
- allocates multiple block
- default value - 0

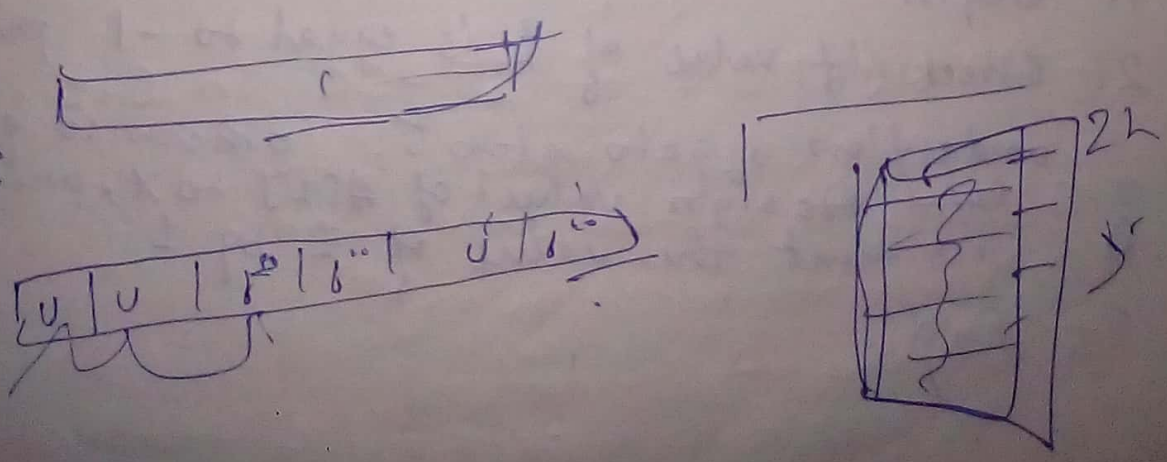
P = realloc (old size, new size)
used to alter memory allocation

free (P)
to deallocate previously allocated memory.

Q Write a C program to implement an array using dynamic memory allocation and find the sum & average of the array elements.

```
void main()
{
float/int *P = NULL, sum = 0, average = 0;
int i, n;
scanf("%d", &n);
P = (float)malloc(N * sizeof(float));
for(i=0; i<n; i++)
    scanf("%f", (p+i))

for (i= 0; i<n; i++)
    sum += *(p+i);

free(p);
average = sum/n;
}
```

Rough



C program to store a no. in an array as per the user's choice, then remove or fetch one element from that array again as per the user choice, then finally display the entire array.

OBJECTIVE: To fetch & store element from the same index.

ASSUMPTION: 'A' is an array, I = -1, size of array is N

ALGO INSERT

1. Begin
2. Check if value of I is equal to N-1 print "overflow", goto step 5. otherwise goto step 3.
3. Increment the value of I by 1
4. Assign the value of X to A[i]
5. End

ALGO DELETE

1. Begin
2. Check if value of I is equal to -1 print underflow, goto step 5, otherwise goto step 3.
3. ~~Print~~ Assign value of A[I] to X, print X
4. Decrement the value of I by 1
5. End

ALGO DISPLAY

1. Begin
2. Assign value of I to j
3. Check if j is greater than 0, repeat step 3 to 5, otherwise goto step 6
4. Print A[j]
5. Decrement the value of j by 1
6. End

FIRST DATA STRUCTURE (STACK)

- Non-Primitive D.S.
- LIFO concept
- One side open D.S.

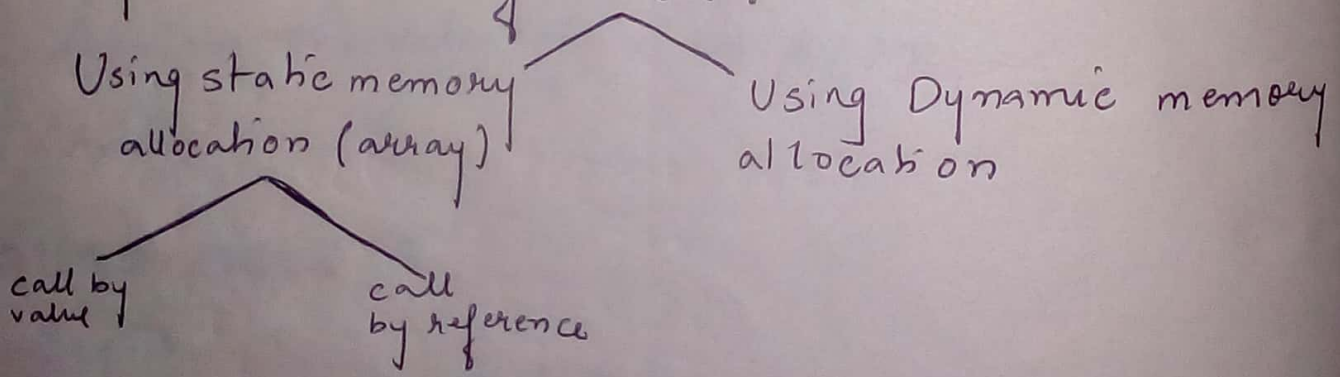
Operations of Stack

1. Push
2. Pop
3. Display
4. Peep - display the position of stack pointer
5. Empty
6. full

STACK POINTER : (TOP / TOS)

index

Implementation of Stack :



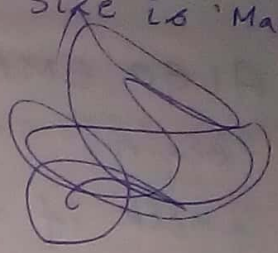
APPLICATION of STACK in C.S.

1. function Call & Recursive Call
2. Infix to postfix conversion.
3. Evaluation of postfix exp^n.
4. Reversing of string
5. As a page replacement policy in operating system.

PROBLEM STATEMENT : Implementation of stack using static memory allocation + call by value concept

OBJECTIVE : To learn the concept of stack with various opⁿ - push, pop, display, peep, empty, full.

