

MODULE 2



STACKS :-

- * A stack is a linear list of elements in which an element may be inserted or deleted at only one end (ie; called as the top of the stack)
- * Stack is a data structure which works on the principle Last In First Out (LIFO) ie; the last element inserted will be the first element to be deleted.

* The Various Operations on stack are :

- ① PUSH - inserting an element into the stack.
- ② POP - removing/deleting an element from the stack.
- ③ Overflow - check whether stack is full or not.
- ④ Underflow - check whether stack is empty or not.

Abstract Datatype Stack Or ADT Stack :-

- * An ADT stack is the one that shows the various operations that can be performed on stack.
- * An ADT stack can be written as :

ADT stack is :

Objects: A set of zero or more elements

Functions: Parameters Used: STACK_SIZE \in Positive Integ,
S \in stack, item \in element

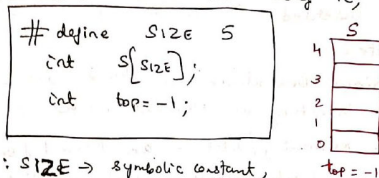
<u>Return type</u>	<u>Function name</u>	<u>operations</u>
① stack	Create(STACK_SIZE)	::= Creates an empty stack whose max. size is STACK_SIZE
② Boolean	IsFull(S, STACK_SIZE)	::= if no. of elements in S is STACK_SIZE return True else FALSE
③ Stack	Push(S, item)	::= if stack is not full, insert item onto stack.



- ④ Boolean $IsEmpty(s) ::=$ if no elements return True else False
- ⑤ Element $Pop(s) ::=$ if stack is empty return error condition as underflow else return the element on top of the stack.

Implementation of Stacks (using Arrays):-

- 1) Create(): The create stack for n can be implemented as a 1D array i.e;



Here: $SIZE \rightarrow$ symbolic constant, specifies the max. no. of elements that can be inserted into stack.

$S[SIZE]; \rightarrow$ an array, to hold the elements of the stack.

- ② IsFull() (check for overflow):-

* In order to perform a push operation we need to check whether the stack is completely filled with elements or not.

i.e;

```

if (top == SIZE - 1)
{
    printf("\n Stack Full - Overflow");
    return;
}

```



③ Push() :-

- * Inserts an element onto the stack.
- * First checks whether sufficient space is available in the stack. If available then increments the value of top by 1, and then inserts the item into stack. i.e;

```
void push()
{
    int item;
    if (top == SIZE - 1)
    {
        printf("Stack Overflow\n");
        return;
    }
    printf("Enter an item\n");
    scanf("%d", &item);
    S[++top] = item;
}
```



- ### ④ IsEmpty() :-
- Before we delete an item from the stack, we must first check whether there are any elements in the stack. i.e;

```
if (top == -1)
{
    printf("Stack is Empty - Underflow\n");
    return;
}
```

∴ top = -1 indicates that the stack is empty.

- ### ⑤ Pop() :-
- To delete an item from the stack.

P.T.O

```

void pop()
{
    if (top == -1)
    {
        printf("Stack Underflow\n");
        return;
    }
    printf("Deleted item = %d", s[top--]);
}

```

5. Display(): Print the contents of the stack

```

void display()
{
    int i;
    if (top == -1)
    {
        printf("Stack is Empty-Underflow\n");
        return;
    }
    printf("Contents of the stack are:\n");
    for (i = top; i >= 0; i--)
        printf("%d\n", s[i]);
}

```

⇒ Write a program to implement stacks using arrays:

```

#include <stdio.h>
#define SIZE 5
int s[SIZE], top = -1;

```



```

void push();
void pop();
void display();

int main()
{
    int ch;
    for(;;) → infinite loop
    {
        printf("1. PUSH 2. POP 3. Display 4. EXIT\n");
        printf("Enter your choice: \n");
        scanf("%d", &ch);
        switch(ch)
        {
            case 1: push(); break;
            case 2: pop(); break;
            case 3: display(); break;
            default: printf("Invalid"); exit(0);
        }
    }
    return 0;
}

```

```

void push()
{
    ___
    ___
    ___
}

void pop()
{
    ___
    ___
    ___
}

```

```

void display()
{
    ___
    ___
    ___
}

```

Stack implementation Using structures :-

```
#define SIZE 5
```

```
struct stack
```

```
{
```

```
    int a[SIZE];
```

```
    int top;
```

```
};
```

```
struct stack S;
```

```
S.top = -1;
```

```
void main() {
```

```
{
```

```
    // same as previous program
```

```
}
```

```
void push()
```

```
{
```

```
    int item;
```

```
    if (S.top == SIZE - 1)
```

```
    {
```

```
    }
```

```
    printf("Enter Item:");
```

```
    scanf("%d", &item);
```

```
    S.top++;
```

```
    S.a[S.top] = item;
```

```
}
```

```
void pop()
```

```
{
```

```
    if (S.top == -1)
```

```
    {
```

```
    }
```

```
    printf("Item deleted: %d", S.a[S.top]);
```

```
    S.top--;
```

```
}
```

```
void display()
```

```
{
```

```
    if (S.top == -1)
```

```
    {
```

```
    }
```

```
    printf("Stack contents:");
```

```
    for (i = S.top; i >= 0; i--)
```

```
        printf("%d\n", S.a[i]);
```

```
}
```

Stacks Using Dynamic Arrays :-

* we can allocate or create a stack dynamically during run time - when we are not sure about the size of the stack, as follows:

