Anisha .H.

Mod-2

lab pro - 14, 18, 28, 37

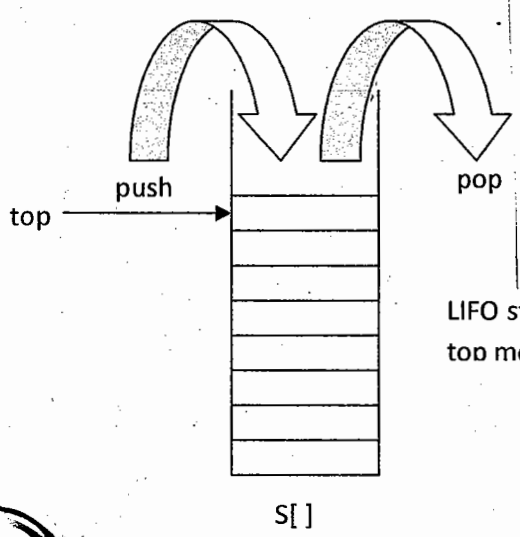
# Stacks

## Basic Operations

Stack is a special type of data structure where elements are inserted from one end called 'top' of the stack and elements are deleted from the same end.

- ✓ Using this approach the last element inserted is the first element to be deleted, so stack is also called as Last-In-First-Out (LIFO) data structure.
- ✓ Inserting an element to the stack is called as 'push' operation. Once the stack is full if we try to insert an element stack overflow occurs.
- ✓ Deleting an element from the stack is called 'pop' operation. Once the stack is empty if we try to delete an element stack underflow occurs.

The schematic representation of a stack is as shown below.




LIFO structure with top pointing to the top most element of the stack.

Dr.Mahesh G, Assoc. Prof., BMSIT & M
Dr.Harish G, Assoc. Prof., Dr. AIT

The different operations that can be performed on a stack are

- ✓ Insertion.
- ✓ Deletion.
- ✓ Display.
- ✓ Check whether the stack is full or not.
- ✓ Check whether the stack is empty or not.

**Implementation of Stacks using Arrays**

```
#include<stdio.h>
#define stacksize 5

// function to insert an element into the stack
void push(int item, int *top, int s[ ])
{
        if(*top = = stacksize -1)
        {
                printf("Stack overflow\n");
                return;
        }
```

```
        *top = *top + 1;
        s[*top] = item;
}
```

**// function to delete an element from the stack**

```
int pop(int *top, int s[ ])
{
        int item;
        if(*top = = -1)
        {
                return -1;
        }
        item = s[*top];
        *top = *top - 1;
        return(item);
}
```

**// function to display the contents of a stack**

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

}

void main( )
{
        int top, choice, item, s[stacksize], y;
        top = -1;
        for(;;)
        {
                printf("1.Push 2.Pop 3.Display 4.Exit\n");
                printf("Enter your choice\n");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1: printf("Enter element to insert\n");
                                scanf("%d", &item);
                                push(item,&top, s);
                                break;
```

Dr.Mahesh G Assoc. Prof. BMSIT & M
Dr.Harish G Assoc. Prof. Dr. AIT

```
                case 2: y = pop(&top, s);
                        if(y = = -1)
                                printf("Stack underflow\n");
                        else
                                printf("item deleted = %d\n", y);
                        break;
                case 3: display(top, s);
                        break;
                case 4: exit(0);
                }
        }
}
```

**Other Stack Functions**

```
// function to check if the stack is full
void isfull(int top)
{
        if(top = = stacksize -1)
        {
                printf("Stack is full\n");
        }
        else
        {
                printf("Stack is not full\n");
        }
}
// function to check if the stack is empty
void isempty(int top)
{
        if(top = = -1)
        {
                printf("Stack is empty\n");
        }
        else
        {
                printf("Stack is not empty\n");
        }
}
```

Dr.Mahesh G, Assoc. Prof., BMSIT & M
Dr.Harish G, Assoc. Prof., Dr. AIT

**Applications of Stacks**

Following are some of the applications of stacks.

- ✓ Stacks are used in recursive functions.
- ✓ Conversion of expressions.
- ✓ Evaluation of Expressions.
- ✓ For Traversing trees and graphs.
- ✓ Reversal of a string.
- ✓ Check if a given string is a palindrome or not.
- ✓ Check for parenthesis matching.

## System Stack and Activation Record

A stack used by the program at runtime is called as a system stack. The system stack is used when functions are invoked and executed.

Note:

- ✓ When a function is called, a structure called an activation record is created on top of the stack.
- ✓ When a function is terminated, the activation record is deleted from the top of the system stack.

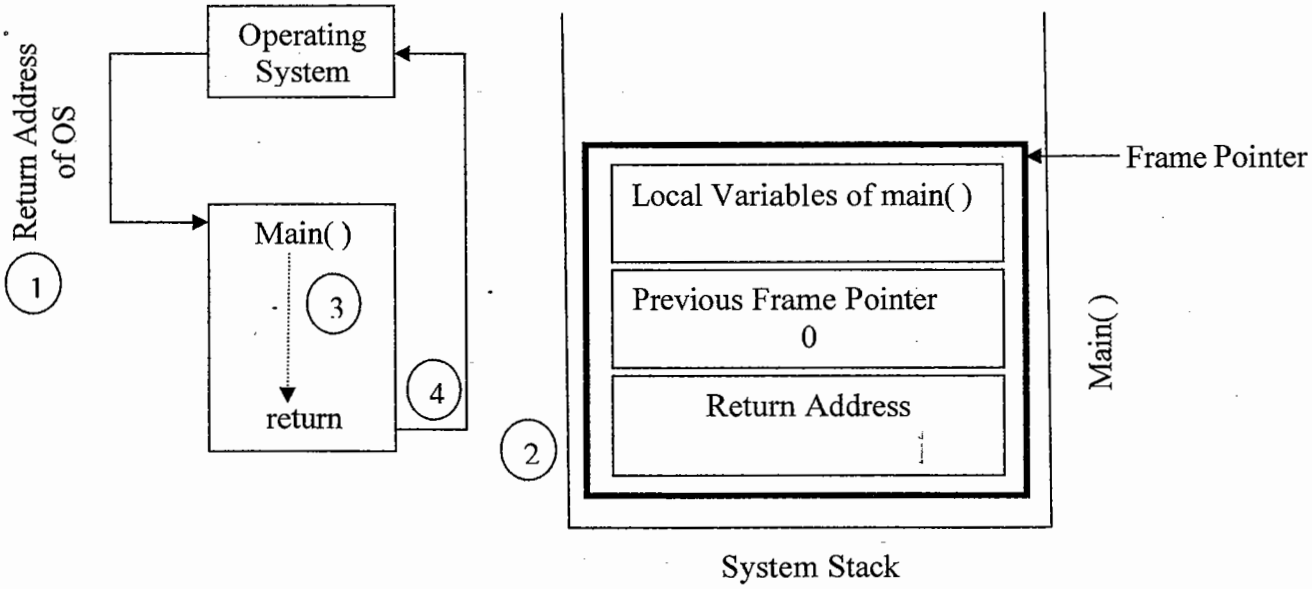
An activation record is also called as a stack frame. It is a structure consisting of various fields to store the local variables (if any), parameters (if any), return addresses and pointer to the previous stack frame. Using the activation record, the functions can communicate with each other.

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

The various actions that take place when we execute a program are,

- ✓ The user gives the command to execute the program. Then, the OS calls the functions main( ) and the body of the main( ) is executed.
- ✓ After executing the body of the function main( ), the control is returned to the operating system.

This is as shown in the following diagram




**Step1:** The operating system transfers the control to the function main( ) along with the return address so that after executing the body of the function main( ), the control is returned to OS.

**Step 2:** A stack frame or activation record is created consisting of

- ✓ Return address of OS
- ✓ Local variables of main( )
- ✓ Previous frame pointer (NULL in this case as it is the first record on the system stack)

This stack frame is pushed onto the stack.

**Step 3:** The main( ) function is executed now.

**Step 4:** When the return statement is executed in the main( ), the activation record is removed from the stack. Using the removed activation record, we can get the return address of the OS using which the control is transferred to OS.

Now suppose a function a1( ) is invoked in main( ), the following are the further steps involved.

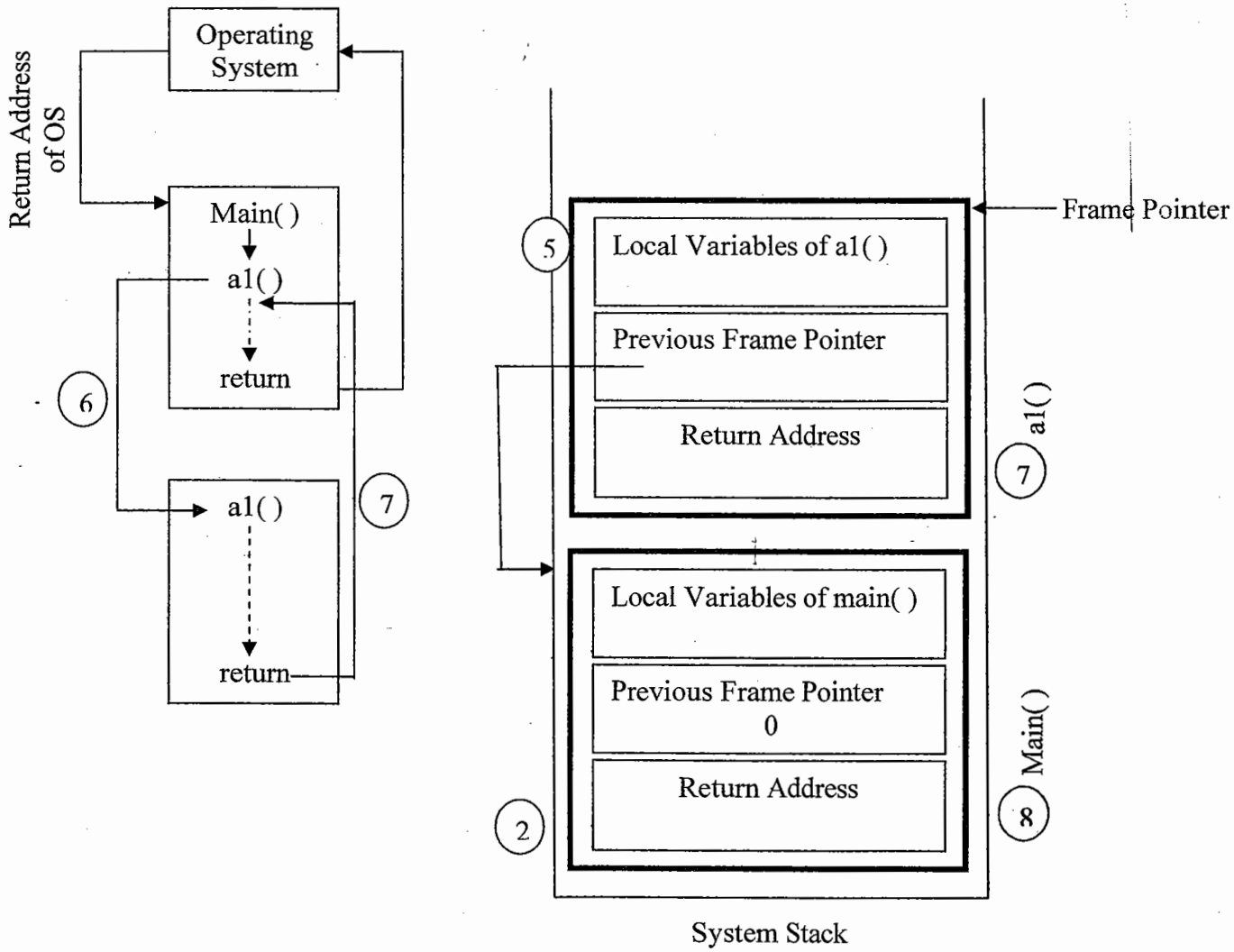
**Step 5:** When main( ) is being executed, if the function a1( ) is invoked, a new stack frame or activation record is created consisting of

- ✓ Return address of main( )
- ✓ Local variables of a1( )
- ✓ Current frame pointer which is on top of the stack.

**Dr.Mahesh G** Assoc. Prof. BMSIT & M
**Dr.Harish G** Assoc. Prof. Dr. AIT

This new stack frame is pushed onto the stack.

Note that the previous frame pointer of activation record which is on top of the stack points to the activation record of the main( ).




**Step 6:** Control is transferred to the function a1( ).

**Step 7:** When the return statement in a1( ) is executed, the activation record on top of the stack is removed. Using the removed activation record, we can get the return address of the main using which the control is transferred to main( ).

**Step 8:** Finally, when the body of the function main( ) is completely executed, the return address of OS is obtained from the activation record and control is transferred to OS.

## Stack Abstract Datatype

**Objects:** A finite ordered list with zero or more elements.

**Functions:**

For all s Є Stack, item Є element, stacksize Є positive integer.

*Stack* Create(stacksize) ::= Creates an empty stack whose maximum size is stacksize

*Boolean* IsFull(s, *stacksize*) := **if** number of elements in stack s is stacksize **return** *TRUE*
**else return** *FALSE*

*Stack* Push*(s, item)* ::= **if** (IsFull(s)) stack is full
**else** insert item on to the stack s and return

*Boolean* IsEmpty(s) ::= **if** no elements are there in stack **return** *TRUE*
**else return** *FALSE*

*Element* Pop(s) ::= **if** (IsEmpty(s)) **return**
**else** remove and return the element at the top of the stack

## Stacks using Dynamic Arrays (with the concept of array doubling)

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT &M
Assoc. Prof. Dr. AIT