ASSUMPTION : 'Stack' is an array, size is 'Max'; Stack pointer initialized by -1



ALGO PUSH

1. Begin
2. Check if value of TOP is equal to Max - 1, print "overflow" and goto step 5, otherwise goto step 3.
3. Increment value of TOP by 1
4. Assign the value of X to stack [TOP].
5. Return (TOP)

ALGO POP

1. Begin
2. Check if value of TOP is equal to -1, print "underflow", goto step 5, otherwise goto step 3.
3. Assign the value stack [TOP] to X, print X.
4. Decrement value of TOP by 1
5. Return (TOP)

ALGO DISPLAY

1. Begin
2. Check if value of TOP is equal to -1, print "empty", goto step 7, otherwise step 3
3. Assign the value of TOP to i
4. Check if i is greater than -1, repeat step 4-6 otherwise step 7
5. Print stack [i].
6. Decrement the value of i by 1
7. End

ALGO PEEP

1. Begin
2. Return(TOP) / Print(TOP)
3. End (optional)

ALGO EMPTY

1. Begin
2. Check if value of TOP is equal to -1, return 1, otherwise return 0

8.

```
#define max 10
int PUSH(int[], int, int);
int POP(int[], int);
void DISP(int[], int);
int PEEP(int);
int EMPTY(int);
int FULL(int);

main()
{ int stack[max], TOP=-1, x, ch;
do
{ printf(" . . . . . ");
  scanf("%d", &ch);
switch(ch)
{
   case 1:
     scanf("%d", &x);
TOP: PUSH(stack, TOP, x);
     break;
```

```
    case 2 :
TOP=POP(stack, TOP);
      break;
    case 3 :
      DISP(stack, TOP);
      break;
    case 4 :
TOP=PEEP(TOP);
      printf(" . . . . ")
      break;
    case 5 :


    case 6 :


  }while(ch!='T');

}
int PUSH(int stack[], int TOP, int x)
{   if(TOP==max-1)
        printf("overflow");

    else
    {
        TOP++;
        stack[TOP]=x;
    }
     return(TOP);
}
```

PROBLEM STATEMENT : Implementation of stack using static memory allocation and call by reference method.

OBJECTIVE : To learn the concept of stacks implementation using call by reference method

ASSUMPTION : '*Tp' which will hold the address of Top.

ALGO PUSH

1. Begin
2. Check the value at *Tp, if it is equal to Max-1, print "overflow", goto step 5, otherwise goto step 3
3. Inc. the value at *Tp by 1
4. Assign the value of x at stack[*Tp]
5. End

declaration:

```
void PUSH ( int[], int *, int );
void POP  ( int[], int* );
void DISP ( int[], int );

void main()
{ int stack[Max], x, Top=-1, ch;

        case 1:
calling:    PUSH (stack, &Top, x);
            break;
```

QUEUE

- Non-primitive linear D.S.
- FIFO concept
- Both side open D.S, ends - RABR & FRONT

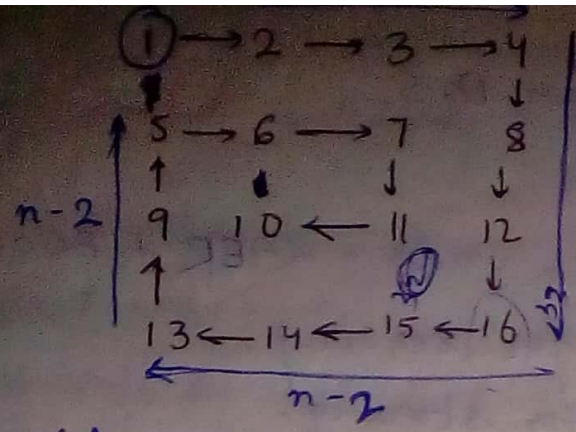
FRONT End is used for serving.
RABR End is used for storing.

Operation of QUEUE

1. Insert
2. Delete / serve
3. Display
4. Empty
5. Full

TYPES of QUEUE

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Deque



if (j == n)

(i = k+1) < n

k = 1

n-1 (i = 0 j = 3

#

```
void main()
{
    int i, j, n
    scanf("%d", n);
    for(i=0; i<n; i++)
    { for(j=0; j<n; j++)
        scanf("%d", a[i][j]);
    }
for(k=0; k<n; k++)
{ for(j=i; j<n; j++)
    printf("%d", a[i][j];
    k = i+1;
    j--;
while(k<n)
{   printf("%d", a[k][j]);
    k++;
}
k--;
for(j=n-2; j>=i; j--)
    printf("%d", a[k][j]);
j++;
for(k=n-2; k>j; k--)
    printf("%d", a[k][j]);
```

for (k=i+1; k<n; k++)

linking → loading → allocation
(compilation)

29/08/17

DYNAMIC MEMORY ALLOCATION

malloc :

void * malloc (size of (block));

e.g. p=(int *) malloc (size of (int));
q=(char *) malloc (size of (char));

int *p;
char *q;

if (P==NULL)

Using malloc(), allocate memory for an array of size n (dynamic array), value of n will be given by user

~~malloc (n*n);~~

n+2

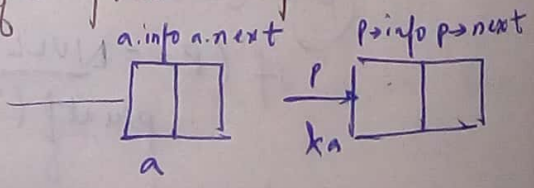
```
int n,*p,i, temp;
scanf("%d", n);
(p)= (int *) malloc (n * size of (int));
if (P==NULL)
    printf("no allocation");
else
{
    for (i=0; i<n; i++)
        scanf("%d", (p+i));

    for (i=0; i< n/2; i++)
    {
        temp = *(p+i);
        *(p+i) = *(p+n-1-i);
        *(p+n-1-i) = temp;
    }
    for (i=0; i<n; i++)
        printf("%d", *(p+i));
}
```

04/09/17

LINKED LIST

```
Typedef struct node
{   int info;
    struct node *next;
} node_type;
```

Q. Using static memory allocation allocate memory for 3 nodes a, b & c. Every node is having two parts. First part in which we can store integer data and in 2nd part we may store address or similar type of node. Input 3 no. x, y & z and store them in info fields of nodes a, b & c. Now make a LL by connecting b after a and c after b. A pointer start should point at the start of the LL, end of LL should be denoted by NULL. Using a pointer print the info fields of the entire linked list




```
struct node
{   int info;
    struct node *next;
} a, b, c, *start, *temp;
void main()
{
    int x, y, z;
    scanf("%d %d %d", &x, &y, &z);
    a.info = x; b.info = y; c.info = z;
    a.next = &b; b.next = &c; c.next = NULL;
    start = &a;
```

```
    temp = start;
    while (temp != NULL)
    {   printf ("%d", temp -> info);
        temp = temp -> next;
    }
}
```

Q. Using dynamic memory allocation create a LL. Every node is having two parts. First info of type int and second part next of same address of similar type. New link will always be added towards the right of the previous node. After creating every node user will be asked to continue. If yes. more nodes will be added.

```
struct node
{   int info;
    struct node *next;
};
void main ()
```

Q. Using dynamic memory allocation and LL implement stack.

ASSUMPTIONS : Using structure define a user defined complex data type node_type having two fields.

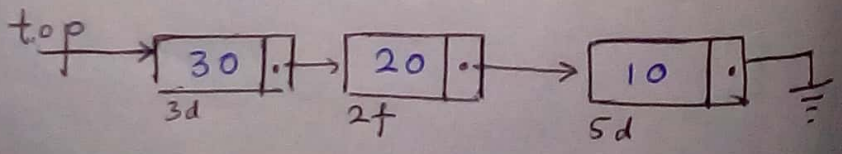
- info of type int
- next which is address of a node_type

Take a node_type pointer TOP and initiallize it to NULL as initially stack is empty.