* assume that the stack size is 1 ie;

```
int stackSize = 1;
int *s;
int top = -1;
s = (int*) malloc (stackSize * sizeof (int));
void Push ()
{
    if (top == StackSize - 1)
    {
        printf ("Stack Full : Increase by 1\n");
        stackSize++;
        s = (int*) realloc (s, stackSize * sizeof (int));
    }
    s[++top] = item;
}

void Pop ()
{
    if (top == -1)
    {
        printf ("Stack underflow");
        return;
    }
    printf ("Item deleted = %d\n", s[top--]);
    printf ("Stack size decreased by 1\n");
    stackSize--;
    s = (int*) realloc (s, stackSize * sizeof (int));
}

int main ()
{
    //
    //
}
```

where *s \Rightarrow is a pointer which holds the memory locations, using malloc, realloc fn

Applications Of Stacks :-

1) Used in Function calls:

Ex: void main()

```
{
    Pf("CS");
    f();
    Pf("Job.");
}

void f1()
{
    Pf("Students");
    f2();
    Pf("Marks and");
}
```

void f2()

```
{
    Pf("Get Good");
}
```

f2
f1
main

2) Used in conversion of expressions i.e; arithmetic expressions.

3) Evaluation of Expressions: An arithmetic expression is represented in the form of either postfix or Prefix.

4) Used in Recursion.

Applications:

Polish Notation:

* An arithmetic expression can be of 3 types:

- Infix Expression - placing the operator b/w the operands i.e; a+b
- Prefix Expression - placing the operator before the operands eg: +ab
(Polish Notation)
- Postfix Expression - placing the operator after the operands i.e; ab+
(Reverse Polish Notation)

The computer usually evaluates an arithmetic expression in 2 steps:

- a) First, it converts the expression (infix) into a postfix notation.
- b) Second, it evaluates the postfix expression.

c) Conversion / Transform Infix expr~~s~~ into Postfix Expressions :-

> Algorithm :-

step 1: Push # onto the stack.

step 2: Scan the expression from left to right & repeat Step 3 to 6 for each element.

step 3: If a '(' parenthesis is encountered, push it onto stack.

step 4: If an operand is encountered, add it into Postfix P.

step 5: If an operator is encountered, then:

a) check whether the precedence of the current operator on top of the stack is greater than or equal to the precedence of the scanned operator, then go on popping all the operators from the stack & place them in the postfix P.

b) otherwise (else) push the scanned operator onto stack

step 6: If an ')' right parenthesis is encountered, then go on popping all the items from the stack & place them in postfix expression; till we get the matching left '(' parenthesis.

note: Do not push the '(' left parenthesis onto postfix P.

Step 7: Check if any operator or symbols are present in the stack. If so, then pop them and place it into postfix P.

⇒ Write a C program to convert infix into postfix expressions. (Refer Lab Program 4).

Example: Convert the fully expressions into postfix expressions using the above algorithm.

① $(a * b) + c$
 0 1 2 3 4 5 6

Initial: top = -1 and pos = 0

Step	Sym	Postfix P	Stack	Variables (optional)
Initial	-	-	#	top = 0, pos = 0
i = 0	(-	#, (top = 1, pos = 0
1	a	a	#, (top = 1, pos = 1
2	*	a	#, (, *	top = 2, pos = 1
3	b	ab	#, (, *	top = 2, pos = 2
4)	ab*	#,	top = 0, pos = 3
5	+	ab*	#, +	top = 1, pos = 3
6	c	ab*c	#, +	top = 1, pos = 4

⇒ Place the remaining operators i.e. + into the postfix: ∴

$ab * c +$ ⇒ postfix expression

② $(A - B) * (D / E)$
 0 1 2 3 4 5 6 7 8 9 10

Step	Sym	Postfix	Stack	Variables
Initial	-	-	#	top = 0, pos = 0
i = 0	(#, (top = 1, pos = 0
1	A	A	#, (top = 1, pos = 1
2	-	A	#, (, -	top = 2, pos = 1
3	B	AB	#, (, -	top = 2, pos = 2
4)	AB-	#	top = 0, pos = 3
5	*	AB-	#, *	top = 1, pos = 3
6	(AB-	#, *, (top = 2, pos = 3
7	D	AB-D	#, *, (top = 2, pos = 4
8	/	AB-D	#, *, (, /	top = 3, pos = 4

9	E	AB-DE	#, *, (, /	top: 3, pos: 5
10)	AB-DE/	#, *	top: 1, pos: 6

\therefore Postfix = AB-DE/*

Note: Precedence of arithmetic operators.

- ^ \Rightarrow P = 4
- %, /, * \Rightarrow P = 3
- +, - \Rightarrow P = 2
- (,) \Rightarrow P = 1
- # \Rightarrow P = 0

Problems on postfix Expressions :-

$$\begin{aligned}
 \text{i) } & 12, 7, 3, -, /, 2, 1, 5, +, *, + \\
 \Rightarrow & 12, (7-3), /, 2, 1, 5, +, *, + \\
 = & 12 / (7-3), 2, 1, 5, +, *, + \\
 = & 12 / (7-3), 2, (1+5), *, + \\
 = & 12 / (7-3), 2, (1+5), *, + \\
 = & 12 / (7-3), (2 * (1+5)), + \\
 = & 12 / 4 + [2 * 6] \\
 = & 3 + 12 \\
 = & 15
 \end{aligned}$$

