

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

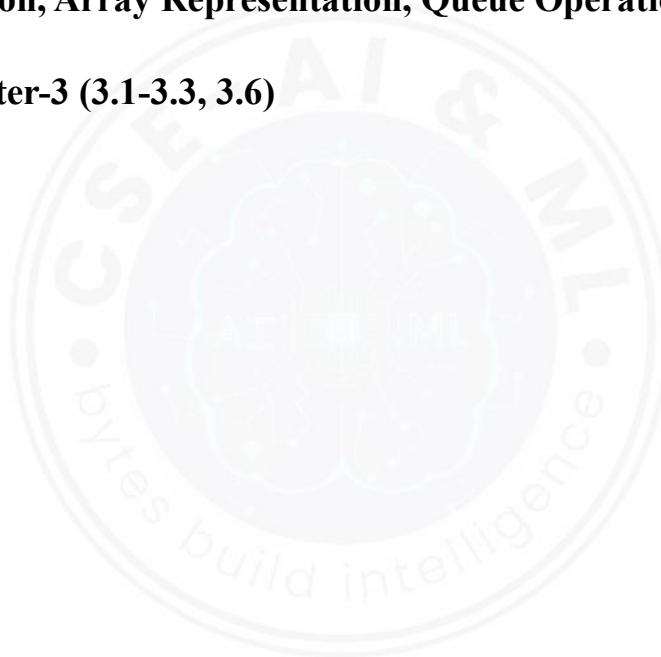
DATA STRUCTURES AND APPLICATIONS – 1BCS305

MODULE 02

Stacks: Definition, Stack Operations, Array Representation of Stacks, Stacks using Dynamic Arrays, Stack Applications: Polish notation, Infix to Postfix Conversion, Evaluation of Postfix Expression.

Queues: Definition, Array Representation, Queue Operations.

Textbook: Chapter-3 (3.1-3.3, 3.6)



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

STACKS

DEFINITION

“A **stack** is an **ordered list** in which insertions (pushes) and deletions (pops) are made at one end called the **top**.”

Given a stack $S = (a_0, \dots, a_{n-1})$, where a_0 is the bottom element, a_{n-1} is the top element, and a_i is on top of element a_{i-1} , $0 < i < n$.

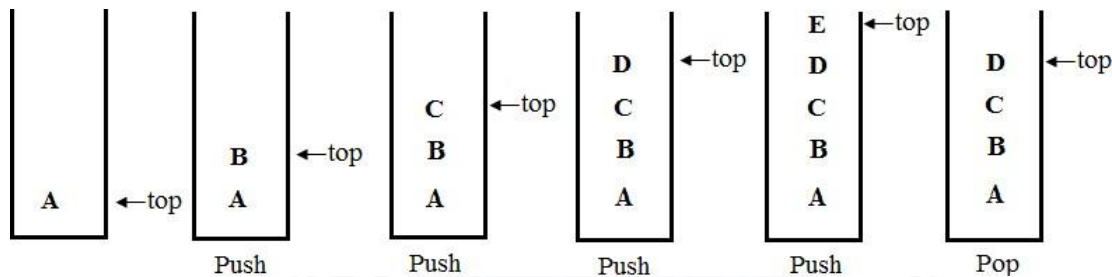


Figure: Inserting and deleting elements in a stack

As shown in above figure, the elements are added in the stack in the order A, B, C, D, E, then E is the first element that is deleted from the stack and the last element is deleted from stack is A. Figure illustrates this sequence of operations.

Since the last element inserted into a stack is the first element removed, a stack is also known as a **Last-In-First-Out (LIFO)** list.

STACK OPERATIONS

Implementation of the stack operations as follows.

1. Stack Create

Stack CreateS(maxStackSize) ::=

```
#define MAX_STACK_SIZE 100 /* maximum stack
size*/ typedef struct
{
    int key;
/*otherfields */
} element;
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
element stack[MAX_STACK_SIZE];
```

```
int top = -1;
```

The **element** which is used to insert or delete is specified as a structure that consists of only a **key** field.