ALGO (PUSH)

1. Begin / start
2. Allocate memory for a node_type and assign its address to a node_type pointer P
3. If P is equal to NULL, then display a message that no enough memory and goto step 7, otherwise goto step 4.
4. Input a no. which has to push in stack and assign it to P → info.
5. Next field of P should point to TOP.
6. Now TOP should point to P
7. End / Stop

ALGO (POP)



1. Begin / start
2. If TOP is equal to NULL, then display message STACK EMPTY, goto step 7.
3. Take a node_type pointer tmp and assign it value of TOP.

4. Display element popped is TOP → info.
5. Increment TOP to TOP → next
6. Deallocate memory addressed by Temp using free().
7. End / STOP

ALGO(DISPLAY)

1. Begin / start
2. If TOP is equal to NULL, then display message STACK EMPTY, goto step .
3. Take a node_type pointer tmp and assign it value of TOP.
4. ~~Display element~~ ~~tmp → info~~
5. ~~Increment~~ ~~tmp to tmp → next~~
4. While tmp is not equal to NULL, Repeat step 5-6
5. Display element tmp → info
6. Update tmp to tmp → next
7. End / stop

Q. Implement a stack using LL. ~~using~~

ASSUMPTIONS : Using structure define a user defined data type 'node_type' having two fields

- info of integer type
- next which is address of a node_type

Take a node_type pointer start initiallize with NULL and pointer of node_type p for dynamic memory allocation.

ALGORITHM

Step 1 : Begin

Step 2 : Using malloc( ) with parameter size of (node type) allocate a memory block dynamically and assign its address to p

Step 3 : Check if p is not equal to NULL

Step 4 : Input a num

Step 5 : Assign num to p→info

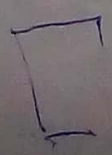
Step 6 : Assign start to p→next

Step 7 : Assign p to start

Step 8 : Input ch

Step 9 : Check if toupper(ch) is equal to 'Y' and p is not equal to NULL , Repeat step 2 - 9 , otherwise goto step 10.

Step 10 : End.



C PROGRAM

```
typedef struct node_ty
{   int info;
    struct node *next;
}node_type;
void main()
{   node_type *start, *P;
                =NULL
    int num; char ch;

do
{   p = (node_type *) malloc (sizeof (node_type));
    if (p != NULL)
    {scanf ("%d", &num);
    p → info = num;
    p → next = start;
    start = p;
    }printf ("Do you want to continue");
    scanf ("%c", &ch);
} while ((toupper(ch) == 'Y') && (p != NULL));
}
```

C PROGRAM

```
typedef struct stack
{
    int info;
    struct stack *next;
}node;

node *PUSH(node*);
node *POP(node*);
void DISP(node*);

void main()
{
    node *TOP=NULL;
    int ch;
    do
    {
        scanf("%d",&ch);
        switch(ch);
        {
            case 1:
            TOP=PUSH(TOP);
            break;
            case 2:
            TOP=POP(TOP);
            break;
            case 3:
            DISP(TOP);
    }
}while((ch<4)&&(ch>0)

node* PUSH(node* TOP)
{   node* P=NULL;
    P=(node*) malloc(sizeof(node));
```

```
if (P == NULL)
    printf (" no enough memory ");
else
{   scanf ("%d", p -> info);
    p -> next = TOP;
    TOP = p;
}
return (TOP);
}
node * POP (node * TOP)
    node * tmp = TOP
{ if (TOP == NULL)
        printf (" stack emphy ");
  tmp = TOP;
  else
  {   printf ("%d", TOP -> info);
  TOP = TOP -> next
  free (tmp);
  }
  return (TOP);
}
void DISP (node * TOP)
{ node * tmp = TOP;
  if (TOP == NULL)
        printf (" stack emphy ")
  else
  {   while (tmp != NULL)
      {   printf ("%d", tmp -> info);
          tmp = tmp -> next;
      }
}
```

Queue using linked list

11/09/17

ASSUMPTION:

ALGO (INSERT)

Step 1 : Begin

Step 2 : Allocate memory for a node_type and assign it's address to a node_type pointer p

Step 3 : Check if p is equal to NULL, then display message no enough memory, goto step 9, otherwise goto step 4.

Step 4 : Input a no. to insert in queue and assign it to p→info.

Step 5 : ~~Check if~~ Assign NULL to p→next.

Step 6 : Check if R is equal to NULL, then assign p to R, goto step 9, otherwise goto step 7.

Step 7 : ~~Assign~~ p to R→next

Step 8 : ~~Assign~~ p to R

Step 9 : Return R.

ALGO (DELETE)

Step 1 : Begin

Step 2 : Take a node_type pointer ~~tmp~~ and assign it value of f

Step 3 : Assign f→next to f.

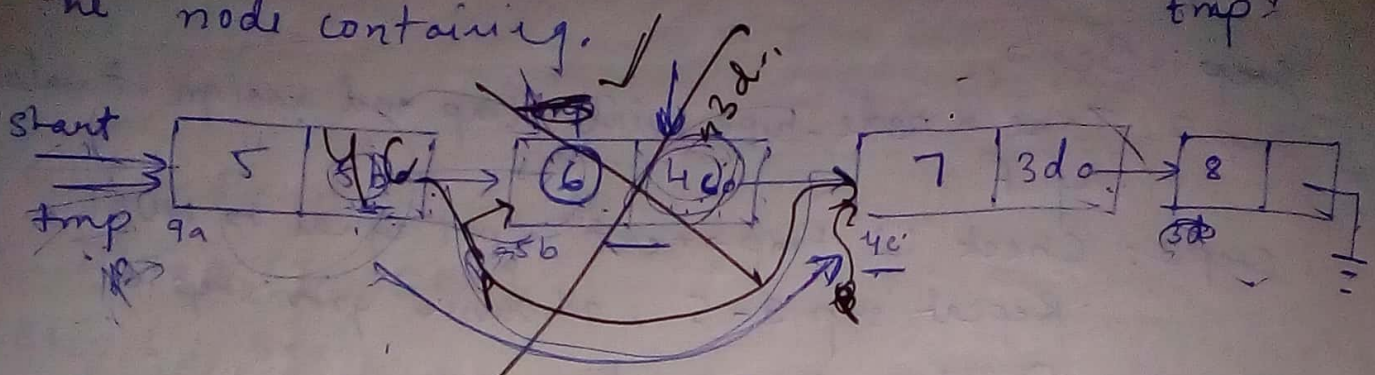
Step 4 : Display tmp→info.