```
#include<stdio.h>
int capacity = 5;
int *s;
// function to insert an element into the stack
void push(int item, int *top)
{
    if(*top = = capacity -1)
    {
        // When stack is full double the size using realloc function
        s = (int *) realloc(s, 2 * capacity * sizeof(int));
        capacity = capacity * 2;
    }
```

```
        *top = *top + 1;
        s[*top] = item;
}

// function to delete an element from the stack
int pop(int *top)
{
        int item;
        if(*top = = -1)
        {
                return -1;
        }
        item = s[*top];
        *top = *top - 1;
        return(item);
}

// function to display the contents of a stack
void display(int top)
{
        int i;
        if(top = = -1)
        {
                printf("Stack is empty\n");
                return;
        }
        printf("Contents of the stack are\n");
        for(i = 0; i< = top; i + +)
        {
                printf("%d\n", s[i]);
        }

}
```

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

```
void main( )
{
        int top, choice, item, y;
        top = -1;
        s = (int *) malloc(capacity*sizeof(int));
        for(;;)
        {
                printf("1.Push  2.Pop  3.Display  4.Exit\n");
                printf("Enter your choice\n");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1: printf("Enter element to insert\n");
                                scanf("%d", &item);
                                push(item, &top);
                                break;
                        case 2: y = pop(&top);
                                if(y = = -1)
                                        printf("Stack underflow\n");
                                else
                                        printf("item deleted = %d\n", y);
                                break;
                        case 3: display(top);
                                break;
                        case 4: free(s);
                                exit(0);
                }
        }
}
```

# Expressions

**Types of Expressions**

The different types of expressions are

- ✓ Infix expression – Here an operator is in between 2 operands
  EX: (a+b) / (c-d)
- ✓ Postfix (Suffix or Reverse Polish) expression – Here an operator follows the 2 operands.
  EX: ab+cd-/
- ✓ Prefix (Polish) expression – Here an operator precedes the 2 operands.
  EX: /+ab-cd

In both postfix and prefix expression operator and operands define correctly the order of operation.

**Problem 1**

Convert the following infix expression to postfix and prefix
( ( 6 + ( 3 – 2 ) * 4 ) ^ 5 + 7 )

((6 + T1 * 4) ^ 5 + 7)
( ( 6 + T2 ) ^ + 7 )
T3 ^ 5 + 7
T4 + 7
T5

Postfix Conversion

( ( 6 + ( 3 – 2 ) * 4 ) ^ 5 + 7 ) // T1 = 3 2 -

( ( 6 + T1 * 4 ) ^ 5 + 7 ) // T2 = T1 4 *

( ( 6 + T2 ) ^ 5 + 7 ) // T3 = 6 T2 +

( T3 ^ 5 + 7 ) // T4 = T3 5 ^

( T4 + 7 ) // T5 = T4 7 +

T5

---

T4 7 +

T3 5 ^ 7 +

6 T2 + 5 ^ 7 +

6 T1 4 * + 5 ^ 7 +

6 3 2 - 4 * + 5 ^ 7 + (Postfix Expression)

| Dr.Mahesh G | Dr.Harish G |
|---|---|
| Assoc. Prof. | Assoc. Prof. |
| BMSIT & M | Dr. AIT |

Prefix Conversion

( ( 6 + ( 3 – 2 ) * 4 ) ^ 5 + 7 ) // T1 = - 3 2

( ( 6 + T1 * 4 ) ^ 5 + 7 ) // T2 = * T1 4

( ( 6 + T2 ) ^ 5 + 7 ) // T3 = + 6 T2

( T3 ^ 5 + 7 ) // T4 = ^ T3 5

( T4 + 7 ) // T5 = + T4 7

T5

---

+T4 7

+ ^ T3 5 7

+ ^ + 6 T2 5 7

+ ^ + 6 * T1 4 5 7

+ ^ + 6 * - 3 2 4 5 7 (Prefix Expression)

**Problem 2**

Convert the following infix expression to postfix and prefix
a + b ^ c ^ d

Postfix Conversion

a + b ^ c ^ d // T1 = c d ^

a + b ^ T1 // T2 = b T1 ^

a + T2 // T3 = a T2 +

T3

---

a T2 +

a b T1 ^ +

a b c d ^ ^ + (Postfix Expression)

Dr.Mahesh G Dr.Harish G
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

Prefix Conversion

a + b ^ c ^ d // T1 = ^ c d

a + b ^ T1 // T2 = ^ b T1

a + T2 // T3 = + a T2

T3

---

+ a T2

+ a ^ b T1

+ a ^ b ^ c d (Prefix Expression)

**Problem 3 (Scan from right to left)**
Convert the prefix expression + - A B C to infix
+ - A B C T1 = A – B

+ T1 C T2 = T1 + C

T2

(T1 + C)

((A – B)+ C)

**Problem 4 (Scan from right to left)**
Convert the prefix expression + A – B C to infix
+ A – B C T1 = B – C

+ A T1 T2 = A + T1

T2

(A + T1)

(A + (B – C) )

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

**Problem 5 (Scan from left to right)**
Convert the postfix expression A B + C - to infix
A B + C - T1 = A + B

T1 C - T1 = T1 – C

T2

(T1 – C)

((A+B) -C)

**Problem 6 (Scan from left to right)**
Convert the postfix expression A B C + - to infix
A B C + - T1 = B + C

A T1 - T2 = A - T1

T2

(A – T1)

(A – (B+C))

# Infix to Postfix Conversion

**Algorithm to convert infix expression to postfix**

The 2 functions which will be used while converting an expression from infix to postfix are

- ✓ Input precedence function
- ✓ Stack precedence function

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

Note:

1. +, -, *, / and % are left associative. ^ and = are right associative.
2. If the operators are left associative, input precedence is less than the stack precedence.
3. If the operators are right associative, input precedence is greater than the stack precedence.
4. The left parenthesis in the input has the highest precedence and on the stack its precedence is zero.
5. The right parenthesis in the input has the precedence value of zero.
6. + and – have the least precedence (left associative) and next comes *, / and % (left associative) then exponentiation (right associative) and then operands and finally left parenthesis.
7.

| Input Precedence | | Stack Precedence | |
|---|---|---|---|
| Symbol | Value | Symbol | Value |
| +, - | 1 | +, - | 2 |
| *, /, % | 3 | *, /, % | 4 |
| ^ or $ | 6 | ^ or $ | 5 |
| Operands | 7 | Operands | 8 |
| ( | 9 | ( | 0 |
| ) | 0 | # | -1 |

**General Procedure**

**Step 1:** Obtain the next input symbol from the infix expression
**Step 2:** While precedence of the symbol which is there on top of the stack is greater than the precedence of the current input symbol, pop an element from the stack and place it into the postfix expression.
**Step 3:** If stack precedence symbol and input precedence symbol values are different push the current symbol on to the stack otherwise (if they are equal) delete an element from the stack,
**Step 4:** Repeat through Step 1 till you encounter end of input.
**Step 5:** If stack contains any other symbols other than #, pop all the symbols from the stack and place them in the postfix expression till '#' is encountered on top of the stack.

Note: If un-parenthesized infix expression is used, only a partial postfix expression is obtained and still some more symbols are left on top of stack. Therefore Step 5 is needed in the general procedure.

**Problem 1**

Convert the expression ( 6 + ( 5 – 3 ) * 4 ) to postfix

| Scanned Symbol | Stack Contents | Operation | Postfix Expression |
|---|---|---|---|
| ( | # | -1 ≠ 9, Push ( | |
| 6 | # ( | 0 ≠ 7, Push 6 | |
| + | # ( 6 | 8 > 1, Pop 6 | 6 |
| | # ( | 0 ≠ 1, Push + | |
| ( | # ( + | 2 ≠ 9, Push ( | |
| 5 | # ( + ( | 0 ≠ 7, Push 5 | 65 |
| - | # ( + ( 5 | 8 > 1, Pop 5 | |
| | # ( + ( | 0 ≠ 1, Push - | |
| | | | |
| 3 | # ( + ( - | 2 ≠ 7, Push 3 | 653 |
| ) | # ( + ( - 3 | 8 > 0, Pop 3 | 653- |
| | # ( + ( - | 2 > 0, Pop - | |
| | # ( + ( | 0 = 0, Delete ( | |
| * | # ( + | 2 ≠ 3, Push * | |
| 4 | # ( + * | 4 ≠ 7, Push 4 | 653-4 |
| ) | # ( + * 4 | 8 > 0, Pop 4 | 653-4* |
| | # ( + * | 4 > 0, Pop * | 653-4*+ |
| | # ( + | 2 > 0, Pop + | |
| | # ( | 0 = 0, Delete ( | |
| End of input | # | | 653-4*+ '\0' |

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

**Problem 2**

Convert the expression 6 + 4 * 3 to postfix

| Scanned Symbol | Stack Con tents | Operation | Postfix Expression |
|---|---|---|---|
| 6 | # | -1 ≠ 7, Push 6 | |
| + | # 6 | 8 > 1, Pop 6 | 6 |
| | # | -1 ≠ 1, Push + | |
| 4 | # + | 2 ≠ 7, Push 4 | |
| * | # + 4 | 8 > 3, Pop 4 | 6 4 |
| | # + | 2 ≠ 3 Push * | |
| 3 | # + * | 4 ≠ 7, Push 3 | |
| End of Input | # + * 3 | Pop 3 | 6 4 3 |
| | # + * | Pop * | 6 4 3 * |
| | # + | Pop + | 6 4 3 * + |
| | # | | 6 4 3 * + '\0' |

29/01/18

Lab Prog 2

**Program to convert infix expression to postfix**

```
#include<stdio.h>
# define stacksize 70

int input_precedence(char symbol)
{
        switch(symbol)
        {
              case '+' :
              case '-' :      return 1;
              case '*' :
              case '/' :
              case '%' :      return 3;
              case '^' :
              case '$' :      return 6;
              case '(' :      return 9;
              case ')' :      return 0;
              default :       return 7;
        }
}

int stack_precedence(char symbol)
{
        switch(symbol)
        {
                case '+' :
                case '-' :      return 2;
                case '*' :
                case '/' :
                case '%' :      return 4;
                case '^' :
                case '$' :      return 5;
                case '(' :      return 0;
                case '#' :      return -1;
                default :       return 8;
        }
}

void push(char item, int *top, char s[])
{
       if(*top = = stacksize-1)
       {
              printf("stack overflow\n");
              return;
       }
```

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

```
        *top = *top + 1;
        s[*top] = item;
}

char pop(int *top, char s[])
{
        char item;
        if(*top == -1)
                return -1;
        item = s[*top];
        *top = *top - 1;
        return item;
}

void infix_postfix(char infix[], char postfix[])
{
        int i, n, j, top;
        char symbol, s[stacksize];
        j = 0;
        top = -1;
        push('#', &top, s);
        n = strlen(infix);
        for(i=0; i<n; i++)
        {
                symbol = infix[i];
                while(stack_precedence(s[top]) > input_precedence(symbol))
                {
                        postfix[j] = pop(&top, s);
                        j = j + 1;
                }
                if(stack_precedence (s[top]) != input_precedence(symbol))
                {
                        push(symbol, &top, s);
                }
                else
                {
                        pop(&top, s);
                }
        }
        while(s[top] != '#')
        {
                postfix[j] = pop(&top, s);
                j = j + 1;
        }
        postfix[j] = '\0';
}
```

