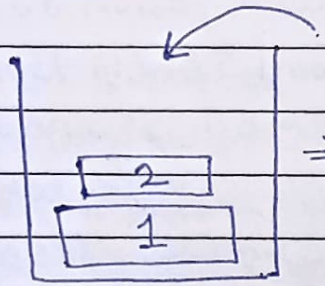
Lec-22 Introduction to Stack in Data Structures:

Stack is a linear data structure. Operations on Stack are performed in LIFO (Last in First Out) order.



⇒ Item 2 which entered the basket last, will be the first one to come out

LIFO (Last in First Out)

Applications of Stack:

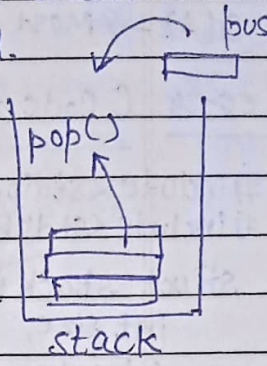
→ Used in function calls
→ Infix to postfix Conversion (and other similar Conversions)
→ Parenthesis matching & More.

Stack ADT : In order to create a stack we need a pointer to the top most element along with other elements which are stored inside the stack.

Green House
Date
Page No.

Some of the Operations of Stack ADT are:

1. push () → push an element into the stack.
2. pop () → remove the topmost element from the stack.
3. peek (index) → value at a given position is returned.
4. is Empty / is full () → Determine whether the stack is empty or full.



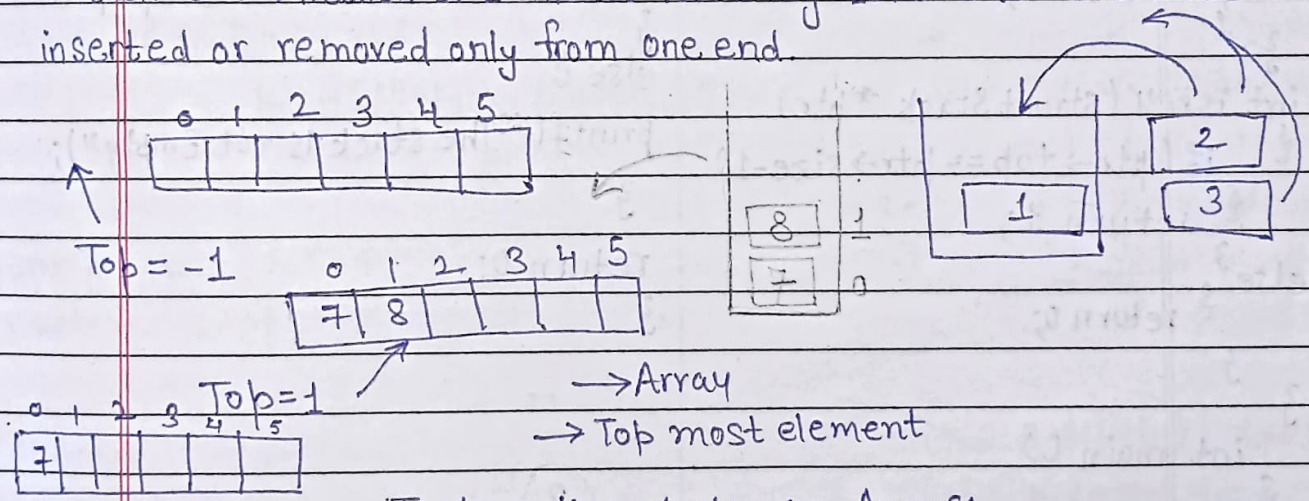
Amb

Implementation:

→ A stack is a Collection of elements with certain operations following LIFO (Last in First Out) discipline.
→ A stack can be implemented using an array or a linked list.

Lec-23 Implementing Stack Using Array in Data Structure:

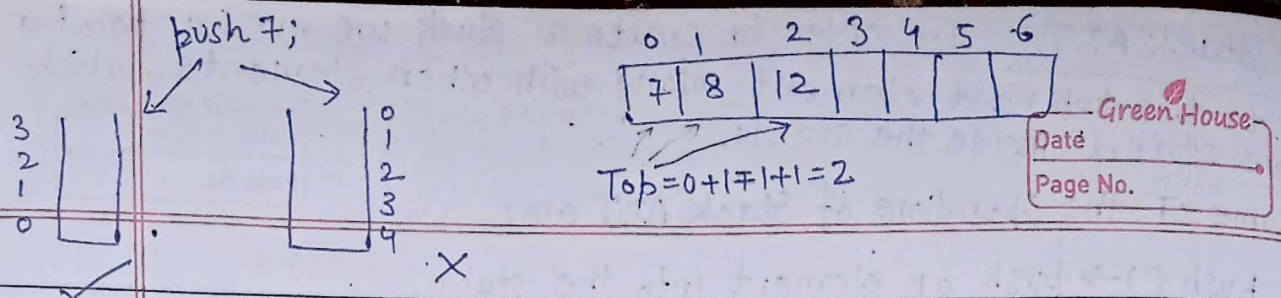
→ Stack is a Collection of elements following LIFO. Items can be inserted or removed only from one end.