Step 5 : Deallocate memory addressed by tmp using free()
Step 6 : Return F

ALGO (DISPLAY)

Step 1 : Begin
Step 2 : Take a node-type pointer tmp and assign it value of f
Step 3 : Check if tmp→next is not equal to NULL, Repeat step 3-5, otherwise goto step 6
Step 4 : Display tmp→info
Step 5 : Assign tmp→next to tmp
Step 6 : Display tmp→info
Step 7 : End

7. Assuming that we have a linked list with pointer start, input a no. and search it in the linked list, if it's found then update the linked list by deleting the node containing. tmp.



Case 1: found in 1st node
value of start has to be updated and 1st node has to be deleted.

In all other cases value of start will not change.

Two possibilities (no. found & no. not found)

p = tmp → next

tmp → next = ~~tmp~~ → next ~~next~~

free(p)

~~if (start → next = n)~~

~~{~~

~~if (start → next == NULL)~~

~~free(start);~~

~~}~~

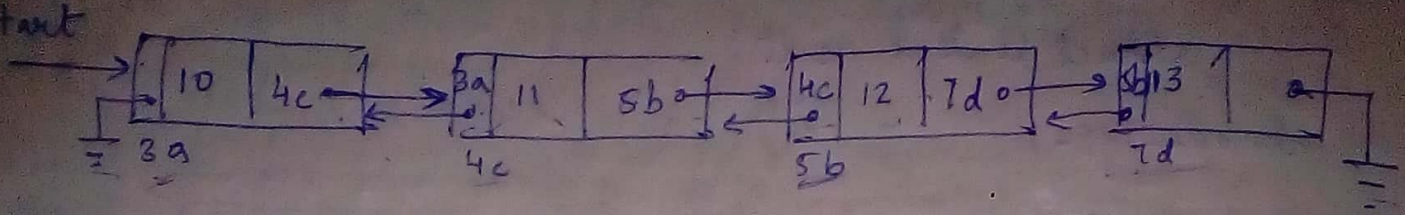
```
if (start → next == n)
{
    p = start;
    start = start → next;
```

```
typedef struct link
{
    int info;
    struct link * next;
}node_type;

void main ()
{
    node_type * tmp = start, * p;
    int n;
    scanf("%d", &n);
    if (start == NULL)
        printf ("NL");
    else if ( start -> info == n )
    {
        p = start;
        start = start -> next;
        printf ("%d", start -> info);
        free (p);
    }
else
{   while ( tmp != NULL )
    {   if ( tmp -> next -> info == n )
        {   p = tmp -> next;
            tmp -> next = p -> next;
            free (p); break;
        }
        else
            tmp = tmp -> next;
    }
    if (tmp == NULL)
        printf ("Not found");
}
```

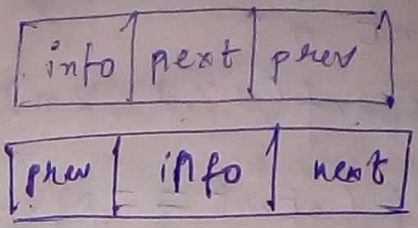
printf("%d",printf("abc"));

18/09/17

DOUBLY LINKED LIST




```
typedef struct dlink
{
    int info;
    struct dlink *next;
    struct dlink *prev;
}node_type;
```



Write an algorithm for creating a doubly linked list where new nodes are added towards the right. After creating doubly linked list print the contents from L to R & R to L.

ASSUMPTION

Using structure define a node type where fields are:

- prev of same type of node
- info of type int
- next of same type of node.

Take two pointers of type node_type and initialize them to NULL, as initially linked list is empty

ALGO FOR ADDING A NODE

Step 1: Begin

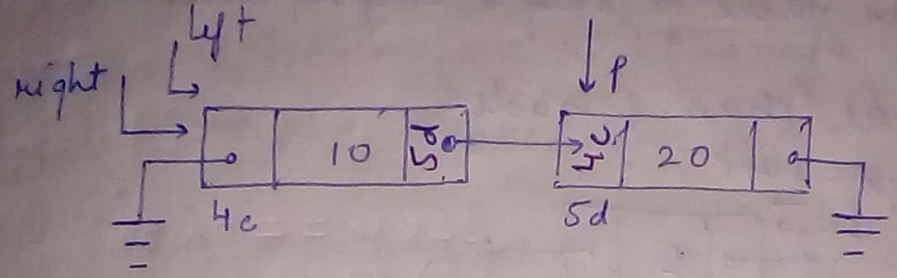
Step 2: Allocate memory for a node_type and assign it's address to a node_type pointer P

If p is equal to NULL, the display a message no enough memory : goto step.

step 4 : Input a no. and assign it to p→info.

step 5 : p→next should point to NULL.

step 6 : If p is equal to NULL, it means node inserted is first node and p→prev should point to NULL goto step



right→next = p;
p→prev = right;
right = p;

ALGO FOR PRINTING R to L

step 1 : Begin

step 2 : If right is equal to NULL, then print empty list, goto step 7

step 3 : Take a node-type pointer tmp, assign it value of right.

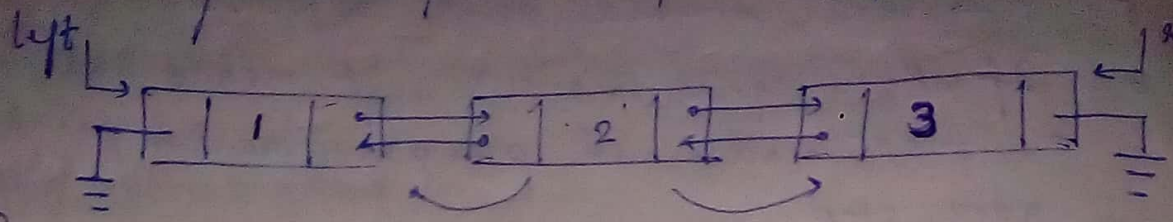
step 4 : While ~~right is~~ tmp is not equal to NULL, repeat step 4-6, otherwise goto step 7

step 5 : print tmp→info

step 6 : Assign tmp to tmp→prev.

step 7 : End

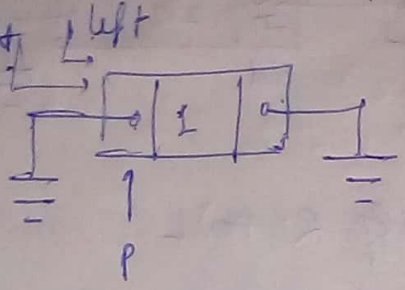
Create a doubly linked list so that all elements are in ascending order from left to right and descending order from right to left.



Case 1 : When there is only a single node.