```
void main( )
{
        char infix[70], postfix[70];
        clrscr( );
        printf("enter the infix expression\n");
        scanf("%s",infix);
        infix_postfix(infix, postfix);
        printf("the postfix expression is %s",postfix);
        getch( );
}
```

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

## Evaluation of Postfix Expression

**Infix Expression:** Evaluating an infix expression requires repeated scanning from left to right and right to left because arithmetic operations have to be performed based on precedence of operations. If there are parenthesis in the expression, the problem becomes quite complex because parenthesis change the order of precedence.

**Postfix and Prefix Expression:** Here the operator and operands define correctly the order of operations. Since no parenthesis is present, there is no repeated scanning and time taken is less.

**Algorithm for Evaluation of Postfix Expression**

**Step 1:** Scan from left to right till we get an operator.
**Step 2:** Obtain immediate 2 left operands.
**Step 3:** Perform the indicated operation.
**Step 4:** Replace the operands and the operators by the result.
**Step 5:** Repeat through Step 1 till the end of input.

**Example:** Consider the postfix expression 6 3 2 – 5 * + for the infix expression ( 6 + ( 3 – 2 ) * 5 ). It is evaluated as shown below.

6 3 2 – 5 * + 3 – 2 = 1
6 1 5 * + 1 * 5 = 5
6 5 + 6 + 5 = 11

**Pseudocode for Evaluation of Postfix Expression**

```
While( not end of input)
{
        Obtain the next input symbol
        if(symbol is an operand)
                push(symbol, top, s)
        else
        {
                op2 = pop(top, s);
                op1 = pop(top, s);
                res = op1 op op2;
                push(res, top, s);
        }
}
return( pop(top, s));
```

**Dr.Mahesh G** Assoc. Prof. BMSIT & M
**Dr.Harish G** Assoc. Prof. Dr. AIT

**Example:** Evaluation of the postfix expression 6 3 2 – 5 * + is as shown below

| Scanned Symbol | op2 | op1 | res = op1 op op2 | Stack Contents |
|---|---|---|---|---|
| 6 | | | | 6 |
| 3 | | | | 6 3 |
| 2 | | | | 6 3 2 |
| - | 2 | 3 | 3 – 2 = 1 | 6 1 |
| 5 | | | | 6 1 5 |
| * | 5 | 1 | 1 * 5 = 5 | 6 5 |
| + | 5 | 6 | 6 + 5 = 11 | 11 |
| End of input | | | | Return 11 |

29/9/18

Top is always int

lab Program

**Program for Evaluation of Postfix Expression**

```
#include<stdio.h>   (math.h) (string.h)
# define stacksize 70

void push(int item, int *top, int s[])
{
        if(*top = = stacksize-1)
        {
              printf("stack overflow\n");
              return;
        }
        *top = *top + 1;
        s[*top] = item;
}
int pop(int *top, int s[])
{
      int item;
      if(*top = = -1)
              return -1;
      item = s[*top];
      *top = *top - 1;
      return item;
}
int evaluate(char postfix[])
{
          int i, n, op1, op2, res, op, top, s[stack_size];
          char symbol;
          top = -1;
          n = strlen(postfix);
          for(i=0; i<n; i++)
          {
               symbol = postfix[i];
               switch(symbol)
               {
                    case '+':op2 = pop(&top, s);
                             op1 = pop(&top, s);
                             res = op1+op2;
                             push(res, &top, s);
                             break;

                    case '-':op2 = pop(&top, s);
                             op1 = pop(&top, s);
                             res = op1-op2;
                             push(res, &top, s);
                             break;
```

| Dr.Mahesh G | Dr.Harish G |
|---|---|
| Assoc. Prof. | Assoc. Prof. |
| BMSIT & M | Dr. AIT |

```
                case '*':op2 = pop(&top, s);
                         op1 = pop(&top, s);
                         res = op1*op2;
                         push(res, &top, s);
                         break;

                case '/':op2 = pop(&top, s);
                         op1 = pop(&top, s);
                         res = op1/op2;
                         push(res, &top, s);
                         break;

                case '%':op2 = pop(&top, s);
                         op1 = pop(&top, s);
                         res = op1% op2;
                         push(res, &top, s);
                         break;

                case '^':
                case '$':op2 = pop(&top, s);
                         op1 = pop(&top,s);
                         res = (int)pow((double)op1,(double)op2);
                         push(res, &top, s);
                         break;

                default: push(symbol-'0', &top, s);
            }
        }
        return(pop(&top, s));
}

void main( )
{
        int res;
        char postfix[70];
        clrscr();
        printf("enter the postfix expression\n");
        scanf("%s",postfix);
        res=evaluate(postfix);
        printf("the value of the postfix exp is %d",res);
        getch( );
}
```

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

**Note:** symbol – '0' is used to convert character to integer.
EX: '6' – '0' = 54 – 48 = 6

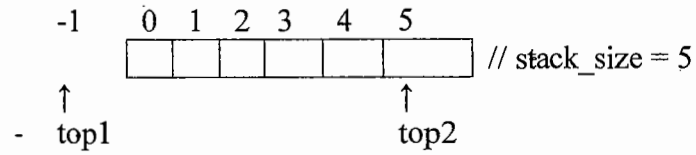
25/9/18

## Multiple Stacks and Queues

If we have to represent 2 stacks in a single array, then we can use

- ✓ s[0] to represent the bottom element of stack 1 and s[stack_size-1] to represent the bottom element of stack 2.
- ✓ top1 corresponding to stack 1 will be initialized to -1 and top2 corresponding to stack 2 will be initialized to stack_size.
- ✓ stack 1 will grow towards stack_size -1 and stack 2 will grow towards 0

With this representation, we can efficiently use all the available space and the idea is as shown in the below figure.




Representing more than 2 stacks in a single array poses problems since we no longer have an obvious point for the bottom element of each stack. Two possible solutions for 'n' stacks in a single array are

- ✓ Divide the array into 'n' segments based on the expected sizes of various stacks.
- ✓ Divide the array into 'n' equal segments.

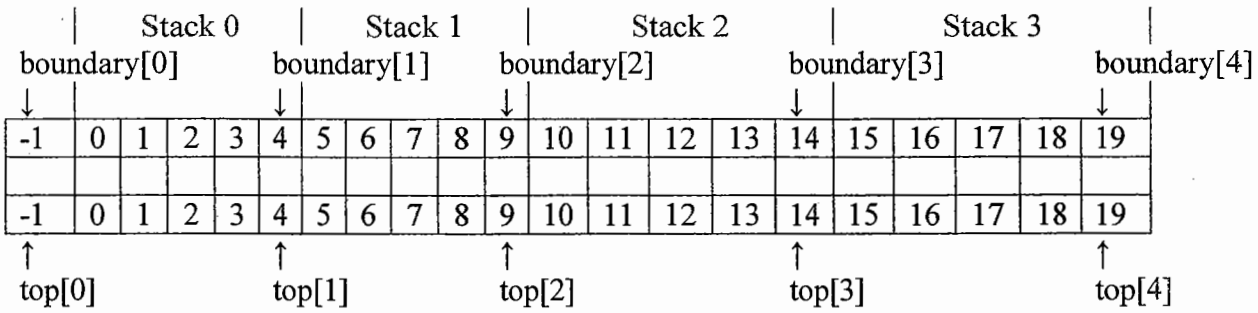
For dividing the array into 'n' equal segments the following pseudocode can be used.

```
#define stack_size 100          // memory size
#define max_stacks 10           // maximum number of stacks
int s[stack_size];              // stack array
int n;                          // number of stacks
int top[max_stacks];            // variable top for each of the 'n' stacks
int boundary[max_stacks];       // to check for boundary conditions

for(j = 0; j <= n; j++)
{
        top[j] = boundary[j] = stack_size / n * j – 1;
}
```

top[0] = B[0] = 20/4 * 0 - 1 = -1

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

The diagrammatic representation of this is as shown below.



```
// function to insert an element into the stack with number 'i'
void push(int i, int item)        // 'i' is the stack number
{
        if(top[i] = = boundary[i+1])
        {
                printf("Stack overflow\n");
                return;
        }
        top[i] = top[i] + 1;
        s[top[i]] = item;
}

// function to delete an element from the stack with number 'i'
int pop(int i)
{
        int item;
        if(top[i] = = boundary[i])
        {
                return -1;
        }
        item = s[top[i]];
        top[i] = top[i] - 1;
        return(item);
}

// function to display the contents of the stack with number 'i'
void display(int i)
{
        int j;
        if(top[i] = = boundary[i])
        {
                printf("Stack is empty\n");
                return;
        }
        printf("Contents of the Stack\n");
        for(j=boundary[i]+1; j<=top[i]; j++)
        {
                printf("%d\n", s[j]);
        }
}
void main( )
{
        int choice, i, j, y;
        printf("Enter  the number of stacks\n");
        scanf("%d", &n);

        for(j = 0; j <= n; j + + )
        {
                top[j] = boundary[j] = stack_size / n * j – 1;
        }
```

```
        for(;;)
        {
                printf("1.Push 2.Pop 3.Display 4.Exit\n");
                printf("Enter your choice\n");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1: printf("Enter the stack number\n");
                                scanf("%d", &i);
                                printf("Enter element to insert\n");
                                scanf("%d", &item);
                                push(i, item);
                                break;

                        case 2: printf("Enter the stack number\n");
                                scanf("%d", &i);
                                y = pop(i);
                                if(y = = -1)
                                        printf("Stack underflow\n");
                                else
                                        printf("item deleted = %d\n", y);
                                break;

                        case 3: printf("Enter the stack number\n");
                                scanf("%d", &i);
                                display(i);
                                break;

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



The push and pop functions for multiple stacks appear to be simple and is almost same as that of a single stack. However, this is not really the case, because top[i] = = boundary[i+1] condition in push implies only that stack ran out of memory and not the entire memory is full. In fact, there may be lot of unused space between other stacks in the array. Therefore, we need to determine if there is any free space in the array, and if there is space available, it should shift stacks so that space allocated to the stack that is full.

There are several methods to deal with this stack full problem. All the methods add elements to the array as long as there is free space in the array. One method to add an element when stack 'i' is full is but the complete array is not full is,

1. Right Check – Determine the least 'j' such that i<j<n and there is free space between stacks 'j' and 'j+1'. If there is one such 'j' then move stacks i+1, i+2,.....j by one position to right. This creates space between stacks 'i' and 'i+1'

2. Left Check – If there is no 'j' as in 1 then make a left check i.e. find the largest 'j' such that $0 \leq j < i$, and there is free space between stacks 'j' and 'j+1'. If there is one such 'j' then move stacks j+1, j+2,.....,i, one space to the left. This also creates a space between stacks 'i' and 'i+1'.
3. If there is no 'j' satisfying 1 or 2 then all the memory space is utilized and there is no free space. In this case the stack is full and space cannot be allocated for adding an element.