Implementing stack using Arrays:
→ Fixed Size Array Creation
→ Top Element

```
Struct stack S
    int size;
    int top;
    int *arr;
}
```

```
struct stack S;
S.size = 80;
S.top = -1;
S.arr = (int *) malloc ( S.size * sizeof (int));
return S.arr[S.top--] ← For pop() operation
```



Green House
Date
Page No.

Ease;

O(1) → Most of the Operations

Lec-24 C Code for Implementing Stack Using Array in Data Structure:

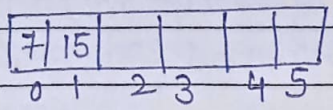
```
#include <stdio.h>
#include <stdlib.h>
struct stack {
    int size;
    int top;
    int *arr;
};
int isEmpty(struct stack *ptr)
{
  if(ptr->top==-1)
   { return 1;
    }
else { return 0;
    }
}
int isFull(struct stack *ptr)
{  if(ptr->top==ptr->size-1)
   { return 1;
else } { return 0;
    }
}
   int main()
{
  // p Struct stack s;=(struct stack*)malloc(sizeof(struct stack));
  //    s.size=80;
  //    s.top= -1;
  //  s.arr=(int*)malloc(s.size * sizeof(int));
```

```
Struct stack *s;= malloc(sizeof(struct stack*));
s->size=80;
s->top= -1;
s->arr=(int*)malloc(s->size *
                    sizeof(int));
// pushing an element manually
// s->arr[0]=7;
// s->top++;
// check if stack is empty
if(isEmpty(s)){
  printf("The stack is Empty");
}
else {
printf("The stack is not Empty");
 }
return 0;
}
```

Lec-25. Push, Pop and other Operations in stack Implented Using an Array.

Green House
Date
Page No.

→ Stack Operations while Implementing Using Arrays:



→ Cannot push if full
→ Cannot pop if Empty

```
struct stack {
    int size;
    int top;
    int *arr;
};
```

Creating a stack →

```
struct stack s;
s.size = ;
s.top = -1;
s.arr =
```

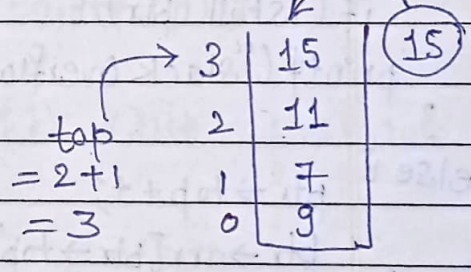
```
struct stack *sp {
    s->size =  ;
    s->top =  ;
    s->arry =  ;
```

Operation:1 Push():

```
struct stack *sp;
sp->size = 8;
sp->top = -1;
sp->arr = (int *)malloc(sp->size * sizeof(int));
```

→ `if (isfull(sp)) {`
`    printf("stack Overflow");`
`}`
`else {`
`    sp->top++;`
`    sp->arr[sp->top] = val;`
`}`

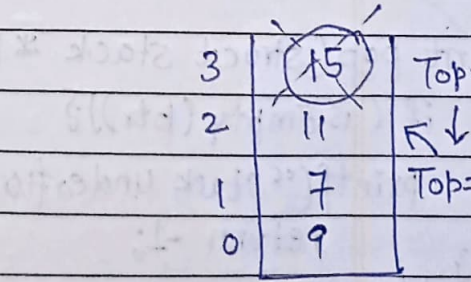
Nothing is Returned



Operation:2 pop()

```
if (isEmpty(sp)) {
    printf("stack underflow");
    return -1;
} else {
    int val = sp->arr[sp->top];
    sp->top = sp->top-1;
    return val;
}
```

Store the topmost value



Lec-26. Coding Push(), Pop(), isEmpty() and isFull() Operations in stack using C Language:

Green House
Date
Page No.

```
#include<stdio.h>
#include<stdlib.h>
struct stack{
    int size;
    int top;
    int *arr;
};
int isEmpty(struct stack *ptr){
    if(ptr->top==-1){
        return 1;
    }
    else{
        return 0;
    }
}
```

isEmpty Function.

```
int isFull(struct stack *ptr){
    if(ptr->top==ptr->size-1){
        return 1;
    }
    else{
        return 0;
    }
}
```

isFull Function.