Case 2 : More than 1 nodes

Case 0 :

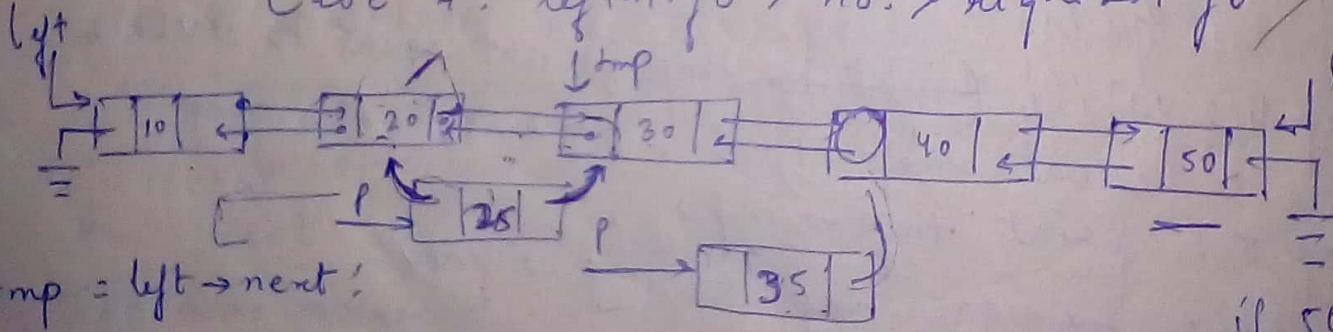


Case 1 : 1st node

Case 2 : no. < left → info

Case 3 : no. > right → info

Case 4 : left → info > no. > right → info



tmp = left → next;

while (p → info > tmp → info)

tmp = tmp → next;

p → next = tmp;

p → prev = tmp → prev;

tmp → prev → next = p;

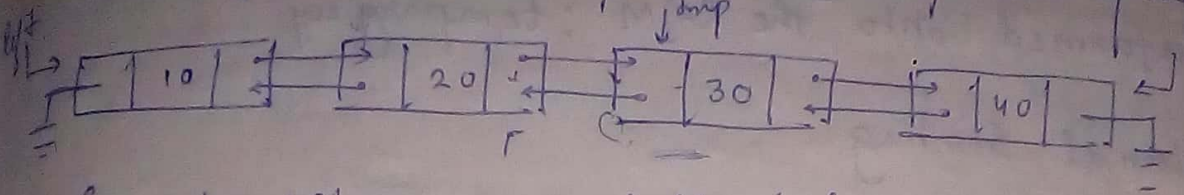
tmp → prev = p;

if 50 + 10

right → info + left → info / 2

10 40 50 60 70

Assuming we already have a doubly linked list with pointers left and right. WAP to input a no. search that no. in doubly linked list, if no. is found, then update the doubly linked list by deleting the no.



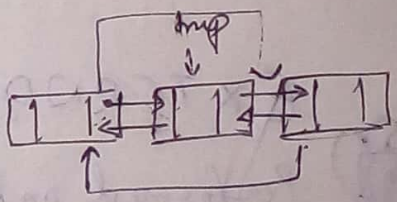
Case 1: When no. == left->info

Case 2: when no. == right->info

case 3: In the linked list_b/w / not found

```
if(left == NULL && RIGHT == NULL)

else if (no. == left->info
```

```
while ((tmp->info != no.) && (tmp != NULL))
{
        tmp = tmp->next;
if (tmp == NULL)
        no. not found
else
{   if (num == left->info)
    { left = left->next;
      left->prev = NULL; }

    if (num == right->info)
    {
        right = right->prev;
        right->next = NULL;
    }

else
{
    tmp->prev->next = tmp->next;
    tmp->next->prev = tmp->prev;
}
free (tmp);
```



SORTING

03/10/17

TYPES of SORTING

1. Internal Sorting
   Performed onto the RAM : temporary
2. External Sorting

Techniques of Internal Sorting

1. Bubble
2. Selection
3. Insertion
4. Shell
5. Counting
6. Radix

EXTERNAL SORTING

Performed onto the external storage memory like HDD

Techniques

1. Quick
2. Merge

] based on divide & conquer approach

→ good locality of referrance.

Virtual Memory : Logical extention of memory from HDD used by the OS for swapping

BUBBLE SORTING
(Exchange Sort)

6 3 20 4 5 7

3 6 20 4 5 7

1st Pass gives the largest element.

SELECTION SORTING

6 3 20 4 5 7

find min. in Pass 1

Swap a[0] with min.

3 6 20 4 5 7

```
for (i = 0; i<n-1; i++)
{   min = a[i];
    pos = i;
    for (j=i+1; j<n; j++)
    {
        if (a[j] < min)
        {
            min = a[j];
            pos = j;
        }
    }
    if (i!=pos)
    {
        a[pos] = a[i];
        a[i] = min;
    }
}
```

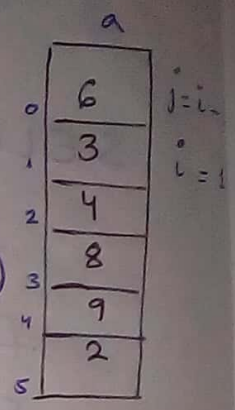
INSERTION SORTING

Incremental sort

Best sorting algorithm among Bubble, Selection, Insertion.

~~for (i=1; i<n; i++)~~
~~{ tmp = a[i];~~
~~for (j=i-1; j>=0; j--)~~
~~{ if (tmp < a[j])~~
~~{ a[j+1] = a[j];~~

~~tmp = a[i]~~
~~j = i-1~~
~~while (j>=0 && tmp < a[j])~~
~~}~~




```
for (i = 1; i < n; i++)
{
    tmp = a[i];
    j = i - 1;
    while (j >= 0 && tmp < a[j])
    {
        a[j+1] = a[j];
        j--;
    }

    a[j+1] = tmp;
}
```

QUICK SORT

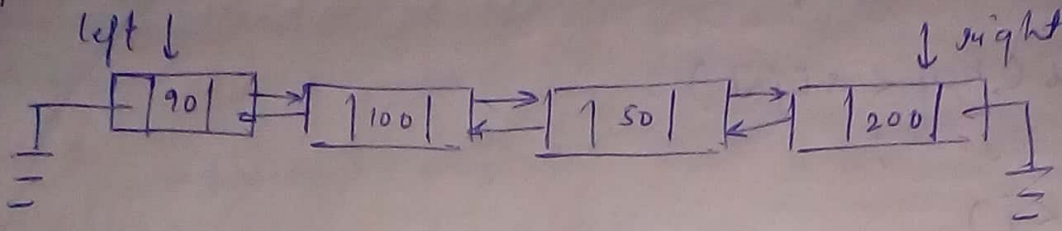
lb = 0 ub = 5

i = 0	16	6	2	50	40	80	j = 5
↑						↑	

key = a[lb]

```
while (key >= a[i] && i<j)
        i++ ;

while (key < a[j])
        j-- ;

if (i<j)
{
   tmp = a[i]
   a[i] = a[j];
   a[j] = tmp;
}
}

a[lb] = a[j];
a[j] = key;
Quicksort(a, lb, j-1);
Quicksort(a, j-1, ub);
}

void Quicksort (int a[], int lb, int ub)
{
   int i = lb, j = ub, key = a[lb], tmp = 0;

   if (lb >= ub)
        return;
```

Assuming that we already have a DLL with pointers left and right. Write a program to input a key and then such that key in DLL, if key is found then LL should be updated by deleting the node where it is found.