**Program to implement 2 stacks in a single array.**

```
#include<stdio.h>
#define stacksize 10

// function to insert an element into the stack 1
void push1(int item, int *top1, int *top2, int s[ ])
{
        if(*top1 = = *top2 -1)
        {
                printf("Stack overflow\n");
                return;
        }
        *top1 = *top1 + 1;
        s[*top1] = item;
}

// function to insert an element into the stack 2
void push2(int item, int *top1, int *top2, int s[ ])
{
        if(*top1 = = *top2 -1)
        {
                printf("Stack overflow\n");
                return;
        }
        *top2 = *top2 - 1;
        s[*top2] = item;
}

// function to delete an element from the stack1
int pop1(int *top1, int s[ ])
{
        int item;
        if(*top1 = = -1)
        {
                return -1;
        }
        item = s[*top1];
        *top1 = *top1 - 1;
        return(item);
}
```

Dr.Mahesh G Assoc. Prof. BMSIT & M
Dr.Harish G Assoc. Prof. Dr. AIT

**// function to delete an element from the stack2**

```
int pop2(int *top2, int s[ ])
{
        int item;
        if(*top2 = = stack_size)
        {
                return -1;
        }
        item = s[*top2];
        *top2 = *top2 + 1;
        return(item);
}
```

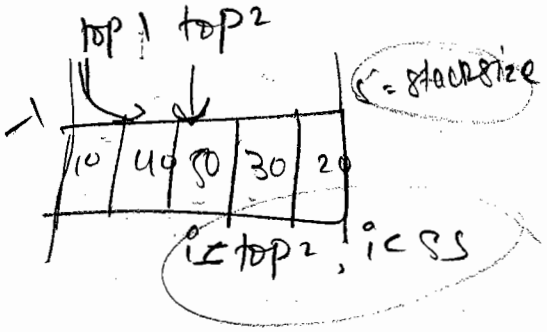
**// function to display the contents of a stack1**

```
void display1(int top1, int s[])
{
        int i;
        if(top1 = = -1)
        {
                printf("Stack1 is empty\n");
                return;
        }
        printf("Contents of the stack1 are\n");
        for(i = 0; i< = top1; i + +)
        {
                printf("%d\n", s[i]);
        }
}
```

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

**// function to display the contents of a stack2**

```
void display2(int top2, int s[])
{
        int i;
        if(top2 = = stack_size)
        {
                printf("Stack2 is empty\n");
                return;
        }
        printf("Contents of the stack2 are\n");
        for(i = top2; i< stack_size; i + +)
        {
                printf("%d\n", s[i]);
        }

}
```



**Program to reverse a given string using stacks**

```
#include<stdio.h>
#define stacksize 50
void push(char item, int *top, char s[ ]) // function to insert an element into the stack
{
        if(*top = = stacksize -1)
        {
                printf("Stack overflow\n");
                return;
        }
        *top = *top + 1;
        s[*top] = item;
}
char pop(int *top, char s[ ])  // function to delete an element from the stack
{
        char item;
        if(*top = = -1)
        {
                return -1;
        }
        item = s[*top];
        *top = *top - 1;
        return(item);
}
void reverse_string(char str[ ], char revstr[ ])  // function to reverse the string
{
        int top, n;
        char s[stacksize], symbol;
        top = -1;
        n = strlen(str);

        for(i = 0; i < n; i + + )
        {
                symbol = str[i];
                push(symbol, &top, s);
        }
        for(i = 0; i < n; i + + )
        {
                revstr[i] = pop(&top, s);
        }
        revstr[i] = '\0';
}
void main( )
{
        char str[50], revstr[50];
        printf("enter the string to be reversed\n");
        scanf("%s", str);
        reverse_string(str, revstr);
        printf("the reversed string is %s\n", revstr);
}
```

| Dr.Mahesh G | Dr.Harish G |
|---|---|
| Assoc. Prof. | Assoc. Prof. |
| BMSIT & M | Dr. AIT |

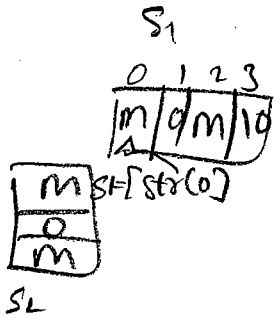
1st sym in string. → Push that sym to top of stack.

Pop that str to top of S = revstr[i]

**Program to check whether a given string is a palindrome or not using stacks**

```
#include<stdio.h>
#define stacksize 50
void push(char item, int *top, char s[ ]) // function to insert an element into the stack
{
        if(*top = = stacksize -1)
        {
                printf("Stack overflow\n");    return;
        }
        *top = *top + 1;   s[*top] = item;
}
char pop(int *top, char s[ ])  // function to delete an element from the stack
{
        char item;
        if(*top = = -1)
        {
                return -1;
        }
        item = s[*top];  *top = *top - 1;  return(item);
}
int check_palindrome(char str[ ])  // function to check for palindrome
{
        int top = -1, n;
        char s[stacksize], symbol, s1, s2;
        n = strlen(str);
        for(i = 0; i < n; i + + )
        {
                symbol = str[i];
                push(symbol, &top, s);
        }
        for(i = 0; i < n; i + + )
        {
                s1 = str[i];
                s2 = pop(&top, s);
                if(s1 ! = s2)
                        return 0;        // string is not a palindrome
        }
        return 1;        // string is a palindrome
}
void main( )
{
        char str[50];    int flag;
        printf("enter the string \n");
        scanf("%s", str);
        flag = check_palindrome(str);
        if(flag = = 0)
                printf("The given string is not a palindrome");
        else
                printf("The given string is a palindrome");
}
```





**Program to check whether a given number is a palindrome or not using stacks**

```
#include<stdio.h>
#define stacksize 50
void push(int item, int *top, int s[ ]) // function to insert an element into the stack
{
        if(*top = = stacksize -1)
        {
                printf("Stack overflow\n");    return;
        }
        *top = *top + 1;   s[*top] = item;
}
int pop(int *top, int s[ ])  // function to delete an element from the stack
{
        int item;
        if(*top = = -1)
        {
                return -1;
        }
        item = s[*top];  *top = *top - 1;  return(item);
}
int check_palindrome(char str[ ])  // function to check for palindrome
{
        int top = -1, n, s[stacksize], s1, s2;
        char symbol;
        n = strlen(str);
        for(i = 0; i < n; i + + )
        {
                symbol = str[i];
                push(symbol - '0', &top, s);
        }
        for(i = 0; i < n; i + + )
        {
                s1 = str[i] - '0';
                s2 = pop(&top, s);
                if(s1 ! = s2)
                        return 0;          // string is not a palindrome
        }
        return 1;          // string is a palindrome
}
void main( )
{
        char str[50];    int flag;
        printf("enter the number \n");
        scanf("%s", str);
        flag = check_palindrome(str);
        if(flag = = 0)
                printf("The given number is not a palindrome");
        else
                printf("The given number is a palindrome");
}
```

26/9/18

## Lab Program

Design, Develop and Implement a menu driven Program in C for the following operations on **STACK** of Integers (Array Implementation of Stack with maximum size **MAX**)

a. ***Push*** an Element on to Stack
b. ***Pop*** an Element from Stack
c. Demonstrate how Stack can be used to check ***Palindrome***
d. Demonstrate ***Overflow*** and ***Underflow*** situations on Stack
e. Display the status of Stack
f. Exit

Support the program with appropriate functions for each of the above operations

```
#include<stdio.h>
#define stacksize 8

// function to insert an element into the stack
void push(int item, int *top, int s[ ])
{
        if(*top = = stacksize -1)
        {
                printf("Stack overflow\n");
                return;
        }
        *top = *top + 1;
        s[*top] = item;
}

// function to delete an element from the stack
int pop(int *top, int s[ ])
{
        int item;
        if(*top = = -1)
        {
                return -1;
        }
        item = s[*top];
        *top = *top - 1;
        return(item);
}

// function to display the contents of a stack
void display(int top, int s[])
{
        int i;
        if(top = = -1)
        {
                printf("Stack is empty\n");
                return;
        }
```

Dr.Mahesh G Assoc. Prof. BMSIT & M
Dr.Harish G Assoc. Prof. Dr. AIT

```
        printf("Contents of the stack are\n");
        for(i = 0; i<= top; i + +)
        {
                printf("%d\n", s[i]);
        }
}

void main( )
{
        int top, choice, item, s[stacksize], y, flag;
        int n, i, s1, s2;
        char str[20], symbol;

        top = -1;

        for(;;)
        {
                printf("1.Push  2.Pop  3.Display 4. Check for Palindrome 5.Exit\n");
                printf("Enter your choice\n");
                scanf("%d", &choice);
                switch(choice)
                {
                        case 1: printf("Enter element to insert\n");
                                scanf("%d", &item);
                                push(item,&top, s);
                                break;

                        case 2: y = pop(&top, s);
                                if(y = = -1)
                                        printf("Stack underflow\n");
                                else
                                        printf("item deleted = %d\n", y);
                                break;

                        case 3: display(top, s);
                                break;

                        case 4: top = -1;      // Empty the stack before demo
                                flag = 0;
                                printf("enter the number \n");
                                scanf("%s", str);

                                n = strlen(str);
                                for(i = 0; i < n; i + + )
                                {
                                        symbol = str[i];
                                        push(symbol - '0', &top, s);
                                }
```

```
                for(i = 0; i < n; i + + )
                {
                        s1 = str[i] - '0' ;
                        s2 = pop(&top, s);
                        if(s1 ! = s2)
                        {
                                flag = 1;       // string is not a palindrome
                                break;
                        }
                }
                if(flag = = 1)
                        printf("The given number is not a palindrome");
                else
                        printf("The given number is a palindrome");
                top = -1;       // Empty the stack after demo
                break;

        case 5: exit(0);
        }
    }
}
```

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

**Recursion**

It is a process in which an object is defined in terms of simpler case of itself.

**Direct Recursion**

A function which calls itself repeatedly is called recursive function.

```
Ex:   a ( formal parameters)
      {
            ---------
            ---------
            a (arguments);
            ---------
      }
```

**Note:** The important requirements or constraints that every recursive function should satisfy are

- ✓ Every time a function is called, it should be nearer to the solution.
- ✓ There should be at least one statement to terminate recursion.

The recursive function calls are initially pushed on to the stack until the terminating condition is met. After encountering the terminating condition the recursive functions are popped from the stack and executed.

**Dr.Mahesh G** Assoc. Prof. BMSIT & M
**Dr.Harish G** Assoc. Prof. Dr. AIT

**Indirect Recursion**

A recursive function need not call itself directly. Rather, it may contain a call to another function which eventually calls the first function and both are termed recursive.

```
Ex:   a ( formal parameters)              b ( formal parameters)
      {                                   {
            ---------                           ---------
            ---------                           ----------
            b (arguments);                      a (arguments);
            ---------                           ---------
      }                                   }
```

Here function 'a' calls 'b', which may in turn call 'a', which may again call 'b'. Thus both 'a' and 'b' are recursive, since they indirectly call on themselves.

However the fact that they are recursive is not evident from examining the body of either of the routines individually.

**Example 1: Factorial Function**

Given a positive integer 'n', n! is defined as the product of all integers between 'n' and 1. The exclamation mark (!) is often used to denote the factorial function.

Ex: 5! = 5x4x3x2x1

3! = 3x2x1

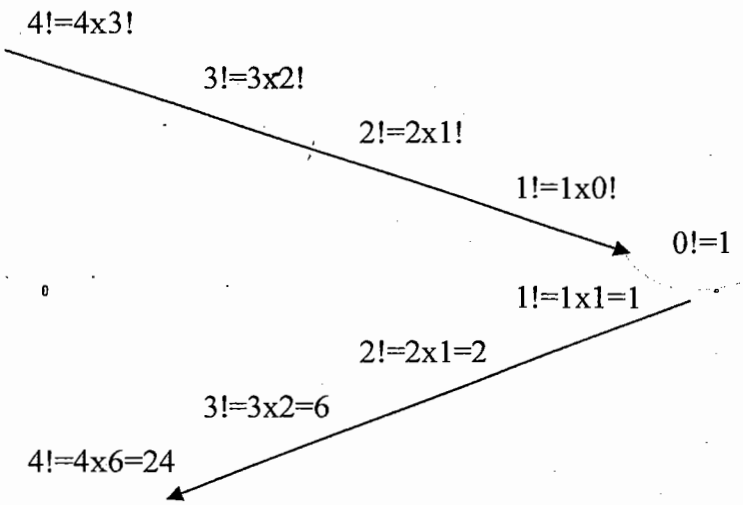
0! = 1

The recursive definition to find the factorial of 'n'

$$fact(n) = \begin{cases} 1 & if(n == 0) \\ n * fact(n-1) & otherwise \end{cases}$$

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

**Ex: To compute 4!**




**Recursive Program to find the factorial of a number**

/*Function to find factorial of a number*/

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

void main( )
{
        int n, result;
        printf("enter the value of n\n");
        scanf("%d", &n);
        result = fact(n);
        printf("factorial of %d = %d\n", n, result);
}
```

**Example 2: Fibonacci Sequence**

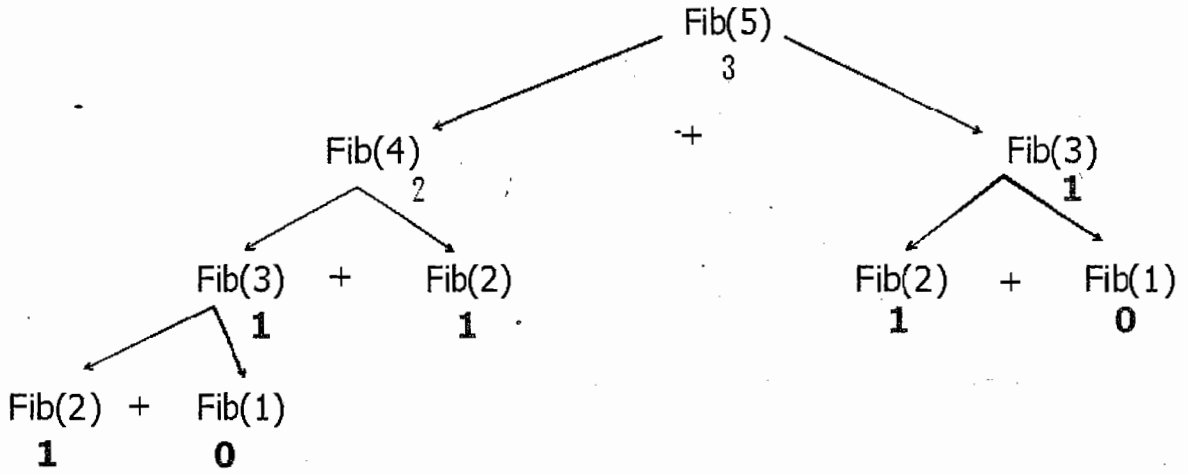
The Fibonacci sequence is the sequence of integers 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, .....

The first two numbers 0 and 1 are the initial values. The $n^{th}$ Fibonacci number is the sum of the two immediate preceding elements.

The recursive definition to find the $n^{th}$ Fibonacci number is given by

$$\text{Fib(n)} = \begin{cases} 0 & if\,(n == 1) \\ 1 & if\,(n == 2) \\ Fib(n-1) + Fib(n-2) & otherwise \end{cases}$$

The **recursive tree** for fib(5) is as shown below




**Recursive Program for Fibonacci number**

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

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

**Main Function to find the $n^{th}$ Fibonacci number**