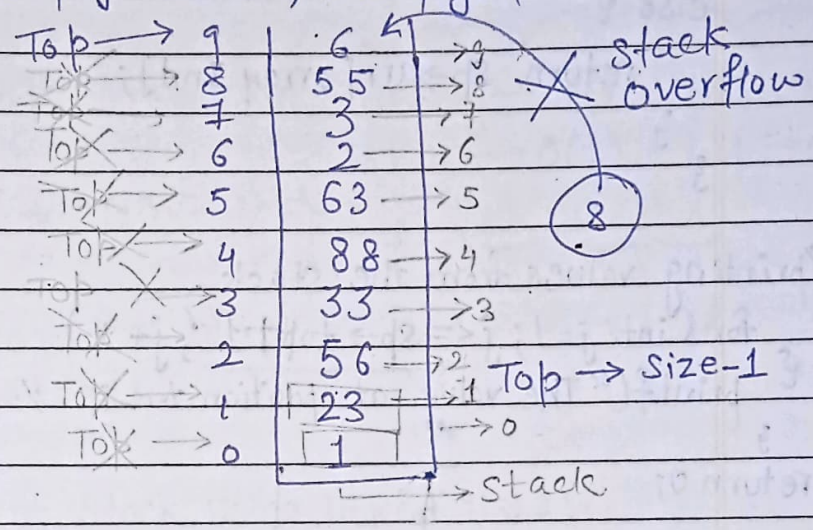
```
void push(struct stack *ptr, int val){
    if(isFull(ptr)){
        printf("Stack Overflow! Cannot push %d to the stack\n",val);
    }
    else{
        ptr->top++;
        ptr->arr[ptr->top]=val;
    }
}
```

← push Function

```
int pop(struct stack *ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow! Cannot pop from the stack\n");
        return -1;
    }
    else{
        int val=ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}
```

← pop Function

```
int main (){
  struct stack *sp=(struct stack *) malloc (sizeof. (struct stack));
  sp->size=10;
  sp->top= -1;
  sp->arr = (int *) malloc(sp->size * sizeof(int));
  printf("stack has been created Successfully \n");
  printf("Before pushing, Full: %d \n", isFull (sp));
  printf("Before pushing, Empty: %d \n", isEmpty (sp));
  push (sp, 1);
  push (sp, 23);
  push(sp, 56);
  push(sp, 33);
  push (sp, 88);
  push (sp, 63);
  push (sp, 2);
  push (sp, 3);
  push (sp, 55);
  push (sp, 6);    // ---> pushed 10 values
  push(sp, 8);  // Stack Overflow, since the size of stack is 10
  printf("After pushing, Full: %d \n", isFull (sp));
  printf("After pushing, Empty: %d \n", isEmpty (sp));

  printf("poped %d from the stack \n", pop(sp)); // List in First Out

  return 0;
}
```



Output

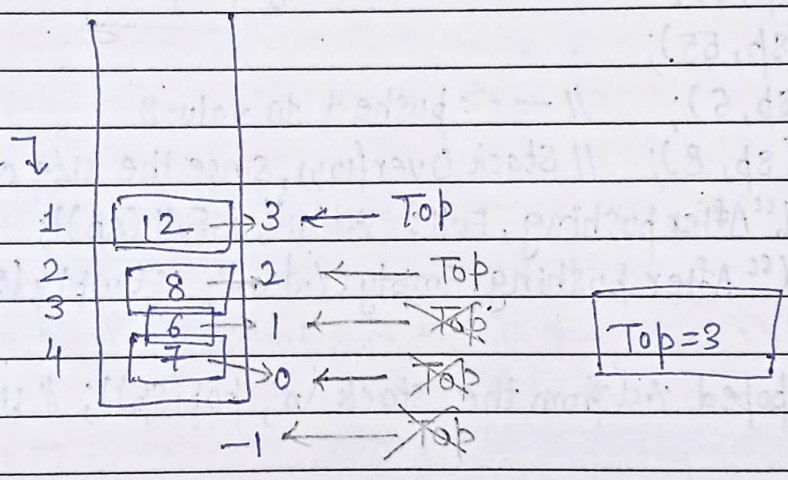
```
Stack is Created Successfully
Before pushing, Full : 0
Before pushing, Empty : 1
Stack Overflow ! cannot push 8 to the stack
After pushing, Full : 1
After pushing, Empty : 0
poped 6 from the stack
```