Convert the infix expression into postfix :-

$$\begin{aligned}
 \text{(i) } & a + b * c / d \\
 = & a + [bc *] / d \\
 = & a + [bc * d /] \\
 = & abc * d / +
 \end{aligned}$$

$$\begin{aligned}
 \text{(ii) } & (A-B) * (D/E) \\
 = & (AB-) * (DE/) \\
 = & AB-DE/+
 \end{aligned}$$

$$\begin{aligned}
 \text{(iii)} \quad & (A + B \wedge D) / (E - F) + G \\
 & = (A + BD \wedge) / (EF -) + G \\
 & = (ABD \wedge +) / (EF -) + G \\
 & = (ABD \wedge + EF - /) + G \\
 & = ABD \wedge + EF - / G +
 \end{aligned}$$

$$\begin{aligned}
 \text{(iv)} \quad & A * (B + D) / E - F * (G + H) / K \\
 & = A * (BD +) / E - F * (G + HK /) \\
 & = (ABD + *) / E - F * (G HK / +) \\
 & = (ABD + * E /) - F * (G HK / +) \\
 & = (ABD + * E /) - (F G HK / + *) \\
 & = ABD + * E / F G HK / + * -
 \end{aligned}$$

II.) Evaluation Of Postfix Expressions :-

Algorithm:

Step 1: Scan the given expressions from left to right.

Step 2 (a): If the symbol is an operand then push its value onto the stack.

(b): If the symbol is an operator, then pop out two values from the stack & assigns them respectively to opnd1 and opnd2.

Then perform the required operator i.e;

$$\text{res} = \text{opnd1} * \text{opnd2}$$

Push the result back to the stack.

Step 3: Repeat Step 2 until all the symbols are over.

Step 4: Pop out the result and print it.

Example:- trace or Evaluate the following postfix expression.

1) $78+65+*$

Step	Sym	Opnd1	Opnd2	Stack
i=0	7	-	-	7
1	8	-	-	7,8
2	+	7	8	15
3	6	-	-	15,6
4	5	-	-	15,6,5
5	+	6	5	15,11
6	*	15	11	165

\therefore Result = 165 //

2) $5,6,2,+,*,12,4,/, -$

	Sym	Opnd1	Opnd2	Stack
i=0	5	-	-	5
1	6	-	-	5,6
2	2	-	-	5,6,2
3	+	6	2	5,8
4	*	5	8	40
5	12	-	-	40,12
6	4	-	-	40,12,4
7	/	12	4	40,3
8	-	40	3	37

\therefore Result = 37 //

Recursion :- It is a programming technique in which the function calls by itself.

Types of Recursion: (1) Direct Recursion (2) Indirect Recursion

P.T.O

1) Direct Recursion

```
void main()
{
    pf("Happy Dazna");
    main();
}
```

2) Indirect Recursion

```
void main()
{
    pf("Happy Dazna");
    d1();
}
d1()
{
    main();
}
```

```
main()
{
    int
    pas
    s;
}
output:
3
```

Requirements of Recursion :-

- ① Base criteria / Terminating condition - for which the procedure does not call itself.
- ② Each time the procedure does call itself, it must be closer to be base criteria.
- ③ stack.

Recursion can be implemented for the following techniques;

1) Factorial of a Number :-

Denoted by $n! = 1 \cdot 2 \cdot 3 \dots (n-2)(n-1) \cdot n$

Factorial function can be defined as:

- a) If $n=0$, then $n! = 1$
- b) If $n > 0$, then $n! = n \cdot (n-1)!$

Program:

```
int fact(int n)
{
    if (n==0)
        return 1;
    else
        return (n * fact(n-1));
}
```

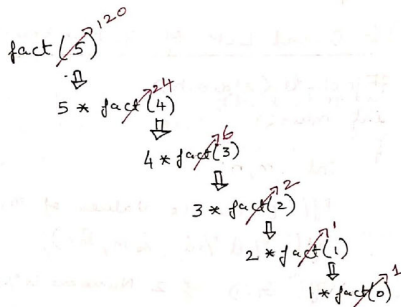
2) Fi
The
ie;
Defi
a)
b)
pa

```

main()
{
    int n;
    printf("Enter a number \n");
    scanf("%d", &n);
    printf("Factorial = %d", fact(n));
}

```

output:



2.) Fibonacci Sequence :-

The fibonacci sequence is as follows:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

ie; $F_0 = 0$ and $F_1 = 1$, and each succeeding term is the sum of the two preceding terms.

Definition: (Fibonacci)

a) If $n = 0$ or $n = 1$, then $F_n = n$

b) If $n > 1$, then $F_n = F_{n-2} + F_{n-1}$

Program int fib(int n)

```

{
    if (n == 0) return 0;
    if (n == 1) return 1;
    else
        return (fib(n-1) + fib(n-2));
}

```



```

main()
{
    int n;
    printf("Enter the value of n");
    scanf("%d", &n);
    printf("Fibonacci = %d", fib(n));
}

```

3.) GCD and LCM of 2 Numbers:-

```

#include <stdio.h>
int gcd(int, int);
int main()
{
    int m, n;
    printf("Enter the values of m, n\n");
    scanf("%d %d", &m, &n);
    printf("GCD of 2 Numbers is %d", gcd(m, n));
    return 0;
}

```

```

int gcd(int a, int b)
{
    if(a == 0) return b;
    if(b == 0) return a;
    if(a > b)
        gcd(a % b, b);
    else
        gcd(a, b % a);
}

```

O/P:
 $\text{gcd}(15, 45)$
 \downarrow
 $\text{gcd}(15, 45 \% 15)$
 $\text{O/P} = 15$

Tower Of Hanoi :- Recursion may be used as a tool in solving the problem of Tower of Hanoi.