ALGO

Step 1: Begin

Step 2: Check if left and right are equal to NULL, then goto step

Step 3: Input key

Step 4: Take a node_type pointer tmp pointing to left.

Step 5: while (tmp→info != key && tmp != NULL)

Step 6: tmp = tmp→next

Step 7: Check if tmp == NULL, the display a message "key not found in LL", goto step

Step 8: tmp→prev→next = tmp→next;

Step 8.1: tmp→next→prev = tmp→prev;

Step 8.2: free(tmp);

Step 7.1: if (tmp == left);

left = left→next;

left→prev = NULL;

free(tmp);

```
    right = right -> prev ;
    right -> next = NULL ;
    free (tmp) ;
```

~~node_type search (node_type ** left , int key)~~

```
if (pos == right )
  { right = right -> prev ;
    right -> next = NULL ;
  }
else
   left = updat (left , pos) ;
```

24/10/17

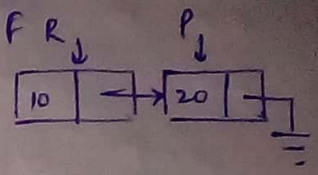
CALL BY REFERRENCE
DOUBLE POINTER

STACK

```
void push( node_type ** ) ;

void push (node_type ** top)
{  node_type * p ;
   p = (node_type *) malloc (size of (node type)) ;
    if (p != NULL )
    {     scanf (" %d", p -> info ) ;
          p -> next = * top
           * top = p ;
    }
}
```

```
void pop(node_type **);
void pop(node_type **top).
{
    node_type* tmp =*top;
    if (*top != NULL)
    {
        printf("%d",*top->info)
    *top =(*top)->next
        free(tmp);
    }
}

void display(node_type*);
void display(node_type * top)
{   node_type * tmp = top;
    if(top != NULL)
    {   while(tmp != NULL)
        {   printf("%d", tmp->info);
            tmp = tmp->next;
        }
    } else printf("no nodes");
}
```

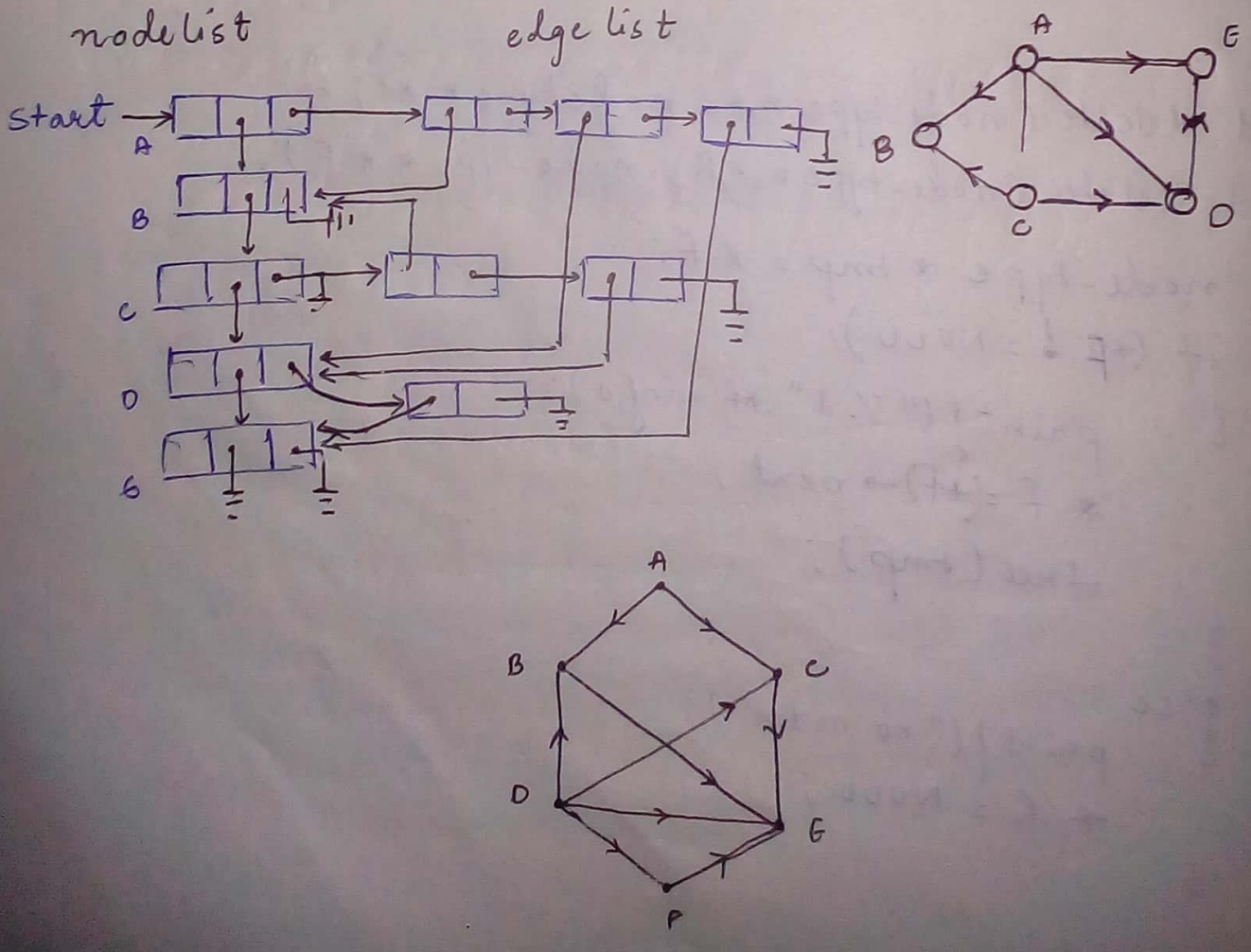
```
void insert (node_type **R , node_type **F)
{
  node_type *p;
  p = (node_type*) malloc (sizeof (node_type));
  if (p != NULL)
  {
    scanf ("%d", p->info);
    p->next = NULL;
    if (*R == NULL)
    { *R = p ; *F = p }
    else
    {
      *R->next = p
       *R = p ;
    }
  }
}
void delete (node_type **, node_type **);
void delete (node_type **R, node_type **F).
{
  node_type *tmp = *F;
  if (*F != NULL)
  {
    printf ("%d", *f->info);
    *F = (*f)->next;
    free (tmp);
  }
  else
  {
    printf ("no nodes");
    *R = NULL;
  }
```