Lec-27 Peek Operation in Stack using Arrays (with C Code & Explanation)

```
int peek (struct stack * sp, int i){
  int arrayInd = sp->top - i + 1;
  if(arrayInd < 0){
  printf("Not a valid position for the stack \n");
    return -1;
  }
  else {
     return sp->arr[arrayInd];
  }
}
```

→ //printing values from the stack.

```
for(int j=1; j<= sp->top+1 ; j++)
{ printf("The value at position %d is %d \n", j, peek(sp, j));
}
return 0;
}
```



Top - i + 1

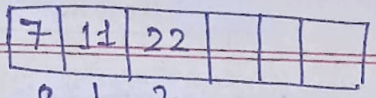
position (i)	Array Index
1	3 = (Top - i + 1)
2	2
3	1
4	0

3 - 1 + 1 = 3

```
int peek (struct stack *sp, int i){
  if(sp->top-i+1 < 0 ){
       printf("Not a valid position");
       return -1; }
  else{ return sp->arr[top-i+1];
  }
}
```

sp→

tion) lec-28 StackTop, StackBottom & Time Complexity of Operations in Stack Using Arrays:



Top

2	22 ← Top
1	11
0	7

StackTop

```
return sp->arr[sp->top]; → O(1)
```

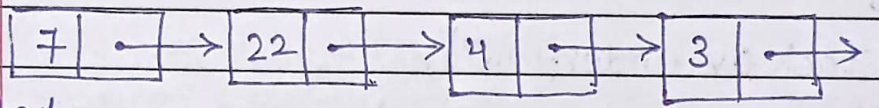
StackBottom

```
→ return sp->arr[0];
→ O(1)
```

isEmpty → O(1) push → O(1)
isFull → O(1) pop → O(1)
peek → O(1)

```
int stackTop(struct stack *sp){
  return sp->arr[sp->top];
}
```

lec-29. How to Implement Stack Using Linked Lists?



Side 1 — Side 2

O(1) → Insertion & deletion.

Side 1 will be used for push & pop Operations.

Head is Now → Top

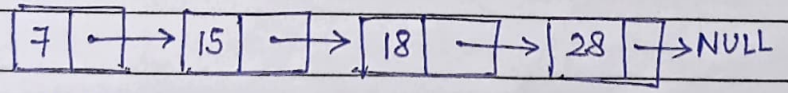
Stack Empty Condition → (Top == NULL) ✓ Amb

Stack Full when → Heap memory is exhausted ✓ [ptr == NULL]

You can always set a custom size.

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

lec-30. Implementing all the Stack Operations using linked list:



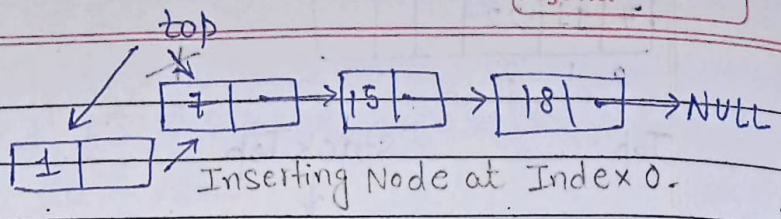
7
15
18
28

(1) → IsEmpty():

```
void isEmpty:
int if (top==NULL)
    { return 1; }
else { return 0;
```

(2) → isFull():

```
Struct Node *n = (Struct Node*) malloc (sizeof(struct Node));
if(n == NULL){
    return 1;
}
else {
    return 0;
}
```

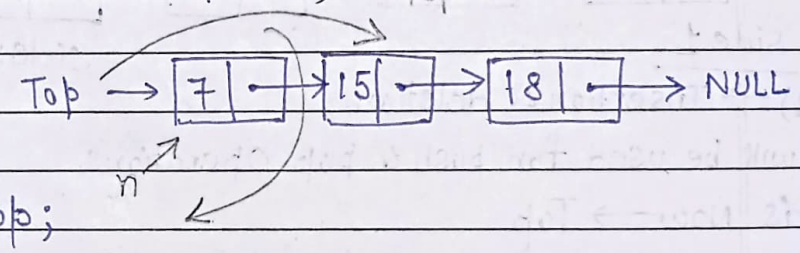


(3) → Push(x):

```
Struct Node *n = (struct Node*) malloc(sizeof(struct Node));
if(n==NULL){
printf("stack Overflow");
}.
else {
    n->data = x;
    n->next = top;
    top = n;
}
```