Problem statement : Consider 3 poles (pegs) A, B and

C. There are 'n' disks on pole A of different parameters and are placed one above the other, such that always smaller disk is placed above the larger disk.

Initially pole B and C are empty.

rules: ① Only one disk is moved at a time, i.e; move only the top disk on any peg.

② Smaller disk is on top of the larger disk at any time.

③ A disk can be moved from one pole to another only.

Solution :

1) If there is only one disk, i.e; $n=1$, then move that disk from A to C. ($A \rightarrow C$)

2) If there are two disks i.e; $n=2$, then move one disk from A to B, 2nd from A to C and then from B to C.

i.e; $A \rightarrow B$

$A \rightarrow C$

$B \rightarrow C$

3) In general to move 'n' disks from A to C, the following recursive technique is used:

a) Move the top $(n-1)$ disks from A to B. [$A \rightarrow B$]

b) Move the n^{th} disk from A to C. [$A \rightarrow C$]

c) Move the $(n-1)$ disks from B to C. [$B \rightarrow C$]

Example : For $n=3$; the solution consists of the following 7 moves:

- (i) Move top disk from A to C
- (ii) Move top disk from A to B
- (iii) Move top disk from C to B
- (iv) Move top disk from A to C
- (v) " " " " B to A
- (vi) " " " " B to C
- (vii) " " " " A to C

Algorithm for Tower of Hanoi
 TOWER (n, BEG, AUX, END)

step 1: If $n = 1$, then

- a) write: BEG \rightarrow END
- b) Return

step 2: [Move $n-1$ disks from peg BEG to peg AUX]
 call TOWER($n-1$, BEG, END, AUX)

step 3: write BEG \rightarrow END

step 4: [Move $n-1$ disks from peg AUX to peg END]
 call TOWER($n-1$, AUX, BEG, END)

step 5: Return.

Program to implement Tower of Hanoi:

Refer Lab. program 5.(b)

5) Ackermann Function :-

* The Ackermann function is a function with 2 argument each of which can be assigned any non-negative integer: 0, 1, 2, ... This function is defined as;

a) If $m = 0$; then $A(m, n) = n + 1$

b) If $m \neq 0$
 $A(m, n)$
 \rightarrow observe that
 $m = 0$. The 1
 (0,)
 * The acker
 etc impos

Tracing for

tower(3, a, b)

QUEUE:

* Queue works
 ie; e
 large
 * How diff
 re
 f.

Adv:

So.
 Fl
 OS

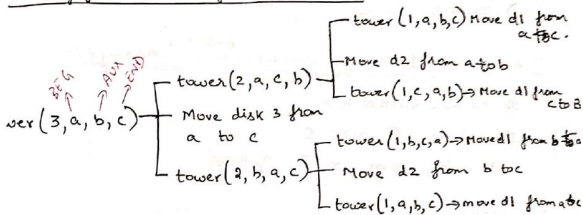
b) If $m \neq 0$ and $n \neq 0$, then

$$A(m, n) = A(m-1, A(m, n-1))$$

observe that $A(m, n)$ is explicitly given only when $m=0$. The base criteria are the pairs: $(0,0)$, $(0,1)$, $(0,2)$...

The Ackermann function A is too complex to evaluate and its importance comes from its use in mathematical logic.

Tracing for Tower of Hanoi: $n=3$



QUEUES :-

Queue is a linear list data structure which works on the principle FIFO (First In First Out) i.e., element that is present in the queue for the longest time will get deleted first.

Here insertion and deletion takes place at different ends: rear and front.

rear: insertion into the queue takes place at the rear (back) end.

front: deletion from the queue takes place at the front end.

Advantages:- Used in application and system software development. Eg: CPU process the jobs in FIFO order, Printer prints the file in FIFO, OS allocates the memory for the objects in FIFO order.

Types of Queues :-

- 1.) Ordinary Queues (Linear)
- 2.) Double ended Queues
- 3.) Circular Queues
- 4.) Priority Queues.

Implementation of Queues :-

- 1.) static Memory allocation - arrays, structures
- 2.) DMA - linked list.

Implementation of Queues ^(ordinary) using arrays :-

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

```
#define Q_SIZE 5
```

```
int ch, item, r, f, q[Q_SIZE];
```

1) Insert() :-

Algorithm : Step 1: Before inserting item, first check for overflow i.e;

```
if (r == Q_SIZE - 1)
```

Step 2: Accept item

3: Increment r by 1 i.e; r=r+1

4.) Assign q[r] = item

5.) end

4) Function

```
void insertQueue()
```

```
{
```

```
int item, r = -1;
```

```
if (r == Q_SIZE - 1)
```

```
{
```

```
printf("Q overflow");
```

```
return;
```

```
}
```

```
printf("Enter the item:\n");
```

```
scanf("%d", &item);
```

```
q[t+r] = item;
```

```
}
```

```
) delete():
```

Algorithm: Step 1: Check for underflow i.e; if $(f > r)$

Step 2: Delete item i.e; $q[f]$

Step 3. Increment f by 1 : $f = f + 1$

Step 4. End

function

```
void deletefront()
```

```
{
```

```
    f = 0;
```

```
    if (f > r) (or) if ((f == 0) && (r == -1))
```

```
    { printf("Q is empty"); return;
```

```
    }
```

```
    printf("Element deleted = %d\n", q[f++]);
```

```
}
```

```
) display():
```

```
void display()
```

```
{
```

```
    int i;
```

```
    if (f > r) (or) if ((f == 0) && (r == -1))
```

```
    { printf("Q is Empty"); return;
```

```
    }
```

```
    printf("Contents of Queue are:\n");
```

```
    for (i = f; i <= r; i++)
```

```
        printf("%d\n", q[i]);
```

```
}
```

using arrays:

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

```
#define B_SIZE 5
```

```
int ch, r, f, i, item, arr[B_SIZE];
```

```
void insertrear();
```

```
void deletefront();
```

```
void display();
```

```
void main()
```

```
{
```

```
    f = 0; r = -1;
```

```
    for (;;) {
```

```
        printf("1. INSERT 2. DELETE 3. DISPLAY 4. EXIT\n");
```

```
        printf("Enter your choice\n");
```

```
        scanf("%d", &ch);
```

```
        switch(ch)
```

```
        {
```

```
            case 1: insertrear(); break;
```

```
            case 2: deletefront(); break;
```

```
            case 3: display(); break;
```

```
            case 4: exit(0); break;
```

```
            default: printf("Invalid choice\n");
```

```
        }
```

```
    }
```

```
}
```

```
void insertrear()
```

```
{
```

```
    _____
```

```
    _____
```

```
}
```

```
void deletefront()
```

```
{
```

```
    _____
```

```
    _____
```

```
}
```

```
void display()
```

```
{
```

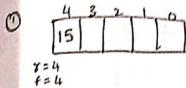
```
    _____
```

```
    _____
```

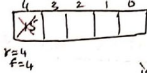
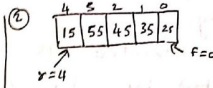
```
}
```

Advantages of OQ:- Simple and easy to implement.

Disadvantages:



Here insertion is not possible even though enough empty spaces are available



After deleting 15, f=5, r=4 now insertion is not possible even though entire Q is empty

Solution: Reset $f=0$, $r=-1$, when $f > r$

ADT (Abstract Data Type) Queues

Objects: a finite ordered list with zero or more elements.

Functions:

- ① Queue Create(Q, size) ::= Create an empty queue whose max size is Q-size
- ② Boolean IsFull(Q, queue, Q-size) ::= if (no. of ele == Q-size) return TRUE else FALSE
- ③ Queue AddQ(queue, item) ::= if (IsFull(queue)) queue full else insert item at rear end.
- ④ Boolean IsEmpty(Q, queue) ::= if (queue == create(Q-size)) return TRUE else FALSE.
- ⑤ Element DeleteQ(queue) ::= if (IsEmpty(Q, queue)) return true else remove item at front end.

CIRCULAR QUEUES:- (CQ)

- * It is one of the best data structures which uses FIFO. i.e; elements are stored efficiently in an array so as to wrap around - so that end of the queue is followed by the front of the queue.
- * CQ is used in application & system S/W development. It overcomes the drawback of OQ. In OQ, if $r == size-1$; insertion is not possible

whereas in CQ, we can insert the data in circular fashion. \therefore the memory is used optimally.

* Before insertion the value of r is calculated by: $r = (r+1) \% \text{SIZE}$

* After deletion the value of f is calculated by $f = (f+1) \% \text{SIZE}$

* To check for overflow & under flow, global variable count is maintained.

If $\text{Count} == \text{SIZE} \rightarrow$ then Q is Full.

If $\text{Count} == 0$, \rightarrow Q is empty.

* The operations of CQ are:

- insert
- delete
- display.

Implementation of Circular Queues (using array)

```
#define SIZE 5
```

```
int q[SIZE], f=0, r=-1, count=0;
```

```
void insertq()
```

```
{
```

```
    int item;
```

```
    if (count == SIZE)
```

```
    { printf("Q is FULL"); return;
```

```
    }
```

```
    r = (r+1) % SIZE;
```

```
    printf("Enter the item: \n");
```

```
    scanf("%d", &item);
```

```
    q[r] = item;
```

```
    count++;
```

```
}
```

```

void delete(Q q)
{
    if (count == 0)
    {
        Pf("Q is empty");
        return;
    }
    Pf("Element deleted = %d\n", q[f]);
    F = (F+1) % SIZE;
    count--;
}

void display()
{
    int i, j;
    if (count == 0)
    {
        Pf("Q is empty"); return;
    }
    j = F;
    Pf("Contents of Q are:");
    for (i = 1; i <= count; i++)
    {
        Pf("%d\n", q[j]); j = (j+1) % SIZE;
    }
}

```

Advantages: optimal utilization of memory.

Disadvantage: ① May become infinite loop.
 ② Keeping track of r and f is difficult.

Circular Queues using Dynamic Arrays:-

* The various operations that can be performed on CQ using dynamic arrays are:

① create(): Let us assume Q-SIZE is 1
 int QSIZE = 1;
 int *q, F = 0, r = -1, count = 0;

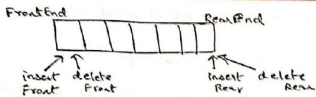
② Insert(): Before inserting, check for sufficient space in the queue. But once the queue is full, we can increase the size of queue using `realloc()`

```
void insertQ()
{
    if (count == QSIZE)
    {
        printf("Q Full - Increase size by 1");
        QSIZE++;
        q = (int*) realloc(QSIZE, sizeof(int));
    }
    if (f > r)
    {
        for (i = QSIZE - 2; i >= f; i--)
        {
            q[i+1] = q[i];
        }
        f++;
    }
    r = (r+1) % QSIZE;
    q[r] = item;
    count++;
}
```

* `Delete()` and `display()` - are same as in CB arrays.