2. Boolean IsEmpty(Stack)::= top < 0;

3. Boolean IsFull(Stack)::= top >= MAX_STACK_SIZE-1;

The **IsEmpty** and **IsFull** operations are simple, and is implemented directly in the program push and pop functions. Each of these functions assumes that the variables **stack** and **top** are global.

4. Push()

Function **push** checks whether stack is full. If it is, it calls stackFull(), which prints an error message and terminates execution. When the stack is not full, increment top and assign item to stack [top].

```
void push(element item)
{
    /* add an item to the global stack */
    if (top >= MAX_STACK_SIZE-1)
        stackFull();
    stack[++top] = item;
}
```

5. Pop()

Deleting an element from the stack is called pop operation. The element is deleted only from the top of the stack and only one element is deleted at a time.

```
element pop ( )
{
    /*delete and return the top element from the
    stack */ if (top == -1) return
    stackEmpty(); /*returns an error key
    */ return stack[top--]; }
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

6. `stackFull()` The **stackFull** which prints an error message and terminates execution.

ARRAY REPRESENTATION OF STACKS

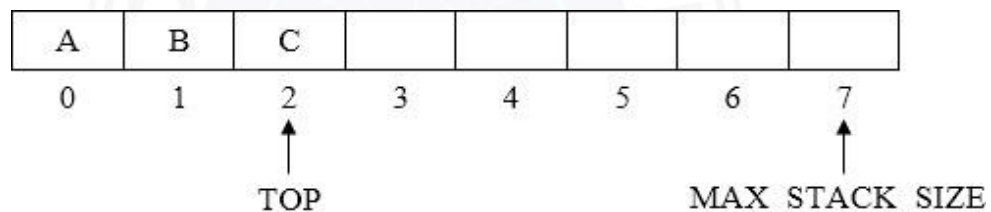
Stacks may be represented in the computer in various ways such as one-way linked list (Singly linked list) or linear array.

Stacks are maintained by the two variables such as TOP and MAX_STACK_SIZE.

TOP which contains the location of the top element in the stack. If TOP = -1, then it indicates stack is empty.

MAX_STACK_SIZE which gives maximum number of elements that can be stored in stack.

Stack can be represented using linear array as shown below



Stack Operations using dynamic array

The main stack operations are:

1. **Push** – Insert an element
2. **Pop** – Delete the top element
3. **Peek/Top** – View the top element
4. **isEmpty** – Check whether stack is empty
5. **isFull** – Check whether stack is full

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Stack full with array doubling

Analysis

In the **worst case**, the realloc function needs to allocate $2 * \text{capacity} * \text{sizeof}(*\text{stack})$ bytes of memory and copy $\text{capacity} * \text{sizeof}(*\text{stack})$ bytes of memory from the old array into the new one. Under the assumptions that memory may be allocated in $O(1)$ time and that a stack element can be copied in $O(1)$ time, the time required by array doubling is $O(\text{capacity})$. Initially, capacity is 1.

Suppose that, if all elements are pushed in stack and the capacity is 2^k for some $k, k > 0$, then the total time spent over all array doublings is $O(\sum_{i=1}^k 2^i) = O(2^{k+1}) = O(2^k)$.

Since the total number of pushes is more than $2^k - 1$, the total time spent in array doubling is $O(n)$, where n is the total number of pushes. Hence, even with the time spent on array doubling added in, the total run time of push over all n pushes is $O(n)$.

STACK APPLICATIONS: POLISH NOTATION

Expressions: It is sequence of operators and operands that reduces to a single value after evaluation is called an expression.

$$X = a / b - c + d * e - a * c$$

In above expression contains operators (+, -, /, *) operands (a, b, c, d, e).

Expression can be represented in in different format such as

- Prefix Expression or Polish notation
- Infix Expression
- Postfix Expression or Reverse Polish notation

Infix Expression: In this expression, the binary operator is placed in-between the operand. The expression can be parenthesized or un-parenthesized.

Example: $A + B$

Here, **A** & **B** are operands and + is operand

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Prefix or Polish Expression: In this expression, the operator appears before its operand.

Example: + A B

Here, A & B are operands and + is operand

Postfix or Reverse Polish Expression: In this expression, the operator appears after its operand.

Example: A B +