(4) Pop:

```
if (isEmpty){
    printf("stack ~~of~~ underflow");
}
else {
Struct Node *n = top;
top = top->next;
int x = n->data;
free (n);
return x;
}
```



```
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node *next;
};
void linkedListTraversal (struct Node *ptr) {
    while (ptr != NULL)
    {
        printf("Element: %d\n", ptr->data);
        ptr = ptr->next;
    }
}
int isEmpty (struct Node *top){
    if (top == NULL){
        return 1;
    }
    else {
        return 0;
    }
}
int isFull (struct Node *top) {
    struct Node *p = (struct Node*) malloc(sizeof (struct Node));
    if (p==NULL){
        return 1;
    }
    else {
        return 0;
    }
}
struct Node* push (struct Node* top, int x) {
    if (isFull (top)) {
        printf("Stack Overflow\n");
    }
    else {
        struct Node *n = (struct Node*) malloc (sizeof(struct Node));
        n->data = x;
        n->next = top;
        top = n;
        return top;
    }
```

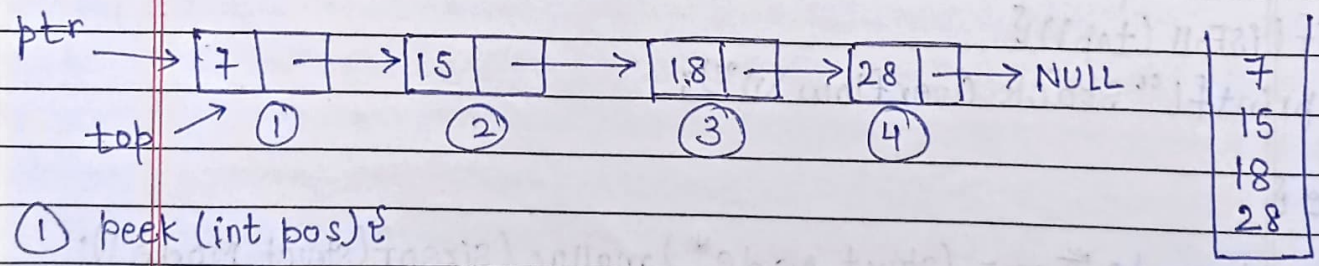
```
}
int pop(struct Node** top){
    if(isEmpty(*top)){
    printf("Stack underflow\n");
}
else{
    struct Node * n = *top;
    *top = (*top)->next;
    int x = n->data;
    free(n);
    return x;
}
}
int main(){
    struct Node* top = NULL;
    top = push(top, 78);
    top = push(top, 7);
    top = push(top, 8);

    linkedlistTraversal(top);
    int element = pop(&top);
    printf("Popped Element is: %d \n", element);
    linkedlistTraversal(top);
    return 0;
}
```

Output

```
Element is: 8
Element is: 7
Element is: 78
popped Element is: 8
Element is: 7
Element is: 78
```

Green House
Date
Page No.

Lec-31 peek(), StackTop() and other Operations on Stack using linked list:



① peek(int pos){

```
    struct Node *ptr = top;
    for(i=0; (i<pos-1 && ptr!=NULL); i++){
        ptr = ptr->next;
    }
```

position	n times move-ment
1	0
2	1
3	2

```
if(ptr != NULL)
    return ptr->data;
else
    return -1;
```

(2) stackTop

```
int stackTop(){
    return ptop->data;
}
```

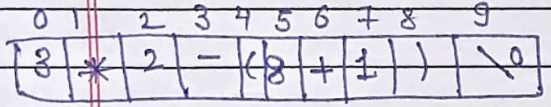
(3) stackBottom → N.W.

peek Operation:

```
int peek(int pos){
  Struct Node *ptr = top;
  for(int i=0; i<pos-1 && ptr!=NULL; i++)
  {
    ptr = ptr->next;
  }
  if(ptr!=NULL){
    return ptr->data;
  }
  else{
    return -1;
  }
}
```

→ printf("value at position 1 is: %d, peek(1));

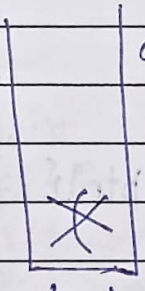
Lec-32 Parenthesis Matching problem using Stack Data Structure:



→ size n

Exp = 3 * 2 - (8 + 1)

① C? No → if yes push into the stack
②)? No → if yes pop out of stack



Cond^n

((3*2) - 1(8-2)) ✓

a=1-3)*4(8; → unbalanced ①

(1*(8-3(7) → unbalanced ②

(7*8)-3(7)) → unbalanced ①

Conditions for balanced Expressions:

Unbalanced Expression

① while popping stack should not underflow ⇒ if it happens
② At the End of Expression (EOE), the stack must be Empty
⇒ if not Empty → Unbalanced Expression.