DEQUEUES (Double-ended queues/deck/deques)

- * A deque is a linear list in which elements can be added or removed at either end.
- * Insertion is possible at both front and rear ends. Similarly deletion can be done at front and rear end.



- * The Operations Performed are:
- InsertFront()
 - InsertRear()
 - DeleteFront()
 - DeleteRear()
 - display.

Implementation of Double-ended Queue: Using Arrays

```
#define SIZE 5
int q[SIZE];
int f=0, r=-1;
```

- a) InsertFront(): insertion of element at the front end.

```
void insertFront()
```

```
{
    if (f==0 && r==-1) // q is empty then insert item
```

```
{
    q[++r] = item;
    return;
```



Insert item=10
after ++r

```
}
    if (f != 0)
```

// Insert when items are Present

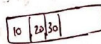
```
{
    q[--f] = item;
    return;
```



insert new item
before the value 10

```
}
```

```
printf("Front Insertion Not Possible");
```



insertion not possible as
an item already exists at front

```
}
```

b) Insertrear(): Insertion of element at rear end

// Refer Ordinary Queue for insertrear

c) deletefront(): deleting an item at the front

// Refer Ordinary Queue for deletefront

d) deleterear(): deleting an item at rear end.

```
void deleterear()
```

```
{ if (f > r)
```

```
{ Pj("Queue Underflow");
```

```
f = 0, r = -1; // reset Queue  
return;
```

```
}
```

```
Pj("Element deleted = %d\n", q[r--]);
```

```
}
```

e) display()

```
void display()
```

```
{ int i;
```

```
if (f > r)
```

```
{ Pj("Q is Empty - Underflow");
```

```
f = 0, r = -1;
```

```
return;
```

```
}
```

```
Pj("Contents of Queues are :");
```

```
for (i = f; i <= r; i++)
```

```
Pj("%d\n", q[i]);
```

```
}
```

main() function

```
void main()
```

```
{ int ch;
```

```
for(;;)
```

```
{
```

```
Pj("1. I
```

```
4. 1
```

```
Pj("En
```

```
of (" %
```

```
Switch
```

```
{
```

```
case
```

```
case
```

```
case
```

```
case
```

```
case
```

```
case
```

```
case
```

```
case
```

```
case
```

```
case
```

```
case
```

```
case
```

```
case
```

```
case
```

PRIORITY

* The pri

in wh

on pr

* Always

Proc

* If the

then

the

the

the

the

the

the

```

main() function for DQ
void main()
{
    int ch;
    for(;;)
    {
        Pf("1. INSERTFRONT 2. INSERTREAR 3. DELETEFRONT
          4. DELETREAR 5. DISPLAY 6. EXIT \n");
        Pf("Enter your choice:");
        sf("%d", &ch);
        switch (ch)
        {
            case 1: Pf("Enter Item\n"); sf("%d", &item);
                    InsertFront(); break;
            case 2: Pf("Enter Item\n"); sf("%d", &item);
                    InsertRear(); break;
            case 3: deleteFront(); break;
            case 4: deleteRear(); break;
            case 5: display(); break;
            default: exit(0);
        }
    }
}

```

PRIORITY QUEUES :-

- * The priority queue is a special type of data structure in which items can be deleted or inserted based on priority.
- * Always an element having highest priority is processed first.
- * If the elements in the queue are of the same priority, then the element which is inserted first into the queue is processed.

Applications :-

- ① Used in Operating Systems for the design of Job scheduling Algorithms.

Priority Queues are classified into:

- a) Ascending Priority Queue - Here elements are inserted in any order. But while deleting an item from the queue, only the smallest element is removed first.
- b) Descending Priority Queue - Here also elements are inserted in any order. But while deleting an element, only the largest element is deleted first.

Implementation of Priority Queue

Design 1: using ascending Priority queue - where the smallest element is removed first. Here the elements are added at the rear end.

Design 2: The 2nd technique is to insert the items based on the priority. Here we assume the item to be inserted itself denotes the priority. So the items with least value can be considered as the items with highest priority and elements with highest ~~priority~~ value can be considered as the items with least priority. So, we insert the elements into queue in such a way that they are always ordered in increasing order. Hence highest priority of elements are at the front end hence delete elements at the front end.

Priority Queue

void insert()

```
{
    int f;
    if (r == SIZE - 1)
    {
        pf("Q is Full"); return;
    }
    j = r;
    while (j >= 0 && item < q[j]) // Find appo posito
                                   // for the new item
    {
        q[j+1] = q[j];
        j--;
    }
    q[j+1] = item;
    r = r+1;
}
```

Complete Program

```
#include <stdio.h>
#define SIZE 5
int q[SIZE], f=0, r=-1, item, ch;

void insert()
{
    _____
    _____
}

void deletefront()
{
    _____
    // Same as Ordinary Queue's - delete()
    _____
}
```



```

void display()
{
    // same as OQ's - display
}

main()
{
    dox(;;)
    {
        Pj("1. INSERT 2. DELETE ----- \n");
        Pj("Enter UR choice");
        Pj("%d", &ch);
        switch (ch)
        {
            case 1: Pj("Enter Item: \n");
                    Pj("%d", &item);
                    insert();
                    break;
            case 2: delete(); break;
            case 3: display(); break;
            default: exit(0);
        }
    }
}