Here, A & B are operands and + is operand

Precedence of the operators

The first problem with understanding the meaning of expressions and statements is finding out the order in which the operations are performed.

Example: assume that a = 4, b = c = 2, d = e = 3 in below expression

X = a / b - c + d * e - a * c

$$\begin{aligned} & ((4/2)-2) + (3*3)-(4*2) \\ & = 0+9-8 \\ & = 1 \end{aligned}$$

OR

$$\begin{aligned} & (4/ (2-2 +3)) * (3-4)*2 \\ & = (4/3) * (-1) * 2 \\ & = -2.66666 \end{aligned}$$

The first answer is picked most because division is carried out before subtraction, and multiplication before addition. If we wanted the second answer, write expression differently using parentheses to change the order of evaluation **X = ((a / (b - c + d)) * (e - a) * c**

In C, there is a precedence hierarchy that determines the order in which operators are evaluated. Below figure contains the precedence hierarchy for C.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Token	Operator	Precedence	Associativity
() [] →	function call array element struct or union member	17	left-to-right
-- ++	Increment, Decrement	16	left-to-right
--++ ! ~ -+ & * sizeof	decrement, increment logical not one's complement unary minus or plus address or indirection size (in bytes)	15	right-to-left
(type)	type cast	14	right-to-left
* / %	Multiplicative	13	left-to-right
+ -	binary add or subtract	12	left-to-right
<< >>	shift	11	left-to-right
> >= < <=	relational	10	left-to-right
= = ! =	equality	9	left-to-right
&	Bitwise and	8	left-to-right
^	bitwise exclusive or	7	left-to-right
	Bitwise or	6	left-to-right
&&	logical and	5	left-to-right
	logical or	4	left-to-right
?:	conditional	3	right-to-left
= += -= /= *= % = <<= >>= &= ^= =	assignment	2	right -to-left
,	comma	1	left-to-right

- The operators are arranged from highest precedence to lowest. Operators with highest precedence are evaluated first.
- The associativity column indicates how to evaluate operators with the same precedence. For example, the multiplicative operators have left-to-right

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

- associativity. This means that the expression $a * b / c \% d / e$ is equivalent to $(((a * b) / c) \% d) / e$
- Parentheses are used to override precedence, and expressions are always evaluated from the innermost parenthesized expression first

INFIX TO POSTFIX CONVERSION

An algorithm to convert infix to a postfix expression as follows:

1. Fully parenthesize the expression.
2. Move all binary operators so that they replace their corresponding right parentheses.
3. Delete all parentheses.

Example: Infix expression: $a/b - c + d * e - a * c$

Fully parenthesized : $((((a/b) - c) + (d * e)) - a * c)$

: $a b / e - d e * + a c *$

Example [Parenthesized expression]: Parentheses make the translation process more difficult because the equivalent postfix expression will be parenthesis-free.

The expression $a * (b + c) * d$ which results $abc + * d *$ in postfix. Figure shows the translation process.

Token	Stack			Top	Output
	[0]	[1]	[2]		
a				-1	a
*	*			0	a
(	*	(		1	a
b	*	(		1	ab
+	*	(	+	2	ab
c	*	(	+	2	abc
)	*			0	abc+
*	*			0	abc +*
d	*			0	abc +*d
eos	*			0	abc +*d*

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

- The analysis of the examples suggests a precedence-based scheme for stacking and unstacking operators.
- The left parenthesis complicates matters because it behaves like a low-precedence operator when it is on the stack and a high-precedence one when it is not. It is placed in the stack whenever it is found in the expression, but it is unstacked only when its matching right parenthesis is found.
- There are two types of precedence, **in-stack precedence (isp)** and **incoming precedence (icp)**.

The declarations that establish the precedence's are:

```
/* isp and icp arrays-index is value of precedence lparen rparen, plus, minus,
times, divide, mod, eos */ int isp[] = {0,19,12,12,13,13,13,0}; int icp[] =
{20,19,12,12,13,13,13,0};
```