Time Complexity of Parenthesis Matching
→ O(N) → O(1) in Best Case

Green House
Date
Page No.

Lec-33

Parenthesis Checking Using Stack in C Language:

3	*	2	-	(	8	+	1	)	\0
0	1	2	3	4	5	6	7	8	9

3*2-(8+1)

① (? → No → if yes push into the stack
②) ? → No → if yes pop out of the stack

Conditions:

→ while popping stack should not underflow.
→ At the End of Expression (EOE) stack must be Empty.

```
int ParenthesisMatch(char *exp){
    struct stack *sp;
    //create the stack
    for(i=0;exp[i]!='\0';i++){
        if(exp[i]=='('){
            push(sp,exp[i]);
        }
        elseif(exp[i]==')'){
            if(isEmpty(sp)){
                return 0;
            }
            pop(sp);
        }
    }
    if(isEmpty(sp)) return 1;
    else return 0;
```

```
#include<stdio.h>
#include<stdlib.h>
struct stack
{ int size;
  int top;
  char *arr;
};
int isEmpty(struct stack *ptr){
    if(ptr->top==-1)
    {
      return 1;
    }
  else{
      return 0;
    }
}
```

```
int isFull (struct stack *ptr)
{ if(ptr->top==ptr->size-1)
{ return 1;
  }
else {
    return 0;
  }
}

void push (struct stack *ptr, char val){
    if(isFull(ptr)){
     printf("Stack Overflow! Cannot push");
  }
else {  ptr->top++;
    ptr->arr[ptr->top]=val;
  }
}

char pop (struct stack* ptr){
    if(isEmpty(ptr)){
    printf("Stack O UnderFlow! Cannot pop");
    return -1;
} else {
      char val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}

int parenthesisMatch (char *exp){
    //create and initialize the stack
  struct stack *sp;
  sp->size = 100;
  sp->top = -1;
  sp->arr = (char*)malloc(sp->size * sizeof(char));
for(int i=0; exp[i]!='\0'; i++)
{
  if (exp[i]=='('){
      push(sp,'(');
```

```
    }
    else if (exp[i] == ')') {
        if (isEmpty(sp)) {
            return 0;
        }
        pop(sp);
    }
}
if (isEmpty(sp)) {
    return 1;
}
else { return 0;
}
}
int main()
{
    char *exp = "((8)(*--$$9))";
// check if stack is Empty
if (parenthesisMatch(exp)) {
    printf("The parenthesis is Matching");
}
else {
    printf("The parenthesis is Not Matching");
}
return 0;
}
```

Note:→ This program doesn't check the validity of Expression.

Green House
Date
Page No.

Output

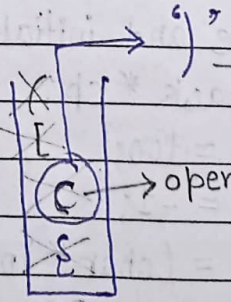
The parenthesis is Matching

Lec-34 Multiple parenthesis Matching using stack with C Code:

a = {7 - (3 - 2) + [8 + (99 - 11)]}

opening → { ([→ push

closing → })] → pop



check pop's Success

yes → keep checking

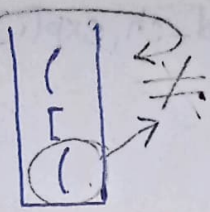
No → Not Balanced

is Stack Empty?

NO → Not Balanced

yes → Balanced.

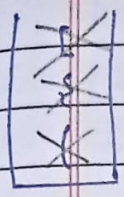
→ ([a*a-[3+2]])



≠ Not Balanced.

Green House
Date
Page No.