```

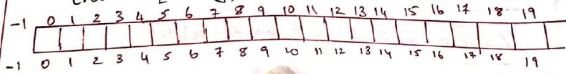
Multiple Stacks and Queues :-

Let us, implement Multiple stacks using arrays :-

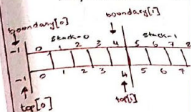
```

#define SIZE 20
int s[SIZE];

```



* Now let us divide
 Eg: If $n=4$, then
 represented as

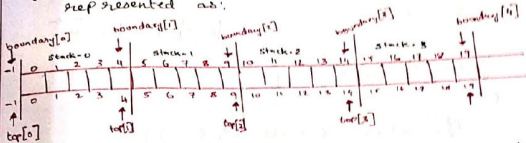


* To find the :
 top[0], top[1]
 to
 ie; $j=0$
 $j=1$
 $j=$
 $j=$
 \therefore , we
 ?

* To find
 Value
 ie;

* Now let us divide the array into 'n' stacks.

Eg: If $n=4$, then 4 empty stacks can be pictorially represented as:



* To find the initial value of top of each stack, i.e; $top[0]$, $top[1]$, $top[2]$, $top[3]$, use the expression:

$$top[j] = size/n * j - 1 \quad \text{where } j=0 \text{ to } n$$

$$\text{i.e; } j=0; \quad top[0] = 20/4 * 0 - 1 = -1$$

$$j=1; \quad top[1] = 20/4 * 1 - 1 = 4$$

$$j=2; \quad top[2] = 20/4 * 2 - 1 = 9$$

$$j=3; \quad top[3] = 20/4 * 3 - 1 = 14$$

\therefore , we can write as follows:

for ($j=0; j \leq n; j++$)

$$\{ \quad top[j] = size/n * j - 1;$$

}

* To find the boundary, of each stack, copy initial value of $top[0]$ to $boundary[0]$, and so on,

i.e; for ($j=0; j \leq n; j++$)

$$\{ \quad boundary[j] = top[j];$$

}

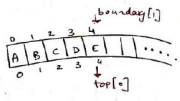
$$OR \quad boundary[j] = top[j] = size/n * j - 1;$$

*Push(): First check for overflow then insert item.

```

void push()
{
    if (top[i] == boundary[i+1])
    {
        printf("Stack is full in %d", i);
        return;
    }
    s[++top[i]] = item;
}

```

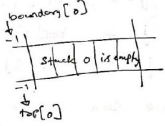


*Pop(): First check for underflow, then try to delete.

```

void pop()
{
    if (top[i] == boundary[i])
    {
        printf("Stack is empty", i);
        return;
    }
    printf("Item deleted = %d", s[top[i]--]);
}

```



*display()

```

void display()
{
    if (top[i] == boundary[i])
}

```

```

{
    printf("Stack is full")
}
}

```

```

complete pr
#include <stdio.h>
#include <string.h>
#define size 10
int s[size];
int top;
int n;

```

```

void push()
{
    // ...
}
void pop()
{
    // ...
}

```

day[1]

.....

```

{
    printf("Stack %d is empty", i)
    return;
}
for(j = boundary[i] + 1; j <= top[i]; j++)
    printf("%d\n", s[j]);
}

```

Complete Program

```

#include <stdio.h>
#include <stdlib.h>
#define SIZE 20
int s[SIZE];
#define MAX_STACKS 10
int boundary[MAX_STACKS];
int top[MAX_STACKS];
int n, i, j, item, ch;

```

```

void main()
{
    printf("Enter No. of Queues\n");
    scanf("%d", &n);
    for(j = 0; j <= n; j++)
        boundary[j] = top[j] = size/n;
    for(i = 0; i < n; i++)
    {
        printf("Stack Number: ");
        for(j = 0; j < n; j++)
            printf("%d ", j);
    }
}

```

```

void push()
{
    _____
    _____
    _____
}

```

```

void pop()
{
    _____
    _____
    _____
}

```

```

void display()
{
    _____
}

```

```

printf("Enter stack No. to perform operation");
scanf("%d", &i);
printf("1: Push 2: Pop 3: display 4: Exit\n");
printf("Enter your choice\n");
scanf("%d", &ch);
switch(ch)
{
    case 1: printf("Enter item"); scanf("%d", &item);
            push(); break;
    case 2: pop(); break;
    case 3: display(); break;
    default: exit(0);
}
}

```

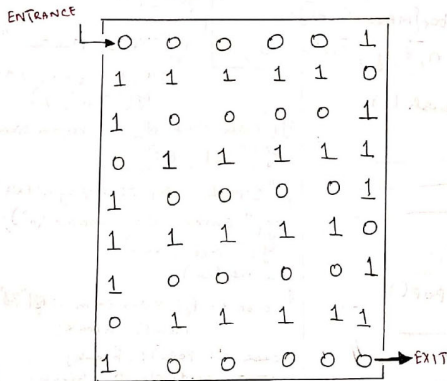
The Mazing Problem:-

- * A maze is a confusing network of paths, through which it is very difficult to find one's way.
- * The experimental psychologists train the rats to search mazes for food. A maze problem is a nice application of stack.

Maze Representation: A maze is represented as a two dimensional array in which:

- Zero's (0's) represent the open paths
- One's (1's) represent barriers

Example: A simple maze, along with path is shown:

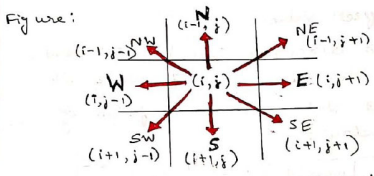


- * We assume that the rat starts at the top left and is to exit at the bottom right.