```
void postfix(void)
```

```
{
```

```
    char symbol;
```

```
    precedence token;
```

```
    int n = 0, top = 0;
```

```
    /* Place end-of-stack symbol on stack */
```

```
    stack[0] = eos;
```

```
    for (token = getToken(&symbol, &n);
```

```
        token != eos;
```

```
        token = getToken(&symbol, &n))
```

```
    {
```

```
        if (token == operand)
```

```
        {
```

```
            printf("%c", symbol);
```

```
        }
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
else if (token == rparen)
{
    while (stack[top] != lparen)
        printToken(pop());

    pop(); /* Remove left parenthesis */
}
else
{
    while (isp[stack[top]] >= icp[token])
        printToken(pop());

    push(token);
}
}

while ((token = pop()) != eos)
    printToken(token);

printf("\n");
}
```

Program: Function to convert from infix to postfix

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Analysis of postfix: Let n be the number of tokens in the expression. $\Theta(n)$ time is spent extracting tokens and outputting them. Time is spent in the **two while loops**, is $\Theta(n)$ as the number of tokens that get stacked and unstacked is linear in n . So, the complexity of function postfix is $\Theta(n)$.

EVALUATION OF POSTFIX EXPRESSION

The evaluation process of postfix expression is simpler than the evaluation of infix expressions because there are no parentheses to consider.

To evaluate an expression, make a single **left-to-right** scan of it. Place the operands on a stack until an operator is found. Then remove from the stack, the correct number of operands for the operator, perform the operation, and place the result back on the stack and continue fashion until the end of the expression. We then remove the answer from the top of the stack.

```
int eval(void)
{
    precedence token;
    char symbol;
    int op1, op2, n = 0;

    top = -1;

    token = getToken(&symbol, &n);

    while (token != eos)
    {
        if (token == operand)
        {
            push(symbol - '0');
        }
    }
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
else
{
    op2 = pop();
    op1 = pop();

    switch (token)
    {
        case plus:
            push(op1 + op2);
            break;

        case minus:
            push(op1 - op2);
            break;

        case times:
            push(op1 * op2);
            break;

        case divide:
            push(op1 / op2);
            break;

        case mod:
            push(op1 % op2);
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
    }

    }

    token = getToken(&symbol, &n);
}

return pop();
}

getToken() Function
precedence getToken(char *symbol, int *n)
{
    *symbol = expr[(*n)++];

    switch (*symbol)
    {
        case '(':
            return lparen;

        case ')':
            return rparen;

        case '+':
            return plus;
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
case '-':  
    return minus;  
  
case '/':  
    return divide;  
  
case '*':  
    return times;  
  
case '%':  
    return mod;  
  
case '':  
    return eos;  
  