```
void main( )
{
        int n, result;
        printf("enter the value of n\n");
        scanf("%d", &n);
        result = fib(n);
        printf(" %d fibonacci number = %d\n", n, result);
}
```

**Main Function to generate 'n' Fibonacci numbers** (sequence)

```
void main( )
{
        int n, i;
        printf("enter the value of n\n");
        scanf("%d", &n);
        printf("The fibonacci numbers are\n");
        for(i=1; i<=n; i++)
        printf(" %d",fib(i));
}
```

**Example 3: GCD (Greatest Common Divisor)**

The GCD of two numbers 'm' and 'n' denoted by GCD(m,n) is defined as the largest integer that divides both 'm' and 'n' such that the remainder is zero.

**Example:** To find GCD(10, 30)

The numbers 1,2,5 and 10 can divide 10.

The numbers 1,2,3,5,6,10, 15 and 30 can divide 30.

The common divisors are 1,2,5 and 10. The Greatest Common Divisor (GCD) is 10

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

The recursive definition to find GCD of two numbers 'm' and 'n' is given by

$$GCD(m,n) = \begin{cases} m & if(m == n) \\ GCD(m-n,n) & if(m > n) \\ GCD(m,n-m) & if(m < n) \end{cases}$$

m = 10 n = 30
m ≠ n 10 ≠ 30
10 < 30 (10, 20)
again compare GCD (10, 10)
10 = = 10
m = 10

**Recursive Program to find GCD of two numbers 'm' and 'n'**

```
/*Function to find GCD of 2 numbers*/
int GCD(int m, int n)
{
        if(m= = n)
                return m;
        if(m > n)
                return GCD(m-n, n);
        if(m < n)
                return GCD(m, n-m);

}
void main( )
{
        int m, n, result;
        printf("enter the values of m and n\n");
        scanf("%d%d", &m,&n);
        result = GCD(m,n);
        printf("GCD of %d and %d = %d\n", m, n, result);
}
```

27/9/18

**Example 4: Binomial Coefficient (nCr)**

Binomial coefficients are a family of positive integers that occur as coefficients in the binomial theorem. They are indexed by two non negative integers called the binomial coefficient indexed by 'n' and 'r'. It is the coefficient of the $X^r$ term in the polynomial expansion of the binomial power $(1+X)^n$.

In mathematics, it is denoted as nCr and is given by n! / (n-r)! * r!

The recursive definition to find nCr is given by

$$C(n, r) = \begin{cases} 1 & if(n == 1 \| n == r) \\ C(n-1, r-1) + C(n-1, r) & otherwise \end{cases}$$

**Recursive Program to find nCr**

```
/*Function to find nCr*/
int C(int n, int r)
{
        if(n = = 1 || n = = r)
                return 1;
        return C(n-1, r-1) + C(n-1, r);
}
void main( )
{
        int n, r, result;
        printf("enter the values of n and r\n");
        scanf("%d%d", &n,&r);
        result = C(n, r);
        printf("%d C %d = %d\n", n, r, result);
}
```

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

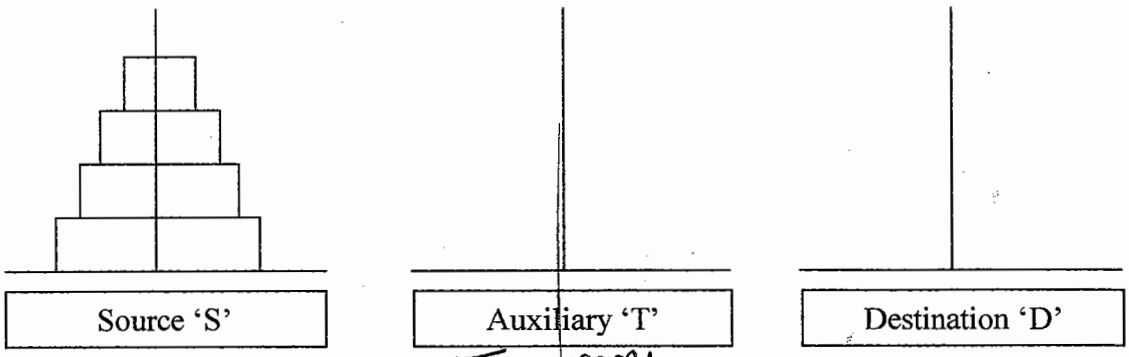
**Alternative way using the mathematical definition nCr = n! / (n-r)! * r!**

```
/*Function to find factorial of a number*/
int fact(int n)
{
        if(n= = 0)
                return 1;
        return(n*fact(n-1));
}

void main( )
{
        int n, r, result;
        printf("enter the values of n and r\n");
        scanf("%d%d", &n,&r);
        result = fact(n) / fact(n-r) * fact(r);
        printf("%d C %d = %d\n", n, r, result);
}
```

Part of lab Prog

**Example 5**
**The Towers of Hanoi Problem**

The initial set up of the towers of Hanoi problem is as shown below.



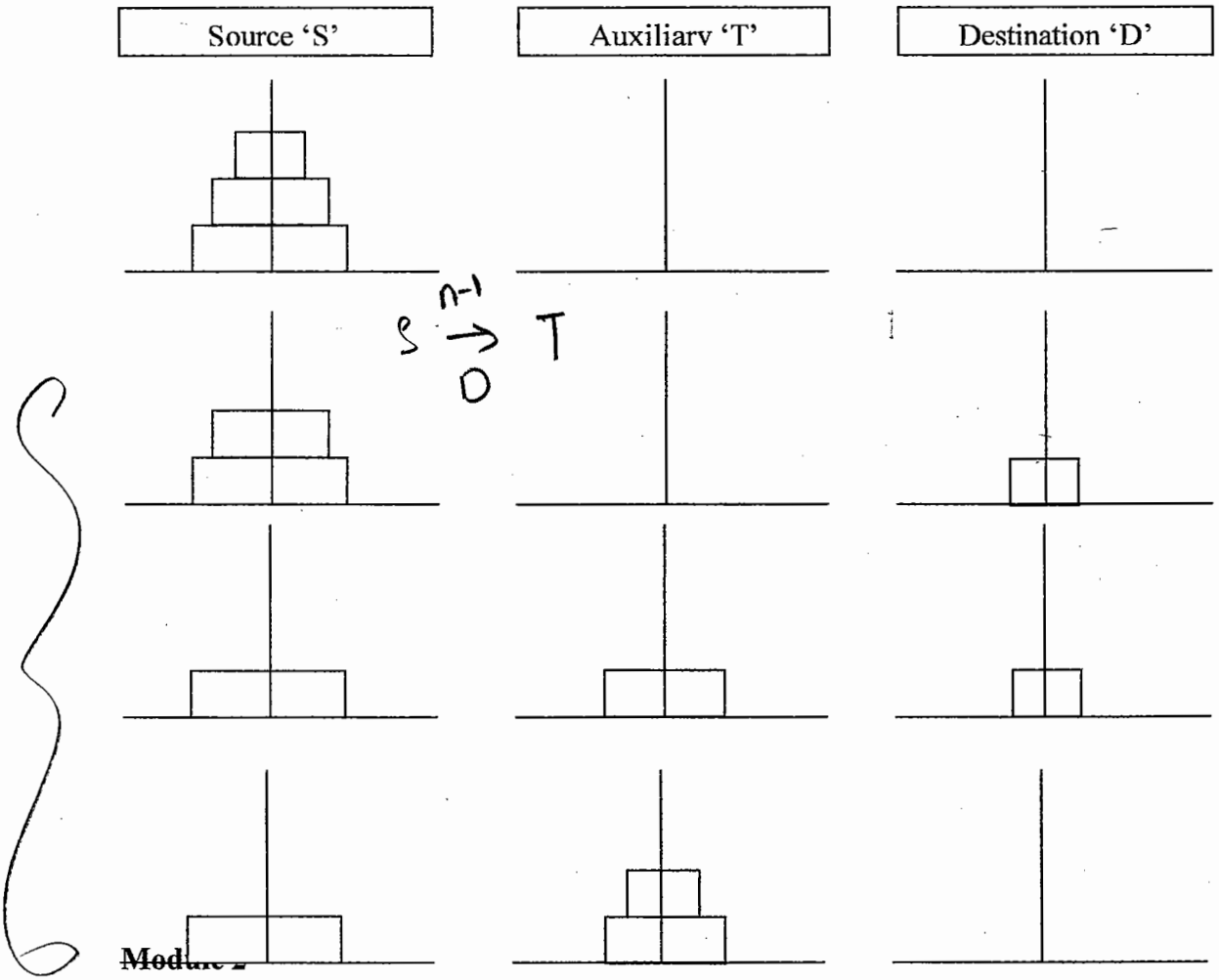

Temporary.

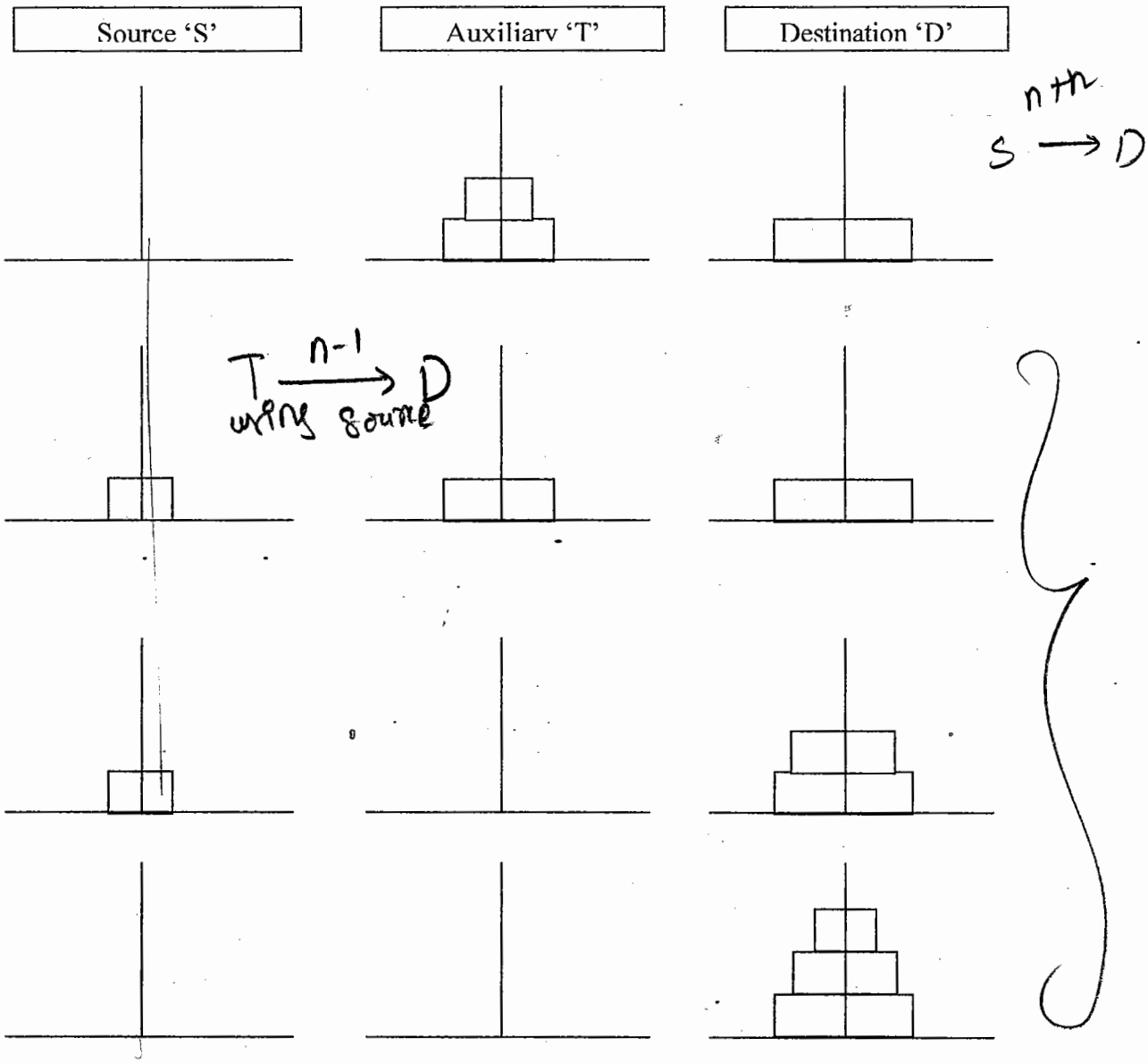
Three pegs S, T and D exists. There will be 'n' disks (in general) on Peg 'S' of different diameters placed such that a larger disk is always below a smaller disk. The objective is to move 'n' disks to peg 'D', using peg 'T' as auxiliary. The rules to be followed are

1. Only one disk is to be moved at a time.
2. Only the top disk on any peg may be moved to any other peg.
3. A larger disk may never rest on a smaller one.

**Dr.Mahesh G** Assoc. Prof. BMSIT & M
**Dr.Harish G** Assoc. Prof. Dr. AIT

**Example of 3 Disks on Source**






A recursive solution to the towers of Hanoi problem is as follows.

To move 'n' disks from S to D, using T as auxiliary,

Step 1: If n==1, move the single disk from S to D and stop (trivial case). If 1 disk is there, directly move S → D

Step 2: Recursively move the top n-1 disks from S to T, Using D as auxiliary.

Step 3: Move the $n^{th}$ disk from S to D.

Step 4: Recursively move the n-1 disks from T to D using S as auxiliary.

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

/*Recursive Program for Towers of Hanoi Problem*/

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

int count = 0;
```

```
void tower(int n, char source, char destination, char temp)
{
        if(n = = 1)
        {
                printf("move %d disc from %c to %c\n",n,source,destination);
                count++;
                return;
        }
        tower(n-1, source, temp, destination);
        printf("move %d disc from %c to %c\n",n,source,destination);
        count++;
        tower(n-1, temp, destination, source);
}
void main( )
{
        int n;
        clrscr( );
        printf("enter the number of discs\n");
        scanf("%d",&n);
        tower(n,'S','D','T');
        printf("the total number of disc moves=%d",count);
        getch( );
}
```