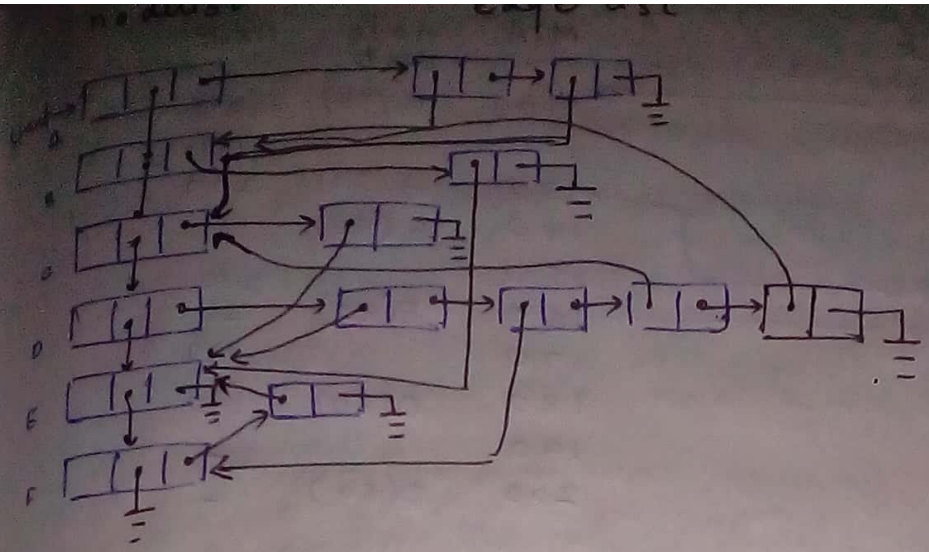


```
void display (node_type * );
void display (node_type * f )
{
    node_type * tmp = f;
    if (f != NULL)
    {   while (tmp != NULL)
        {   printf("%d", tmp -> info);
            tmp = tmp -> next;
        }
    }
    else
        printf("no nodes");
}
```

25/10/17

LINKED REPRESENTATION of GRAPH





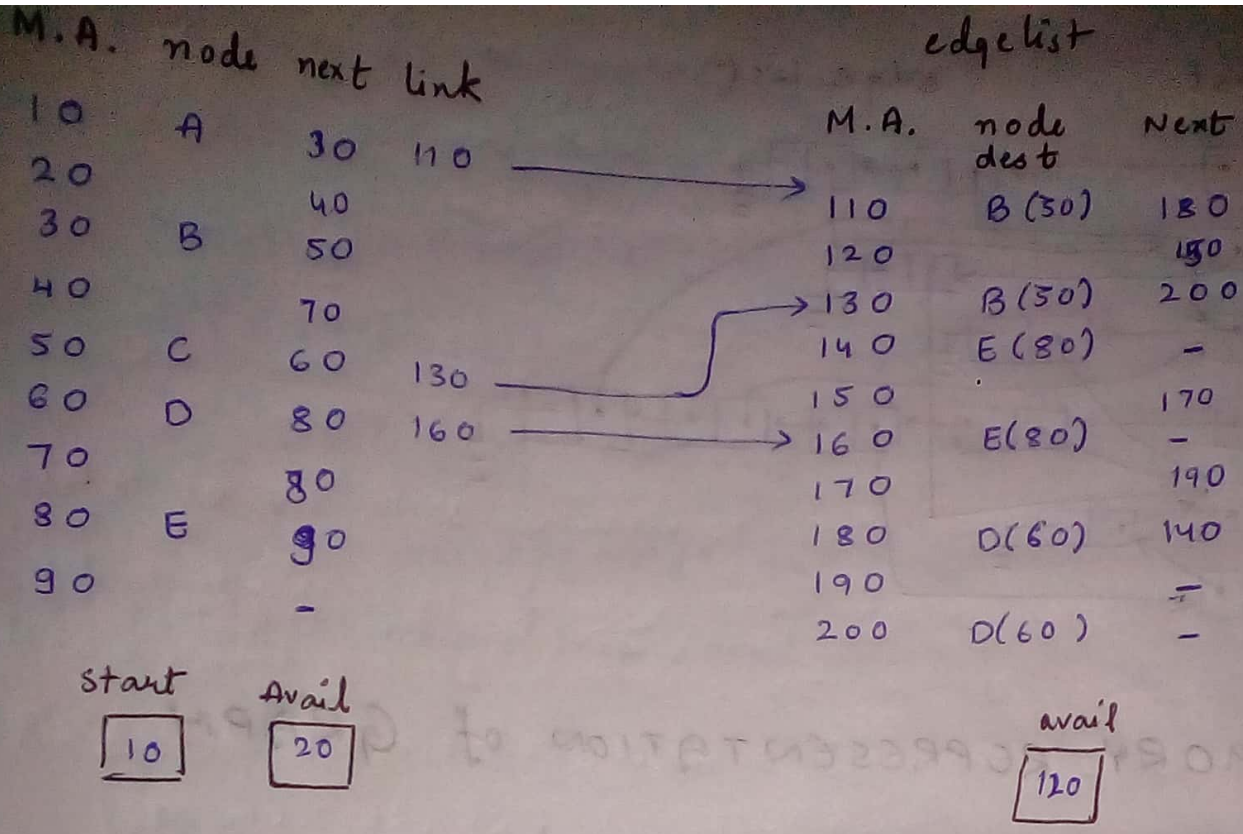
MEMORY REPRESENTATION of GRAPH

Memory Address | node | next | LINK

Start 50	10	C	70	250
	20		40	
	30	B	10	210
Avail 20	40		60	
	50	A	30	230
	60		80	
	70	D	110	270
	80		100	
	90	E	-	-
	100			
	110	F	90	280

nodelist → C, B, A, D, F

	Mem Add	DEST	NEXT
E	200	C(10)	290
	210	E(90)	
	220		
A	230	B(30)	270
	240	C(10)	-
C	250	E(90)	-
	260	E(90)	-
D	270	B(30)	200
F	280	E(90)	-
	290	F(110)	260



26/10/17

CIRCULAR LINKED LIST →

ALGORITHM :

for INSERT

```
void insert (nodetype **fst, node_type **lst)
{
    node_type *p;
    p = (node_type*) malloc (sizeof(node_type));
    if (p != NULL)
    {
        scanf("%d", p->info);
        if (*lst == NULL)
        {
            *lst = p; *fst = p;
            *lst -> next = *fst;
        }
        else
        {
            *lst -> next = p;
            *lst = p;
            *lst -> next = *fst;
        }
```

Assuming that we already have a circular LL with pointer start, write a function to count total no. of nodes, function does not return any value but the total no. of nodes are printed in main function.