default:  
    return operand;  
}  
}
```

Program: Function to get a token from the input string

The function **eval ()** contains the code to evaluate a postfix expression. Since an operand (symbol) is initially a character, convert it into a single digit integer.

To convert use the statement, **symbol-'0'**. The statement takes the ASCII value of **symbol** and subtracts the ASCII value of '0', which is 48, from it. For example, suppose **symbol = '1'**. The character '1' has an ASCII value of 49. Therefore, the statement **symbol-'0'** produces as result the number 1.

The function **getToken()**, obtain tokens from the expression string. If the token is an operand, convert it to a number and add it to the stack. Otherwise remove two operands

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

from the stack, perform the specified operation, and place the result back on the stack.
When the end of expression is reached, remove the result from the stack.

RECURSION

A recursive procedure

Suppose P is a procedure containing either a Call statement to itself or a Call statement to a second procedure that may eventually result in a Call statement back to the original procedure P. Then P is called a recursive procedure. So that the program will not continue to run indefinitely, a recursive procedure must have the following two properties:

1. There must be certain criteria, called base criteria, for which the procedure does not call itself.
2. Each time the procedure does call itself (directly or indirectly), it must be closer to the base criteria.

Recursive procedure with these two properties is said to be well-defined.

A recursive function

A function is said to be recursively defined if the function definition refers to itself. A recursive function must have the following two properties:

1. There must be certain arguments, called **base values**, for which the function does not refer to itself.
2. Each time the function does refer to itself, the argument of the function must be closer to a **base value**

A recursive function with these two properties is also said to be well-defined.

Factorial Function

“The product of the positive integers from 1 to n, is called "n factorial" and is denoted by n!” $n! = 1 * 2 * 3 \dots (n - 2) * (n - 1) * n$

It is also convenient to define $0! = 1$, so that the function is defined for all nonnegative integers.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Definition: (Factorial Function)

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

Observe that this definition of $n!$ is recursive, since it refers to itself when it uses $(n - 1)!$

- (a) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value)
- (b) The value of $n!$ for arbitrary n is defined in terms of a smaller value of n which is closer to the base value 0.