**Example 6**

**Ackerman's Function**

In computability theory, the Ackermann function, named after Wilhelm Ackermann, is one of the simplest and earliest-discovered examples of a total computable function. The Ackermann function is a classic recursive example in computer science. It is a function that grows very quickly (in its value and in the size of its call tree). It is defined as follows:

$$A(m,n) = \begin{cases} n+1 & if\ (m = 0) \\ A(m-1,1) & if\ (n = 0) \\ A(m-1,(A(m,n-1))) & otherwise \end{cases}$$

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

To see how the Ackermann function grows so quickly, it helps to expand out some simple expressions using the rules in the original definition. For example, we can fully evaluate $A(1,2)$ in the following way:

$$\begin{aligned} A(1,2) &= A(0,A(1,1)) \\ &= A(0,A(0,A(1,0))) \\ &= A(0,A(0,A(0,1))) \\ &= A(0,A(0,2)) \\ &= A(0,3) \\ &= 4. \end{aligned}$$

To demonstrate how $A(4,3)$'s computation results in many steps and in a large number:

$$
\begin{aligned}
A(4,3) &= A(3,A(4,2)) \\
&= A(3,A(3,A(4,1))) \\
&= A(3,A(3,A(3,A(4,0)))) \\
&= A(3,A(3,A(3,A(3,1)))) \\
&= A(3,A(3,A(3,A(2,A(3,0))))) \\
&= A(3,A(3,A(3,A(2,A(2,1))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(2,0)))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(1,1)))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,A(1,0))))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,A(0,1))))))) \\
&= A(3,A(3,A(3,A(2,A(1,A(0,2)))))) \\
&= A(3,A(3,A(3,A(2,A(1,3))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(1,2)))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(1,1))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,A(1,0)))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,A(0,1)))))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,A(0,2)))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,3)))))) \\
&= A(3,A(3,A(3,A(2,A(0,A(0,4)))))) \\
&= A(3,A(3,A(3,A(2,5)))) \\
&= \ldots \\
&= A(3,A(3,A(3,13))) \\
&= \ldots \\
&= A(3,A(3,65533)) \\
&= \ldots \\
&= A(3,2^{65536}-3) \\
&= \ldots \\
&= 2^{2^{65536}}-3.
\end{aligned}
$$

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

A+1 m=0
m-1, 1 n=0

**Recursive Program for Ackerman's Function**

```
int ackerman(int m, int n)
{
        if(m= = 0)
                return n+1;
        if(n= = 0)
                return (ackerman(m-1,1));
        return(ackerman(m-1, (ackerman(m, n-1))));
}

void main( )
{
        int m, n, result;
        printf("enter the value of m\n");
        scanf("%d", &m);
        printf("enter the value of n\n");
        scanf("%d", &n);
        result = ackerman(m, n)
        printf("ackerman(%d, %d) = %d\n", m,  n, result);
}
```

A (m, n)

Dr.Mahesh G Assoc. Prof. BMSIT & M
Dr.Harish G Assoc. Prof. Dr. AIT

Queues

**Basic Operations**

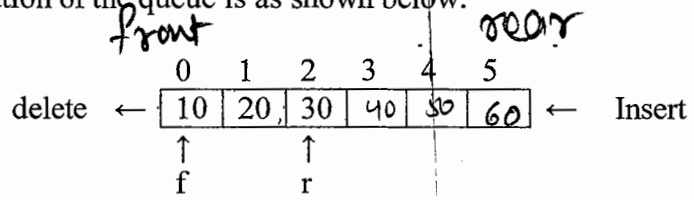
A queue is basically a waiting line, for example, a line of customers in front of a counter in a bank or a ticket counter. Here,

- ✓ Customers join the queue at the end of line (rear or tail).
- ✓ Customer who is at the front gets the service and leaves the line (front or head).

This indicates that a customer gets service in FCFS (First Come First Serve) or FIFO (First In First Out) basis.

As a data structure, a queue is an ordered list of elements from which an element may be removed at one end called front of the queue, and into which an element may be inserted at the other end called the rear of the queue. These restrictions ensure that data are processed through the queue in the order in which they are received. So a queue is also called FIFO data structure.

Pictorial representation of the queue is as shown below.




The various types of queues are

- ✓ Ordinary queue
- ✓ Double ended queue
- ✓ Circular queue
- ✓ Priority queue



**Queue Abstract Datatype**

**Objects:** A finite ordered list with zero or more elements
**Functions:** For all q Є Queue, item Є element, queuesize Є positive integer

*Queue* Create(queuesize) ::= Creates an empty queue whose maximum size is queuesize

*Boolean* IsFull(q, *queuesize*) := **if** number of elements in queue q is queuesize **return** *TRUE*
**else return** *FALSE*

*Queue* InsertQ*(q, item)* ::= **if** (IsFull(q)) queue is full
**else** insert item on to the queue q and return

*Boolean* IsEmpty(q) ::= **if** no elements are there in queue **return** *TRUE*
**else return** *FALSE*

*Element* DeleteQ(q) ::= **if** (IsEmpty(q)) **return**
**else** remove and return the element at the front of the queue

**Ordinary Queue or Linear Queue or Queue**

This queue operates on a first come first serve basis. Items will be inserted from one end and they are deleted at the other end in the same order in which they are inserted.

The various operations that can be performed on a ordinary queue are

1. Insert Rear – Here elements are inserted from rear end of the queue.
2. Delete Front – Here elements are deleted from the front end.
3. Display – Here contents of queue are displayed.

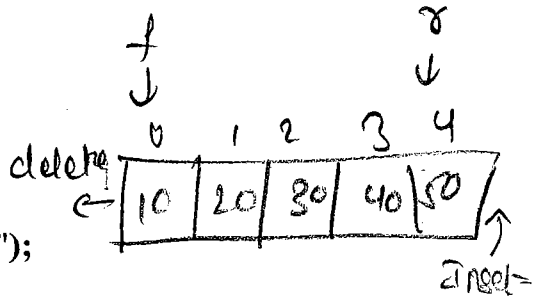
**Observations**

- ✓ Initially when there are no elements f = 0 and r = -1
- ✓ Whenever f > r there are no elements in the queue i.e. the queue is empty.
- ✓ f = r means there is only one element in the queue.

**Implementation of Queues using arrays**

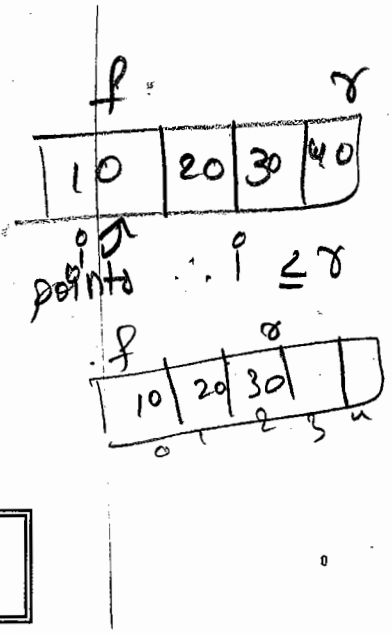
```
#include<stdio.h>
#define queuesize 5

void insert_rear(int item, int *r, int q[])
{
        if(*r = = queuesize-1)
        {
                printf("queue overflow\n");
                return;
        }
        *r = *r + 1;
        q[*r] = item;
}

void delete_front(int *f, int *r, int q[])
{
        int item;
        if(*f > *r)
        {
                printf("queue is empty\n");
                return;
        }
        item = q[*f];
        printf("\nthe item deleted is %d",item);
        *f = *f+1;

        if(*f > *r)
        {
                *f = 0;
                *r = -1;
        }
}
```





```
void display(int f, int r, int q[])
{
        int i;
        if(f > r)
        {
                printf("queue is empty\n");
                return;
        }
        printf("the contents of the queue are\n");
        for(i=f; i<=r; i++)
        {
                printf("%d\n",q[i]);
        }
}

void main( )
{
        int f, r, choice, item, q[30];
        clrscr( );
        f = 0;
        r = -1;
        for(;;)
        {
                printf("\n1:insert_rear\n2:delete_front\n");
                printf("3:display\n4: exit\n");
                printf("enter your choice\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: printf("enter the item to be inserted\n");
                                scanf("%d",&item);
                                insert_rear(item, &r, q);
                                break;

                        case 2: delete_front(&f, &r, q);
                                break;

                        case 3: display(f, r, q);
                                break;

                        case 4: exit(0);
                }
                getch( );
        }
}
```

Same as delete

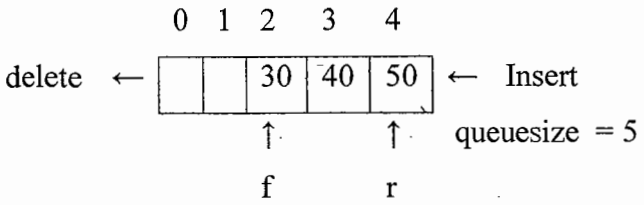



| Dr.Mahesh G | Dr.Harish G |
|---|---|
| Assoc. Prof. | Assoc. Prof. |
| BMSIT & M | Dr. AIT |

29/9/08

**Disadvantages of ordinary queues**

The disadvantage of ordinary queue is that once rear reaches (queuesize -1 ) it is not possible to insert any element into the queue even though the queue is not full. This situation is depicted in the following figure.




Even though there are 2 free locations, since the value of r has reached queuesize – 1, we will not be able to insert rear.

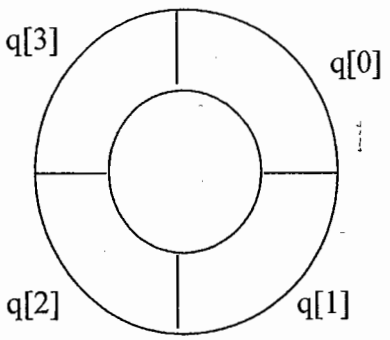
**Solution 1:** Shift data to previous locations such that the elements start from the beginning of the queue. This method proves to be costly in terms of computer time if the data being stored is very large.