→ (7-11+{23-8*2}-[33+1])



→ stack is Empty

Expression → Balanced parenthesis

```
int match(char a, char b){
 if(a=='{' && b=='}'){
    return 1;
 }
 if(a=='(' && b==')'){
  return 1;
 }
 if(a=='[' && b==']'){
  return 1;
 }
 return 0;
}
int parenthesisMatch(char *exp){
  // Create and initialize the stack
  struct stack* sp;
   sp->size = 100;
   sp->top = -1;
   sp->arr = (char*)malloc(sp->size*sizeof(char));
   char popped_ch;
   for(int i=0; exp[i]!='\0'; i++){
   if[exp[i]=='(' || exp[i]=='{' || exp[i]=='['){
        push(sp, exp[i]);
   }
elseif( exp[i]==')' || exp[i]=='}' || exp[i]==']'){
      if(isEmpty(sp)){
        return 0;
   }
    popped_ch = pop(sp);
```

```
if (!match (popped_ch, exp[i])){
    return 0;
}
}
```

Green House
Date
Page No.

Lec-35 Infix, Prefix and Postfix Expressions:

→ Notation to write an Expression

Infix: → a+b → p/q → a-b → x-y

operand <operator> operand

Prefix:

<operator> <operand 1> <operand 2>

+ a b - p q

- x y * p b

Postfix:

<operand 1> <operand 2> <operator>

ab +

xy -

pq *

Infix	Prefix	Postfix
a*b	*ab	ab*
a-b	-ab	ab-

Infix: A*(B+C)*D

Postfix: ABC+*D

→ ③

① ((A * (B+C)) * D)

② ①

② ABC(+) * D*

A → 9 D → 2

B → 4

C → 4

Let B+C=8

A 8 * D*

→ 72D*

72 * D

Let A * 8 = 72

→

→ 144

Q1. Convert to prefix & postfix?

x-y*z

Que:1. Convert into prefix & Postfix ?

$x-y*z$

Green House
Date
Page No.

① Prefix:

step:1 → Parenthesize the expression

$(x-(y*z))$ → using Precendance & Associativity

$(x-[*yz])$ → $-[x][*yz]$

$-x*yz$ Done ☺

② Postfix:

step:1 → Parenthesize the expression

$(x-(y*z))$

$(x-[yz*])$ → $[x][yz*]-$

$xyz*-$

Que:2. $p-q-r/a$

① Prefix:

step:1 $((p-q)-(r/a))$ .... Parenthesize the Expression

→ $([-pq]-[/ra])$ → $-[-pq][/ra]$

$--pq/ra$ = Prefix !

② Postfix:

step:1 $((p-q)-(r/a))$

$([pq-]-[ra/])$ → $[pq-][ra/]-$

$pq-ra/-$ Done ☺

Que:3. $(m-n)*(p+q)$ → $[mn-][pq+]*$ → $*[-mn][+pq]$

$((m-n)*(p+q))$

Postfix: $([mn-]*[pq+])$

$mn-pq+*$

$((m-n)*(p+q))$

Prefix: $([-mn]*[+pq])$

$*-mn+pq$

→ postfix Equivalent of this Expression

Lec-36 Infix To Postfix Using stack:

Let, *, / → 2
+, − → 1

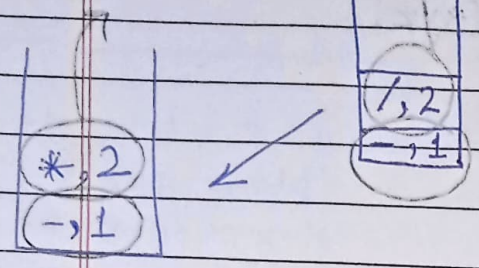
Green House
Date
Page No.

Infix

x-y/z-k*d

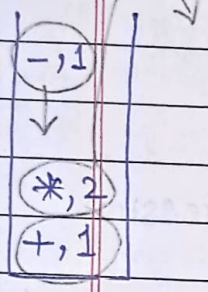
Postfix Expression:

xyz/-kd*-



Step 1: → ((x-(y/z))-(k*d))
((x-[yz/])-[kd*])
([xyz/-]-[kd*])
[xyz/-][kd*]-
xyz/-kd*-

∴ x+y*z-k



postfix Expression:

xyz*+k-

→ ((x+(y*z))-K)
→ ((x+[yz*])-k)
→ ([[xyz*]+]-k)
→ ([xyz*+]-k)
→ [xyz*+][k]-
→ xyz*+k-

Lec-37 Coding Infix to Postfix in C using Stack:

Infix to postfix using stack:

Infix → | a | - | b | * | d | + | c | \0 |
i ↑

a-b*d+c

postfix → | | | | | | | | |
j

```
char * IntoPo(char *infix){
    Struct stack *sp; // initializing the stack
    char *postfix = (char*) malloc(strlen(infix+1)*sizeof(char));
    int i=0; // Infix Scanner
    int j=0; // to fill postfix
    while(infix[i]!='\0'){
        if(!isoperator(infix[i])){
            postfix[j] = infix[i];
            i++; j++;
        }
```