The following are two procedures that each calculate n factorial .

- 1. **Using for loop:** This procedure evaluates $N!$ using an iterative loop process

Procedure: FACTORIAL (FACT, N)

This procedure calculates $N!$ and returns the value in the variable FACT.

- 1. If $N = 0$, then: Set FACT: = 1, and Return.
- 2. Set FACT: = 1. [Initializes FACT for loop.]
- 3. Repeat for $K = 1$ to N . Set FACT: = $K \cdot \text{FACT}$.
[End of loop.]
- 4. Return.

- 2. **Using recursive function:** This is a recursive procedure, since it contains a call to itself

Procedure: FACTORIAL (FACT, N)

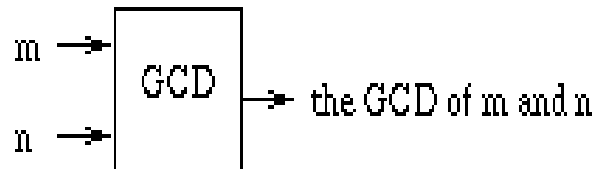
This procedure calculates $N!$ and returns the value in the variable FACT.

- 1. If $N = 0$, then: Set FACT: = 1, and Return.
- 2. Call FACTORIAL (FACT, $N - 1$).
- 3. Set FACT: = $N \cdot \text{FACT}$.
- 4. Return.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

GCD

The **greatest common divisor** (GCD) of two integers m and n is the greatest integer that divides both m and n with no remainder.



For $m \geq n \geq 0$, $\text{gcd}(m, n) = \begin{cases} n & \text{If } n \text{ divides } m \text{ with no minimum} \\ \text{gcd}(n, \text{remainder of } \frac{m}{n}) & \text{Otherwise} \end{cases}$

Procedure: GCD (M, N)

1. If $(M \% N) = 0$, then set $\text{GCD} = N$ and RETURN
2. Call GCD (N, $M \% N$)
3. Return

Fibonacci Sequence

The Fibonacci sequence (usually denoted by $F_0, F_1, F_2, \dots$) is as follows: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55

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

Definition:

(Fibonacci

Sequence) a) If $n =$

0 or $n = 1$, then $F_n =$

n

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

Here

- (a) The base values are 0 and 1

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

- (b) The value of F_n is defined in terms of smaller values of n which are closer to the base values.

A procedure for finding the n^{th} term F_n of the Fibonacci sequence follows

Procedure: FIBONACCI (FIB, N)

This procedure calculates F_N and returns the value in the first parameter FIB.

1. If $N = 0$ or $N = 1$, then: Set FIB: = N , and Return.
2. Call FIBONACCI (FIBA, $N - 2$).
3. Call FIBONACCI (FIBB, $N - 1$).
4. Set FIB: = FIBA + FIBB.
5. Return.

Tower of Hanoi

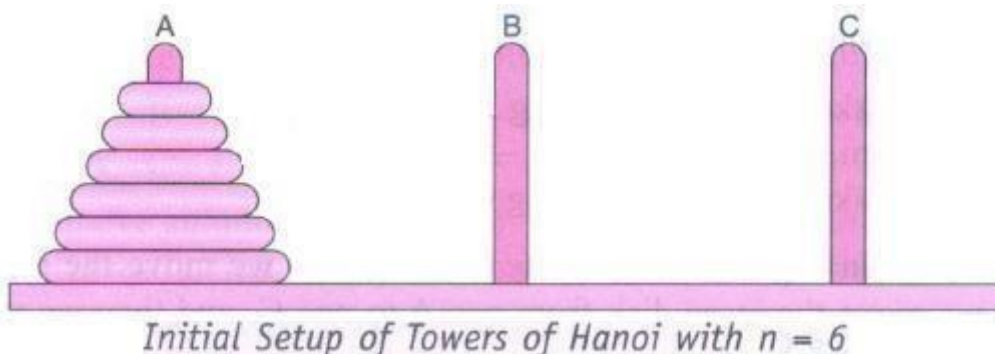
Problem description

Suppose three pegs, labeled A, B and C, are given, and suppose on peg A a finite number n of disks with decreasing size are placed.

The objective of the game is to move the disks from peg A to peg C using peg B as an auxiliary.

The rules of the game are as follows:

1. Only one disk may be moved at a time. Only the top disk on any peg may be moved to any other peg.