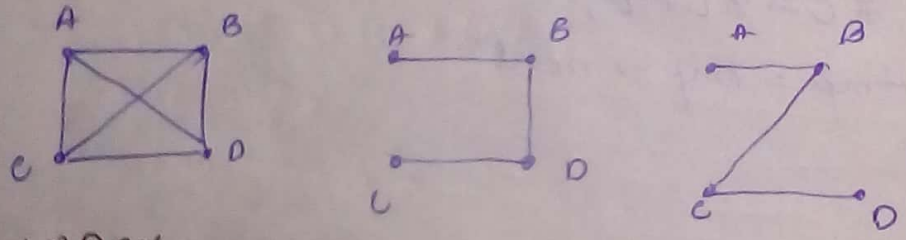
```
void count(nodetype **, int *)

void main()
{
    int count=0;
    count(start, &count);
    printf("%d", count);
}
void count(node_type **start, int *c)
{
    node_type *tmp = start;
    if(*start != NULL)
    {
        while(tmp != start)
        {
            *c = *c + 1;
            tmp = tmp -> next
        }
    }
}
```

Assuming that we already have a circular LL, with pointer start, write a fn which prints last element first, then first element, then second element and so on:

```
void print(node_type * start
{
    node_type * tmp = start;
    while(tmp -> next != start)
        tmp = tmp -> next;
    printf("%d", tmp -> info); tmp = tmp -> next;
```

```
while (tmp->next != start)
{
    printf("%d", tmp->info);
    tmp = tmp->next;
}
}
```

WAP to do time scheduling for jobs j1, j2..... on a multiprogramming environment on time sharing basis.. At a time a time slot of 10 sec will be alloted to a job. Create a Circular LL consisting of job no. j1, j2..., total time required to execute that job and address of node having description of next job. Execute this program to find out which job will be completed when. Once a job is completed its link should be deleted from LL.



30/10/17

BINARY TREE

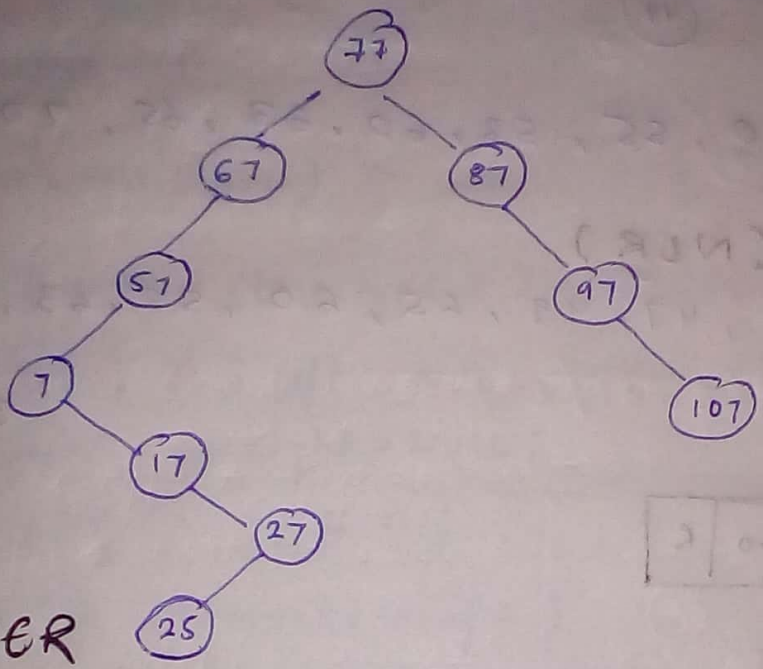
A binary tree is a tree where every node has at the max. 2 children, means no child or 1 child or 2 children.

In binary tree, we follow left-right relationship, it means that a node either does not have any child or one left or right child or both left and right children.

BINARY SEARCH TREE

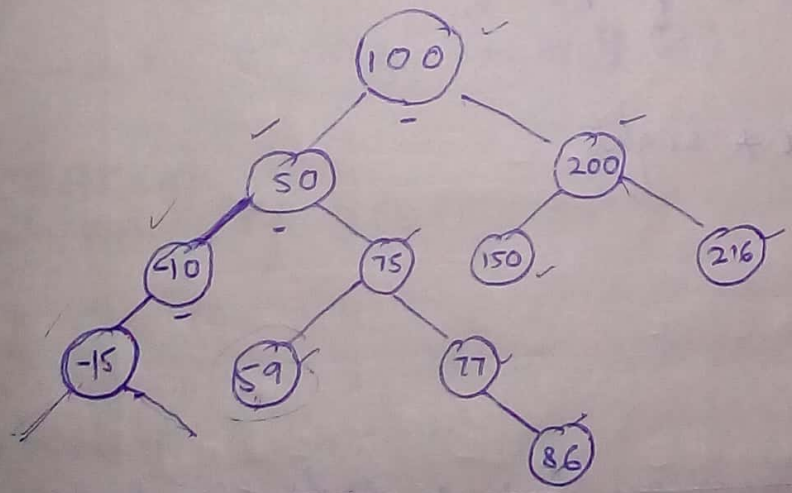
In binary search tree, in every node the left child has less data than the parent and the right child has more data than the parent.

77, 87, 67, 57, 97, 107, 7, 17, 27, 25



INORDER

100, 200, 50, 75, 59, 150, 216, 77, 86, -10, -15



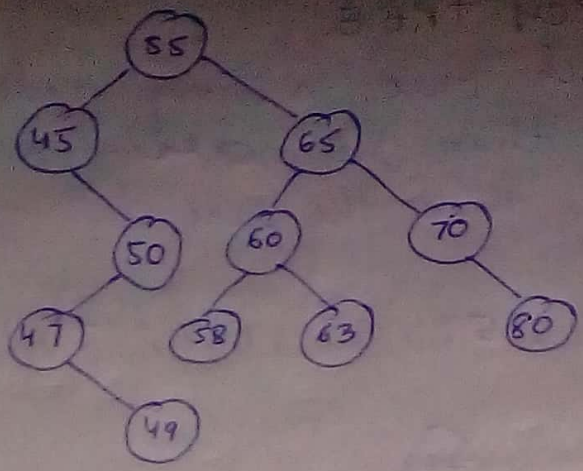
-15, -10, 50, 59, 75, 77, 86, 100, 150, 200, 216

PREORDER (NLR)

100, 50, 10, -15, 75, 59, 77, 86, 200, 150, 216

POST ORDER (LRN)

-15, -10, 59, 86, 77, 75, 50, 150, 216, 200, 100



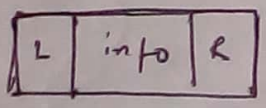
45, 47, 49, 50, 55, 58, 60, 63, 65, 70, 80

PRE-ORDER (NLR)

55, 45, 50, 47, 49, 65, 60, 58, 63, 70, 80

31|10|17

Binary Search Tree




```
typedef struct node
{   struct node *left;
    int info
    struct node *right;
} tree_type;
```

Using Recursion

Step 1. Begin

Step 2. ~~Allocate memory for a node tree_type~~

If root == NULL

root → p

root→info = num

L & R == NULL