NOTES (by Dr. Puneet Goyal)

Stack, also called **LIFO** system, is a linear list of elements in which an element may be inserted or deleted only at one end, called the top of the stack.

Operations on Stack:

CreateStack(S), Push(S,i), Pop(S), Top(S), IsFull(S),IsEmpty(S) –

All of these operations in O(1) time

Applications of Stacks: Parenthesis checker, Evaluation of Postfix expression, 'Back' functionality while web browsing, ……

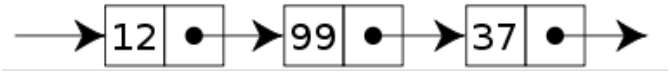
Stack implementation and representation using Arrays (Static implementation)

```
# define N 50
typedef struct {
        int elem[N];
        int top; //Assume top be index of the last element in the stack
         } Stack;
Typedef enum {false, true} boolean;

CreateStack(Stack *ps) {
        ps->top = -1; // We could also write (*ps).top = -1;
        }

Push(Stack *ps, int value) {
        ps->top += 1;
        ps->elem[ps->top] = value;
        }

int Pop(Stack *ps) {
        return ps->elem[ps->top--] ;
        }

____ Top(Stack *ps) {
        ……..
        }
```

Exercise: Write the prog using these functions where you push elements 10,16,93,40,50 and then 3 pop() calls, and then top() followed by 2 more push() for elements 140 & 70 and at last a pop() operation. What will be the final state of Stack?

Limitation of array-based representation of Stacks: Although static implementation using arrays is a simple technique but not flexible of creation, as the size of the stack must be known in advance, and it cannot be varied later; so it may lead to overflow situation. Also its not efficient with respect to memory utilization.

A linked list is a linear collection of data elements, called *nodes*, where the linear order is given by means of pointers. Each node has 2 fields: *data field* and *link field*, which contains the address of the next node in the list.



Linked list allows only sequential access to elements.

The ***self-referential structures*** contain pointers within the structs that **refer** to another identical structure. Linked lists are the most basic self-referential structures. Linked lists allow you to have a chain of structs with related data.

```
typedef struct node
        { int val; struct node *next; } Node;
struct node a,b,c;
a.val=12; b.val=99; c.val=37;
a.next=&b;
b.next=&c;

a.next.next=&c;
a.next ->next=&c;
a.*next.next=&c

*(a.next).next=&c;
(a.next)->next=&c;

struct node *x, *head;
x= (struct node *) malloc (sizeof (struct node));
x->value=12; x->next=NULL;
head=x;
(*x).value =
_____..
_____..
x->next=NULL;
head->next=x;
```

Types of Linked List: Singly Linked List, Doubly Linked List, Circular Linked list,

---

Node* insert(struct node *head, int val)

```
{
  Node *nn, *p;
  nn= (struct node*) malloc(sizeof(struct node));
  nn->val=val;
  nn->n=NULL;
  if (head==NULL)
    { head=nn; printf("\n Inserted Head"); }
  else
     { p=head;
     while (p->n != NULL) p=p->n;
     p->n=nn;
     }
  Return;
}
```

int main(int argc, char **argv)

```
{
Node *head;
head = (struct node*) malloc(sizeof(struct node));
head->val=10; head->n =NULL;
insert(head,12);
insert(head,99);
insert(head,37);
insert(head,40);
delete_val(head, 40);
display(head);
}
```

void display(struct node *hh)

```
{
 Node *p = hh;
  if (hh==NULL)
    { printf("\n Error : No Node to Display"); return; }
 printf("\n");
 while (p)
   {
   printf("%d, ", p->val);
            // if (p == p->n) break; else
   p=p->n;
   }
}
```

Node* delete_val(struct node *hh, int val) {

```
  Node *nn, *p; int x=-1;
  if (hh==NULL)   //CASE 1
   { printf("\n Error2 : No Node."); return ; }
  else if (hh->n == NULL) { // CASE 2
        p=hh;
        if (p->val == val)
          { hh=NULL;
            free(p);
            printf("\n Only Head & Value matched.Head now pointing to NULL");
            return hh;
          }
        else {
           printf("\n Only Head but Value not matched");
           return hh;
           }
       } //End of CASE 2
    else {  //CASE 3
          p=hh; nn= p->n;
          if (p->val == val) //CASE 3a
          { hh=nn;
           p->n = NULL;
           free(p);
           printf("\n Value matched with head element. Head shifted");
           return hh;
          }
          else {//CASE 3b
            while (nn != NULL && nn->val != val)
                  { p=nn; nn=p->n; }
            if (nn == NULL) {
                  printf("\n  Value not found");
                  return hh;
                  }
             else if (nn->val == val) {
                 p->n=nn->n;
                 nn->n=NULL;
                 free(nn);
                 printf("\n Value matched with some element.");
                 return hh;
                 }
            } //End of CASE 3b
     } //End of CASE 3
 }
```

Node* delete_last(struct node *hh) {

```
  Node *nn, *p; int x=-1;
  if (hh==NULL)
   { printf("\n Error : No Node There. Nothing deleted"); return hh; }
  else if (hh->n == NULL)      {
```

```
    p=hh;
    hh=NULL;
    x=p->val;
    free(p); return(hh);
   }
   else {
      p=hh; nn= p->n;
      while (nn->n != NULL) { p=nn; nn=p->n; }
      x=nn->val;
      p->n=NULL;
      free(nn);
      return(hh);
    }
}
```

Advantages: Dynamic data structures (can grow or shrink during the execution), Efficient memory utilization (memory not pre-allocated), insertions and deletions are easier and efficient, can be used in many applications.