* The maze can be represented as 2D array, the location of the rat in the maze can be determined by the rows and column position.

* Let (i, j) is the current position of rat in the maze where i is the row index and j is the column index.

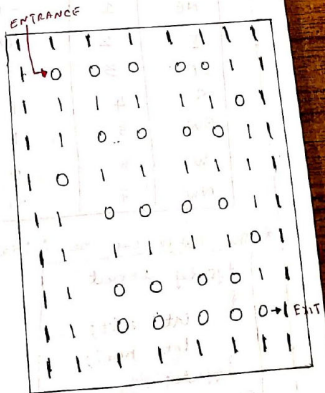
* Now the rat can move in 8 possible directions: north, ^{east}North east, South east, South, Southwest, west and northwest i.e: N, NE, E, SE, S, SW, W, NW resp.



* The rat can be moved in all 8 directions. But, this is not true all the time.

If the position of rat is on any of the border, then it cannot move in all 8 directions.

To avoid checking for these border conditions, we can surround the maze by a border of 1's as shown in figure.



Thus, a maze of $m \times n$ will require $(m+2) \times (n+2)$ size.

* Now the start starts at position $(1,1)$ and leaves at bottom right of the maze from the position (n,b) .

* Observe from the 8 directions figure, that either we add -1, or 0 or 1 to (i,j) to get new position. These values are called Offset Values.

* Thus, the 8 directions can be represented using the numbers 0 to 7 along with above offset values in the form of a table below:

Name	dir	move[dir].Vert	move[dir].Hori
N	0	-1	0
NE	1	-1	1
E	2	0	1
SE	3	1	1
S	4	1	0
SW	5	1	-1
W	6	0	-1
NW	7	-1	-1

* The table can be initialized using array of structures:

```
typedef struct
{
    int vert;
    int hori;
} offsets;
Offsets move[8];
```

* To find more n find the next a particular next next

* As we move several direct which choice and arbitrary; current position if we possible move clock wise. the maze array's or maze[low

Program:

```
typedef
```

```
{
    int
    int
```

```
} offsets;
```

```
Offsets
```

```
typedef
```

```
{
    int
    int
```

```
} Offsets;
```

```
Offsets
```

* To find more moves from the current position, we can find the next valid position of rat in the maze at a particular direction using:

```
nextRow = row + move[dir].vert;  
nextCol = col + move[dir].hor;
```

* As we move through the maze, we have the choice of several directions of movement. Since we do not know which choice is best, we save our current position and arbitrarily pick a possible move. By saving our current position, we can return to it and try another path if we take a wrong path. We examine the possible moves starting from the north and moving clockwise. We use a 2D array mark - to record the maze positions already checked. We initialize this array's entries to zero. When we visit a position maze[row][col], we change mark[row][col] to 1.

Program:

```
#define EXIT_ROW 12  
#define EXIT_COL 15  
typedef struct  
{  
    int vert;  
    int hor;  
} effects;  
effects move[8] = { { -1, 0 }, { -1, 1 }, { 0, 1 }, { 1, 1 }, { 1, 0 },  
                  { 1, -1 }, { 0, -1 }, { -1, -1 } };  
typedef struct  
{  
    int row;  
    int col;  
    int dir;  
} element;  
enum { FALSE, TRUE };
```



```

int maze[20][20] = {
    0 { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    1 { 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    2 { 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    3 { 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
    4 { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
};

```

```
void main()
```

```

{
    int r, c, nextRow, nextCol, dir, found = FALSE, top, i;
    int mark[20][20] = { 0 };
    element s[200], pos;
    mark[1][1] = 1;
    pos.row = 1, pos.col = 1; // Rat starts as pos(1,1)
    pos.dir = 0;
    s[top] = pos; // Push the current pos into stack
    while (top != -1 && !found)

```

```

{
    pos = s[top--];
    row = pos.row;
    col = pos.col;
    dir = pos.dir;

```

```
while (dir < 8 && !found)
```

```

{
    nextRow = row + move[dir].vert;
    nextCol = col + move[dir].horiz;
    if (nextRow == EXIT_ROW && nextCol == EXIT_COL)
        found = TRUE;
}

```

```
else if (maze[nextRow
```

```

{
    mark[nextRow]
    pos.row = r;
    s[top+1] =
    row = next
}

```

```

}
else
    r + dir;
}

```

```

}
}

```

```
if (found)
```

```

{
    for
}

```

```

}
else

```

```

}
result

```

```

1
1
1
1
1

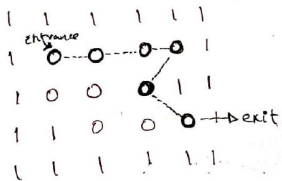
```

```

else if ((MAZE[nextRow][nextCol] == 0 && mark[nextRow][nextCol] == 0)
{
    mark[nextRow][nextCol] = 1;
    Pos.row = row, Pos.col = col, Pos.dir = ++dir;
    S[++top] = Pos;
    row = nextRow, col = nextCol, dir = 0;
}
else
    ++dir;
}
}
}
if (found) // print the path position
{
    for (i=0; i <= top; i++)
        Pj("%d %d \t", S[i].row, S[i].col);
    Pj("%d %d \t", EXIT_ROW, EXIT_COL);
}
else
    Pj("\n Maze has no path \n");
}

```

Resultant path for the above example



Path of Movements:
 (1,1), (1,2), (1,3), (1,4),
 (2,3), (3,4)