**Solution 2:** Use circular queue to overcome this situation.

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

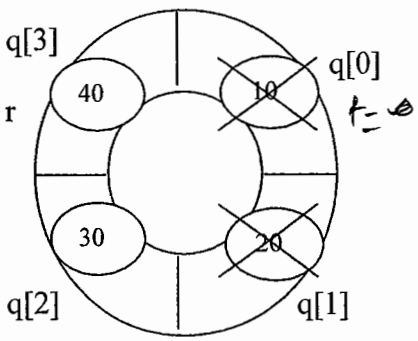
**Circular Queue**

As the name suggests, this queue is not straight but circular; meaning, the last element of the array is logically followed by the first element of the array (element with index 0). It has a structure as shown below




// Initially r = -1 and f = 0

After inserting 4 elements and then deleting 2 elements we have the queue structure as shown below.



If we try to insert an element to the queue we follow 2 steps

- ✓ Increment rear pointer.
- ✓ Insert item to q[ ] pointed by rear.

Since this is a circular queue, once rear pointer reaches the end position the next one will be the first position. To attain this we use the statement r = ( r + 1 ) % queuesize.

EX: ( 3 + 1 ) % 4 = 4 % 4 = 0 (first position)

If this approach is used, to check for queue overflow and underflow, we use a variable called as 'count'. Count contains the number of items or elements present in the queue at any instant. Therefore, count = 0 means the queue is empty and count = queuesize means the queue is full.

**Note:**

1. In a queue (circular or linear), if we want to insert we follow insert rear and if we want to delete we follow delete front.
2. When an item is inserted and if rear points to the last position, we can insert one more item if and only if the queue does not contain any item in the first position, otherwise it will be queue overflow state.
3. For a circular queue to
   - ✓ Insert we follow

     r = ( r + 1 ) % queuesize

     q[r] = item

     count = count + 1

   - ✓ Delete we follow

     item = q[f]

     f = ( f + 1 ) % queuesize

     count = count – 1

   - ✓ Queue empty and full we follow

     count = 0 for queue is empty

     count = queuesize for queue is full

**Implementation of circular queues using arrays**

```
#include<stdio.h>
#define queuesize 5
void insert_rear(int item, int *r, int *count, int q[])
{
        if(*count = = queuesize)
        {
                printf("queue overflow\n");
                return;
        }
        *r = (*r+1) % queuesize;
        q[*r] = item;
        *count = *count + 1;
}

void delete_front(int *f, int *count, int q[])
{
        int item;
        if(*count = = 0)
        {
                printf("queue is empty\n");
                return;
        }
        item = q[*f];
        printf("\nthe item deleted is %d",item);
        *f = (*f+1) % queuesize;
        *count = *count - 1;
}

void display(int f, int count, int q[])
{
        int i;
        if(count = = 0)
        {
                printf("queue is empty\n");
                return;
        }
        printf("the contents of the queue are\n");
        for(i=1;i<=count;i++)
        {
                printf("%d\n",q[f]);
                f = (f+1) % queuesize;
        }
}
```



due to circular queue
f > r
f = 0
r = -1
is not used.
only used in linear queue

it prints queue from front.

```
void main()
{
        int f, r, choice, item, q[5], count;
        clrscr( );
        f = 0;
        r = -1;
        count = 0;
        for(;;)
        {
                printf("\n1:insert_rear\n2:delete_front\n");
                printf("3:display\n4:exit\n");
                printf("enter your choice\n");
                scanf("%d",&choice);
                switch(choice)
                {
                        case 1: printf("enter the item to be inserted\n");
                                scanf("%d",&item);
                                insert_rear(item, &r, &count, q);
                                break;

                        case 2: delete_front(&f, &count, q);
                                break;

                        case 3: display(f, count, q);
                                break;

                        case 4: exit(0);
                }
                getch( );
        }
}
```



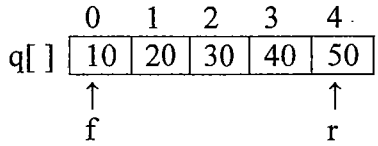
**Applications of Queues**

- ✓ One major application of queue is in simulation which is studied as queuing theory in operation research. The main objective of this simulation is to study the real life situation (problem) under the control of various parameters (waiting time, input flow rate, servicing time etc) which affects the queuing problem.
- ✓ Queues are used in time sharing systems in which processes form a queue while waiting to be executed.
- ✓ They are used in network communication systems.
- ✓ They are used in operating system to handle i/o request, while waiting for a specific device to get the service.

1/10/18

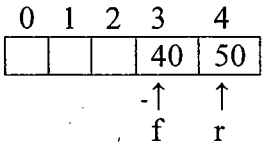
## Circular Queues using Dynamic Arrays (with the concept of array doubling)

In this case just doubling the array size is not sufficient because as ordering of elements and re-initialization of front and rear pointers need to be done. For example consider a circular queue with queue size 5, after inserting 5 elements the structure is as shown below.



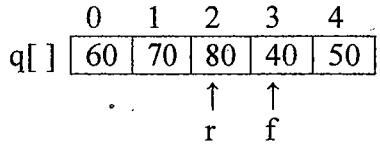

count = queuesize = 5
Queue is full

After deleting 3 elements we obtain the following structure,





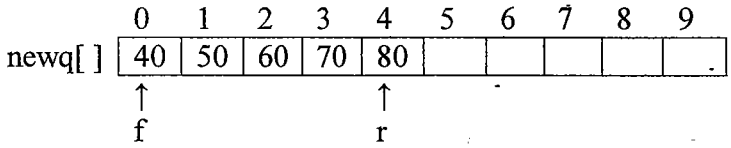

Now insert 3 elements into the queue we get the following structure.




count = queuesize = 5
Queue is full

At this point, just doubling the array size is not sufficient, the elements need to be copied in correct order, and the front and rear pointers need be re-initialized appropriately.

use capacity instead of capacity here




```
#include<stdio.h>
int capacity = 5;
int *q;

void delete_front(int *f, int *count)
{
        int item;
        if(*count = = 0)
        {
                printf("queue is empty\n");
                return;
        }
        item = q[*f];
        printf("\nthe item deleted is %d",item);
        *f = (*f+1) % capacity;
        *count = *count - 1;
}
```

```
void insert_rear(int item, int *r, int *count, int *f)
{
        int i;
        int *newq;  // to create new queue

        if(*count = = capacity)
        {
            // Allocate memory to double the array capacity
             newq = (int*) malloc( 2 * capacity * sizeof(int));

            // Copy the elements from old queue to new queue
            for(i=0;i<count;i++)
            {
                    newq[i] = q[*f];
                    *f = (*f+1) % capacity;
            }

            // Reinitialize pointers
            *f = 0;
            *r = *count – 1;
            capacity = 2 * capacity;
            free(q);
            q = newq;
        }

        *r = (*r+1) % capacity;
        q[*r] = item;
        *count = *count + 1;
}

void display(int f, int count)
{
        int i;

        if(count = = 0)
        {
            printf("queue is empty\n");
            return;
        }
        printf("the contents of the queue are\n");
        for(i=1;i<=count;i++)
        {
            printf("%d\n",q[f]);
            f = (f+1) % capacity;
        }
}
```

```
void main()
{
        int f, r, choice, item, count;
        q = (int *) malloc(capacity*sizeof(int));

        clrscr( );
        f = 0;
        r = -1;
        count = 0;
        for(;;)
        {
            printf("\n1:insert_rear\n2:delete_front\n");
            printf("3:display\n4:exit\n");
            printf("enter your choice\n");
            scanf("%d",&choice);
            switch(choice)
            {
                    case 1: printf("enter the item to be inserted\n");
                            scanf("%d",&item);
                            insert_rear(item, &r, &count, &f);
                            break;

                    case 2: delete_front(&f, &count);
                            break;

                    case 3: display(f, count);
                            break;

                    case 4: free(q);
                            exit(0);
            }
            getch( );
        }
}
```

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

## De-queue (Double Ended Queue)

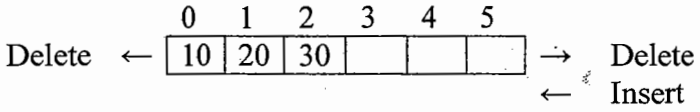
It is a data structure in which insertions and deletions will be done either at front end and / or at the rear end of the queue i.e. insertions and deletions is possible at either ends.

There are 2 variations of De-queue,

**1) Input Restricted De-queue**

An Input Restricted De-queue is one which allows the insertions at only end but allows deletion at both the ends.

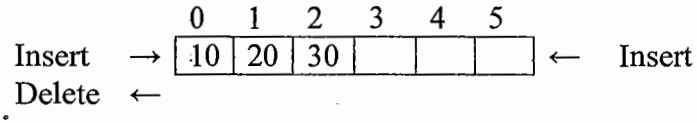
**Example:** Insertion possible only at rear end




**2) Output Restricted De-queue**

An Output Restricted De-queue is one which allows the deletion at only one end, but allows insertions at both ends.

**Example:** Deletion possible only at front end




The operations that can be performed on a De-queue are

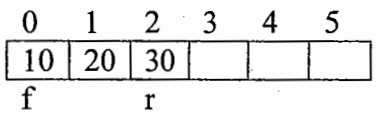
1) Insert Front
2) Insert Rear
3) Delete Front
4) Delete Rear
5) Display

**Dr.Mahesh G Dr.Harish G**
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT

**Note:** insert_rear( ), delete_front( ) and display( ) functions same as that of ordinary queue.

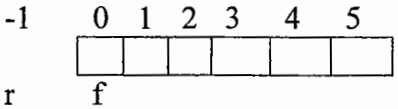
**Function to insert an element at the front end**

**Case 1:**




Insertion Not Possible
f = 0 and r != -1

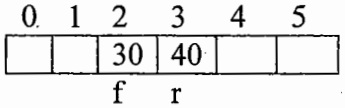
**Case 2:**




Insertion Possible
r = r + 1
q[r] = item

f=0 & r = -1

**Case 3:**




Insertion Possible
f = f - 1
q[f] = item

f != 0 & r != 0

```
void insert_front(int item, int *f, int *r, int q[])
{
        if(*f ==0 && *r != -1)
        {
                printf("Insert front not possible\n");
                return;
        }

        if(*f == 0)
        {
                *r = *r + 1;
                q[*r] = item;
                return;
        }

        *f = *f -1;
        q[*f] = item;
}
```

**Function to delete an element at the rear end**

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

```
void delete_rear(int *f, int *r, int q[])
{
        int item;
        if(*f > *r)
        {
                printf("queue is empty\n");
                return;
        }

        item = q[*r];
        printf("\nthe item deleted is %d",item);
        *r = *r - 1;

        if(*f > *r)
        {
                *f = 0;
                *r = -1;
        }
}
```

## Priority Queues

In linear and circular queues the operations are performed according to FIFO approach. Sometimes, we may need to process items according to their importance in their application. In such a situation, priority queues can be used.

Priority queue is a queue in which each element of the queue will have a priority and the elements are processed (insertion and deletion) according to the following rules.

1) An element of higher priority is processed before any element of lower priority

2) If 2 elements have same priority, then the element which was inserted first into the queue will be processed first.

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

**Types of Priority Queues**

**1) Ascending Priority Queue**

It is a collection elements where elements can be inserted in any order but while deleting, the smallest element is to be removed from the queue.

**2) Descending Priority Queue**

It is a collection elements where elements can be inserted in any order but while deleting, the largest element is to be removed from the queue.

**Priority Queue Implementation**

Two methods have been described below for implementing priority queue using arrays. Here items to be inserted itself denotes the priority.

It is assumed that smaller value element has higher priority and larger value element has smaller priority i.e. 3 has higher priority when compared to 5.

**Method - 1**

Here the aim is to

- Insert elements in any order
- Delete only the smallest element (High priority element)

To do descendly → Delete larger element

The deletion signifies that it is an ascending priority queue and is performed as shown below

**Step 1:** Starting from front, traverse the array to find the smallest element (High priority element) and its position.

**Step 2:** Delete this element from the queue.

**Step 3:** If necessary (deleted element is not the right most), shift all its trailing element to left side by one position after the deleted item to fill the vacant position and update the rear pointer.