Disadvantages: Access to any arbitrary element is bit cumbersome and time consuming, Overhead could be more at times ….Extra space needed for references…

Stack implementation using Linked List;

```
Struct stack {
Struct node *stack_top;
}

Int main () {
Struct stack * st1;
St1 = (..) malloc (size of (     ) );
Push(st1,10);
Push(st1,30);
Push(st1,16);

Struct stack *push (Struct stack *st1, int v1) {
Node *nn, *p;
  nn= (struct node*) malloc(sizeof(struct node));
  nn->val = v1;
  nn->n=st1->stack_top;  // Check if NULL
  st1->stack_top=nn;
  return (st1);
}
```

```
Int pop (Struct stack *st1, int v1) {
Node *nn, *p;  int x;
…

 nn=st1->stack_top;
 st1->stack_top=nn->n;
  x=nn->val;

free(nn); return x;

}
```

The previous code includes an important introduction: the arrow operator (`->`). This is a dereference operator that is used exclusively with pointers to objects with members. This operator serves to access a member of an object to which we have a reference. In the example we used: **pmovie->title**

Which is for all purposes equivalent to: **`(*pmovie).title`**

Both expressions `pmovie->title` and `(*pmovie).title` are valid and both mean that we are evaluating the member `title` of the data structure **pointed by** a pointer called `pmovie`. It must be clearly differentiated from:

which is equivalent to:

```
*pmovie.title
*(pmovie.title)
```

And that would access the value pointed by a hypothetical pointer member called `title` of the structure object `pmovie` (which in this case would not be a pointer). The following panel summarizes possible combinations of pointers and structure members:

Expression	What is evaluated	Equivalent
a.b	Member b of object a	
a->b	Member b of object pointed by a	(*a).b
*a.b	Value pointed by member b of object a	*(a.b)

MALLOC():

void* malloc(size_t size) Request a contiguous block of memory of the given size in the heap. **malloc()** returns a pointer to the heap block or NULL if the request could not be satisfied. The type size_t is essentially an unsigned long which indicates how large a block the caller would like measured in bytes. Because the block pointer returned by malloc() is a void* (i.e. it makes no claim about the type of its pointee), a cast will probably be required when storing the void* pointer into a regular typed pointer.

A call to malloc, with an error check, typically looks something like this:

```
int *ip = (int *) malloc(100 * sizeof(int));
if(ip == NULL)
        {
        printf("out of memory\n");
        exit or return
        }
```

Additional note on MALLOC():

In computing, **`malloc`** is a subroutine for performing dynamic memory allocation in the C and C++ programming languages.

The C prog.language manages memory - statically, automatically, or dynamically.

Static-duration variables are allocated in main (fixed) memory and persist for the lifetime of the program;

Automatic-duration variables are allocated on the stack and come and go as functions are called and return.

For static-duration & automatic-duration variables, the size of the allocation is required to be compile-time constant. If the required size is not known until run-time (for example, if data of arbitrary size is being read from the user or from a disk file), then using fixed-size data objects is inadequate.

The lifetime of allocated memory is also a concern. Neither static- nor automatic-duration memory is adequate for all situations. Automatic-allocated data cannot persist across multiple function calls, while static data persists for the life of the program whether it is needed or not. In many situations the programmer requires greater flexibility in managing the lifetime of allocated memory.

These limitations are avoided by using dynamic memory allocation in which memory is more explicitly (but more flexibly) managed, typically, by allocating it from the heap, an area of memory structured for this purpose. In C, the library function **`malloc`** is used to allocate a block of memory on the heap. The program accesses this block of memory via pointer that **`malloc`** returns. When the memory is no longer needed, the pointer is passed to ***`free`*** which deallocates the memory so that it can be used for other purposes.

Limitation of arrays in C Programming

1 . Static Data

- Array is Static data Structure
- Memory Allocated during Compile time.
- Once Memory is allocated at Compile Time it Cannot be Changed during Run-time

2 . Data types

- Elements belonging to different data types cannot be stored in array
- Example : character and integer cannot be stored in single array

3 . Insertion

- Inserting element is very difficult because before while inserting we have to create space by shifting other elements .

4 . Deletion

- Deletion is not easy because the elements are stored in contiguous memory location.

5 . Bound Checking

- C does not perform Bound Checking
- If the array range exceeds then garbage value may get overwritten.

6 . Shortage of Memory

- Array is Static .
- Shortage of Memory , if we don't know the size of memory in advance

7 . Wastage of Memory

- Wastage of Memory , if array of large size is defined