= (struct stack*) malloc(sizeof (struct stack))

```
else {
    if (Prec(infix[i]) > (Pre (stackTop(sp)))){
        push (sp, infix[i]);
    }
    else {
        postfix[j] = pop(sp);
        j++;
    }
}

while ( ! isEmpty(sp)){
    postfix[j] = pop(sp);
    j++;
}

postfix[j] = '\0';
return postfix;
```

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

struct stack
{
    int size;
    int top;
char int *arr;
};

int stackTop(struct stack *sp){
    return sp->arr[sp->top];
}

int isEmpty (struct stack *ptr)
{
    if ( ptr->top == -1)
    {
        return 1;
    }
else {
    return 0;
    }
}
```

```
int isFull (struct stack *ptr) {
    if ( ptr->top == ptr->size-1) {
        return 1;
    }
    else {
        return 0;
    }
}

void push(struct stack * ptr, char val)
{
    if(isFull(ptr)) {
        printf("Stack OverFlow! ");
    }
    else {
        ptr->top++;
        ptr->arr[ptr->top] = val;
    }
}

Char pop(struct stack *ptr){
    if(isEmpty(ptr)){
        printf("Stack Underflow");
        return -1;
    }
    else {
        char val = ptr->arr[ptr->top];
        ptr->top--;
        return val;
    }
}
```

```
int precedence(char ch){
    if(ch=='*' || ch=='/')
        return 3;
    elseif (char=='+' || ch=='-')
        return 2;
    else
        return 0;
}

int isOperator(char ch){
    if(ch=='+' || ch=='-' || ch=='*' || ch=='/')
        return 1;
    else
        return 0;
}
char * infixtoPostfix(char *infix){
    struct stack* sp = (struct stack*) malloc(sizeof(struct stack));
    sp->size=10;
    sp->top=-1;
    sp->arr=(char*) malloc(sp->size* sizeof(char));
char* postfix = (char*) malloc(strlen(infix)+1)* sizeof(char));
    int i=0; // Track infix Traversal
    int j=0; // Track postfix addition
    while(infix[i]!='\0')
{
    if(!isOperator(infix[i])){
        postfix[j]= infix[i];
        j++;
        i++;
    }
    else {
        if(precedence(infix[i])>precedence(stackTop(sp))){
            push(sp,infix[i]);
            i++;
        }
    else{
        postfix[j]= pop(sp);
        j++;
    }
    }
}
```

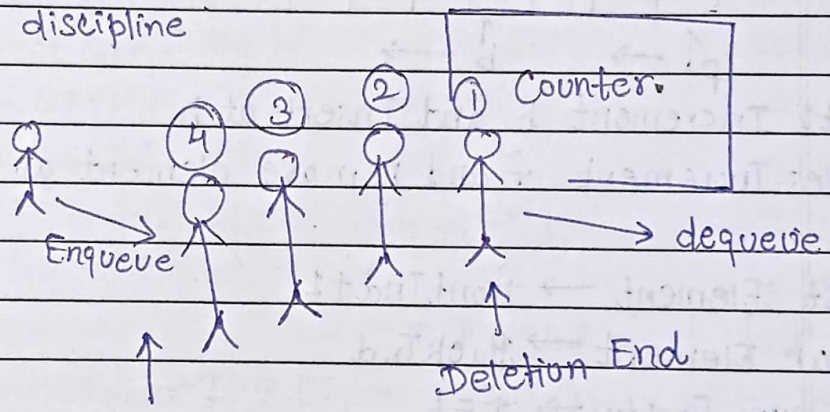
```
    while (!isEmpty(sp))
    {
        postfix[j]=pop(sp);
        j++;
    }
    postfix[j]='\0';
    return postfix;
}
int main()
{
    char * infix="x-y/z-k*d";
    printf("postfix is %s", infixtoPostfix(infix));
    return 0;
}
```

Output:

postfix is xyz/-kd*-

Lec-38 Queue Data Structure:

① Stack → LIFO discipline
② Queue → FIFO discipline



Queue ADT:

Data: ① Storage
② Ins End
③ Del End

Methods:
① Enqueue
② dequeue
③ first value
④ last value
⑤ peek(pos)
⑥ IsEmpty
⑦ IsFull

Note:

Queue can be Implemented in various ways!
→ Arrays
→ using Linked list
→ using other ADT's