**Function to delete the smallest element for implementing priority queue**

```
void delete_small(int *f, int *r, int q[])
{
        int item, i, min, pos;
        if(*f > *r)
        {
                printf("queue is empty\n");
                return;
        }
      if(*f = = *r)
      {
          item = q[*r];
          printf("\nthe item deleted is %d",item);
          *f = 0;
          *r = -1;
      }
      else
      {
          min = q[0];
          pos = 0;
          for(i=1; i<= *r; i++)
          {
                if(q[i] < min)
                {
                        min = q[i];
                        pos = i;
                }
          }
          item = q[pos];
          printf("\nthe item deleted is %d",item);
          for(i=pos; i<*r; i++)
          {
                q[i] = q[i+1];
          }
          *r = *r - 1;
      }
}
```



**Note:** The other functions i.e insert_rear( ), display( ) and main( ) remains same as that of ordinary queue.

**Disadvantage**

It is very inefficient as it involves searching the queue for smaller element and shifting the trailing elements after deletion.

**Method - 2**

Here the aim is to

- Insert elements into queue such that they are always ordered in increasing order (ascending order) i.e. high priority elements are at the front end always.

  **Step 1:** Find an appropriate vacant position to insert based on item priority.

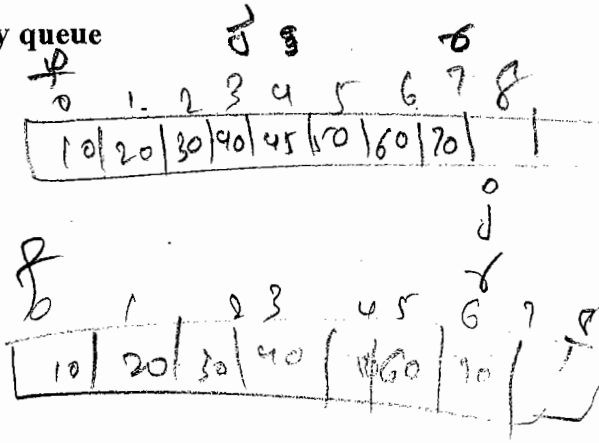
  **Step 2:** Insert the item at that position.

  **Step 3:** Update the rear pointer.

- Simply delete an element from the front end since the smallest element will always be in the front.

**Function to insert an element for implementing priority queue**

```
void insert_order(int item, int *f, int *r, int q[])
{
        int j;
        if(*r == queuesize-1)
         {
                printf("queue is full\n");
                return;
         }
         j = *r;
        while( j >= *f && item < q[j])
        {
                q[j+1] = q[j];
                j = j-1;
        }
        q[j+1] = item;
        *r = *r + 1;
}
```





**Note:** The other functions i.e delete_front( ), display( ) and main( ) remains same as that of ordinary queue.

**Disadvantage**

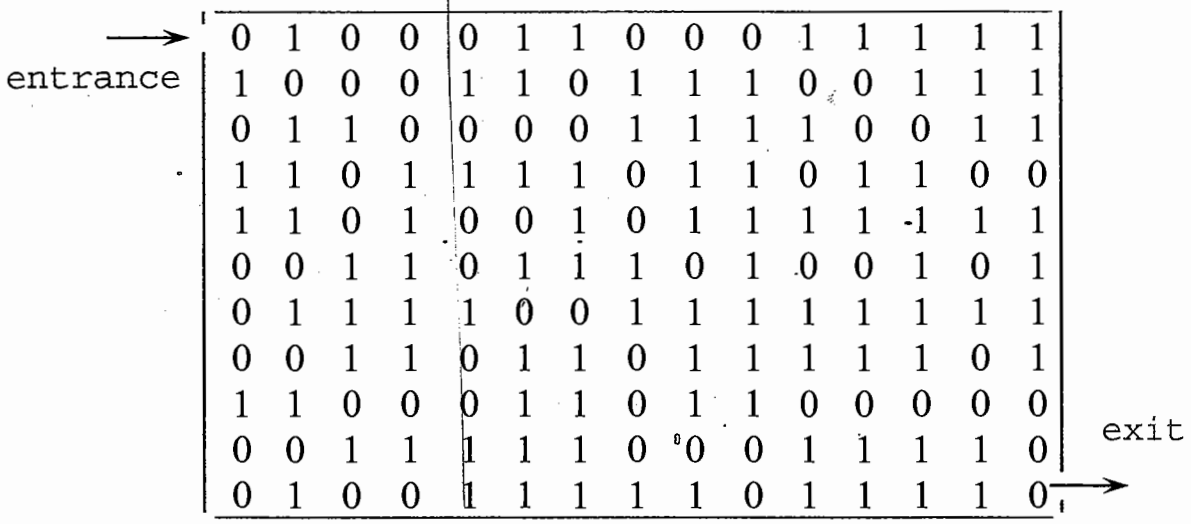
- Insertion requires locating the proper position for the new element and also requires shifting of elements.
- Insertion needs to do compaction, when it runs out of memory (i.e. space left but cannot insert).

**Note:** In general, arrays are not efficient for implementing priority queues.

## A Mazing Problem

A maze is a complex structure with a series of interconnecting pathways. The term is also used to refer to a graphical puzzle which replicates the maze on a two dimensional medium. A maze is viewed as a puzzle which must be solved, and the solver must work his or her way from the entrance of the maze to an exit, or another location.

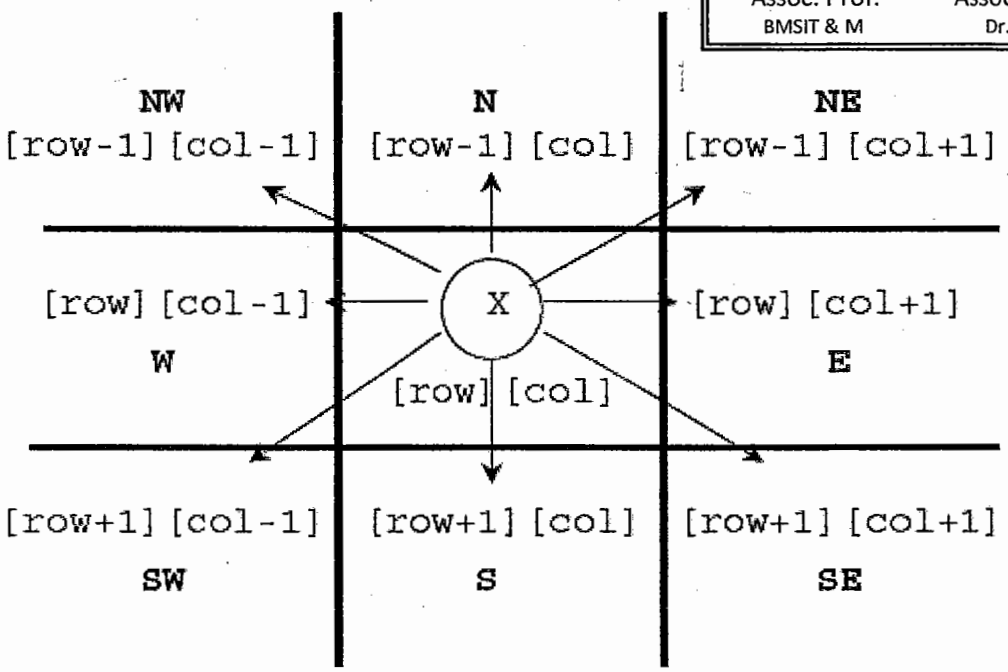
The following shows a maze in which zeros represent open paths and ones represent the barriers. The problem is to find the path from entrance to exit.




| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | -1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |

This maze can be represented in the form of a 2-dimensional array "maze[row][col]". Given a location in the maze, there are 8 possible directions along which we can move, north, north-east, east, south-east, south, south-west, west and north-west. We use the following compass to specify the eight directions of movements.

Dr.Mahesh G Dr.Harish G
Assoc. Prof. BMSIT & M
Assoc. Prof. Dr. AIT



**Note:** Not every position in the maze has eight neighbours. If [row, col] is on a border then less than eight, and possibly three, neighbours exist. To avoid checking for these border conditions we can surround the maze by a border of ones. Thus an m X n maze will require an (m+2) X (n+2) array. The entrance is at position [1][1] and the exit is at [m][n].

We predefine the eight possible directions to travel in the array 'move' represented by index numbers from 0 to 7. For each direction, we indicate vertical and horizontal offset. This is done using the following structure.

```
struct offsets
{
        int vertical;
        int horizontal;
};
```

The table of moves for the 8 possible directions is as shown below.

| Name | Dir | move[dir].vertical | move[dir].horizontal |
|---|---|---|---|
| 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 |

We define an array of structure 'move' and can be initialized as
struct offsets move[8] = {{-1,0}, {-1, 1}, {0,1}, {1,1}, {1,0}, {1,-1}, {0,-1}, {-1,-1}};

To move from the current position maze[row][col] to the next valid position in the maze at a particular direction we use,

nextrow = row + move[dir].vertical;

nextcol = col + move[dir].horizontal;



**Note:**

- ✓ Use of Stacks – As we travel through the maze, there will be a choice of several directions of movement. Since the best choice is not known, we save the current position and arbitrarily pick a possible move. If we are stuck in a hopeless path, we can get back to the current position using stack and tryout another possible path.
- ✓ Using stack, we wish not to get into a previously tried path, therefore a 2d-array 'mark' is defined to record the maze positions already checked. Initially all entries of this array is set to zero. When we visit a position, maze[row][col], we change mark[row][col] to one.

- ✓ EXITROW and EXITCOL gives the coordinates of the maze exit.
- ✓ In this case a stack record need to contain three information i.e. row, col and dir. Hence we define the following structure for the stack.

```
struct stack
{
        int row;
        int col;
        int dir;
};
struct stack s[100];
```

## C Program for Maze Problem

```
#include <stdio.h>

struct offsets
{
        int vertical;
        int horizontal;
};
struct offsets move[8] = {{-1,0}, {-1, 1}, {0,1}, {1,1}, {1,0}, {1,-1}, {0,-1}, {-1,-1}};

struct stack
{
        int row;
        int col;
        int dir;
};

#define EXITROW  5
#define EXITCOL  3

int maze[10][10] = {
                     {1,1,1,1,1},    /* top boundary  */
                     {1,0,0,0,1},
                     {1,1,1,0,1},
                     {1,0,0,0,1},
                     {1,0,1,1,1},
                     {1,0,0,0,1},
                     {1,1,1,1,1}     /* bottom boundary *
                    };
void main( )
{
        int row, col, nextrow, nextcol, dir, found, top, i;
        int mark[10][10] = {{0}};
        struct stack s[100], pos;

        found = 0;
        top = -1;

        mark[1][1] = 1;         // Start  from the initial point
        pos.row = 1;
```

```
        pos.col = 1;
        pos.dir = 0;
        s[++top] = pos;          // Save the position
        while(top!=-1 && found!=1)
        {
                pos = s[top--];          // Obtain the position to try the next possible move

                row = pos.row;           // Make this as the current position
                col = pos.col;
                dir = pos.dir;
                while(dir<8 && found!=1)
                {
                        // Find the next move
                        nextrow = row + move[dir].vertical;
                        nextcol = col + move[dir].horizontal;

                        if ((nextrow = = EXITROW) && (nextcol = = EXITCOL))
                                found = 1;
                        elseif(maze[nextrow][nextcol] == 0 && mark[nextrow][nextcol] == 0)
                        {
                                mark[nextrow][nextcol] = 1;

                                // Fix the previous position and save
                                pos.row = row;
                                pos.col = col;
                                pos.dir = ++dir;
                                s[++top] = pos;

                                // Update row and col to find the next move
                                row = nextrow;
                                col = nextcol;
                                dir = 0;
                        }
                        else
                                ++dir;           // Try in the next direction
                }
        }
        if(found==1)
        {
                printf("the path is\n");
                for(i=0; i<=top; i++)
                {
                        printf("%d   %d\n", s[i].row, s[i].col);
                }
                printf("%d   %d\n", EXITROW, EXITCOL);
        }
        else
                printf("maze does not have a path\n");
}
```

Dr.Mahesh G Dr.Harish G
Assoc. Prof. Assoc. Prof.
BMSIT & M Dr. AIT

mod-3 Linked list

\ : null

interlinked by pointers

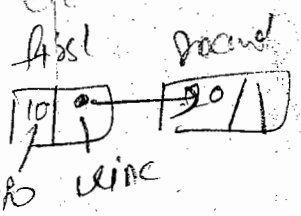
Dynamic memory alloc

LL = Dynamic

Singly LL, Singly with header LL, Doubly LL, doubly LL with header, circular LL,

first = NULL, No ele in LL

first → Link = Null, only 1 ele




struct node * = NODE

delete
Update
insert
create

Node
enter new
Time and
Interrupted