2. At no time can a larger disk be placed on a smaller disk.

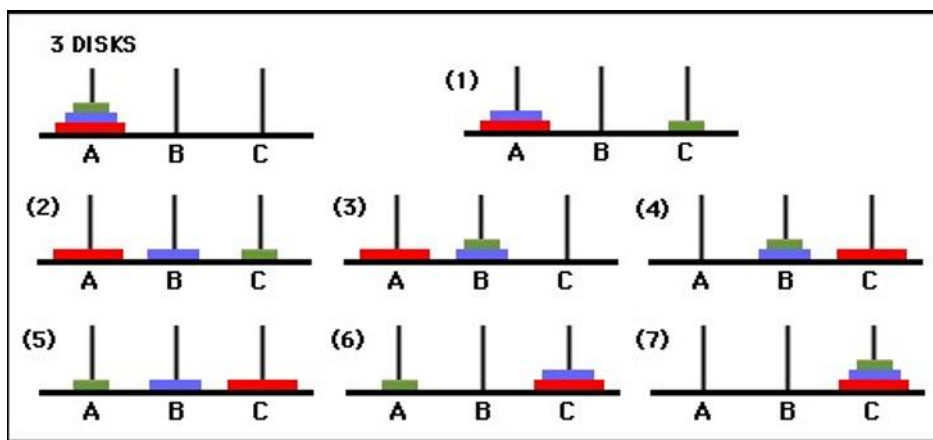


DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

We write $A \rightarrow B$ to denote the instruction "Move top disk from peg A to peg B"

Example: Towers of Hanoi problem for $n = 3$.

Solution: Observe that it consists of the following seven moves



1. Move top disk from peg A to peg C.
2. Move top disk from peg A to peg B.
3. Move top disk from peg C to peg B.
4. Move top disk from peg A to peg C.
5. Move top disk from peg B to peg A.
6. Move top disk from peg B to peg C.
7. Move top disk from peg A to peg C.

In other words,

$n=3$: $A \rightarrow C, A \rightarrow B, C \rightarrow B, A \rightarrow C, B \rightarrow A, B \rightarrow C, A \rightarrow C$

For completeness, the solution to the Towers of Hanoi problem for

$n = 1$ and $n = 2$ $n=1$: $A \rightarrow C$ $n=2$: $A \rightarrow B, A \rightarrow C, B \rightarrow C$

The Towers of Hanoi problem for $n > 1$ disks may be reduced to the following sub-problems:

- (1) Move the top $n - 1$ disks from peg A to peg B
- (2) Move the top disk from peg A to peg C: $A \rightarrow C$.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

(3) Move the top $n - 1$ disks from peg B to peg C.

The general notation

- TOWER (N , BEG, AUX, END) to denote a procedure which moves the top n disks from the initial peg BEG to the final peg END using the peg AUX as an auxiliary.
- When $n = 1$, the solution:
TOWER (1 , BEG, AUX, END) consists of the single instruction
BEG $\rightarrow$ END
- When $n > 1$, the solution may be reduced to the solution of the following three sub- problems:
 - (a) TOWER ($N - 1$, BEG, END, AUX)
 - (b) TOWER (1 , BEG, AUX, END) or BEG $\rightarrow$ END
 - (c) TOWER ($N - 1$, AUX, BEG, END)

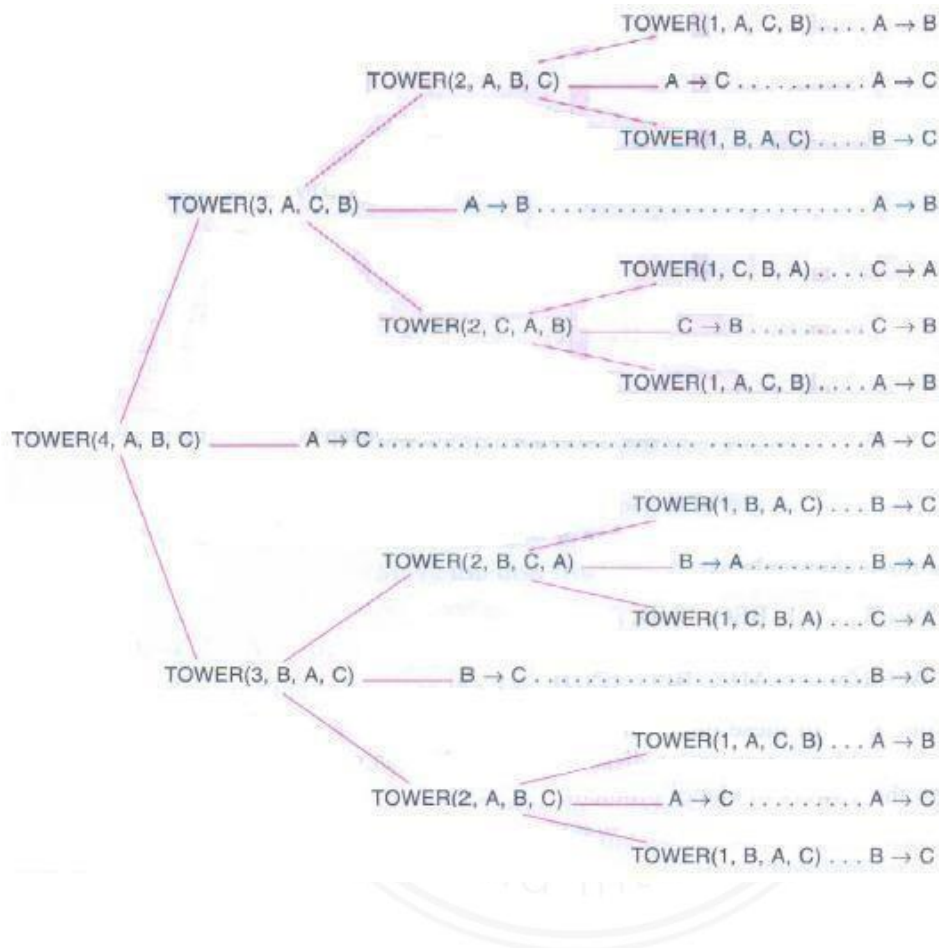
Procedure: TOWER (N , BEG, AUX, END)

This procedure gives a recursive solution to the Towers of Hanoi problem for N disks.

1. If $N=1$, then:
 - (a) Write: BEG $\rightarrow$ END.
 - (b) Return.[End of If structure.]
2. [Move $N - 1$ disks from peg BEG to peg AUX.] Call TOWER ($N - 1$, BEG, END, AUX).
3. Write: BEG $\rightarrow$ END.
4. [Move $N - 1$ disks from peg AUX to peg END.] Call TOWER ($N - 1$, AUX, BEG, END).
5. Return.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

Example: Towers of Hanoi problem for $n = 4$



Ackermann function

The Ackermann function is a function with two arguments each of which can be assigned any nonnegative integer: 0, 1, 2, ...

Definition: (Ackermann

Function) (a) If

$m = 0$, then $A(m,$

$n) = n + 1$.

(b) If $m \neq 0$ but $n = 0$, then $A(m, n) = A(m - 1, 1)$

(c) If $m \neq 0$ and $n \neq 0$, then $A(m, n) = A(m - 1, A(m, n - 1))$

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

QUEUES

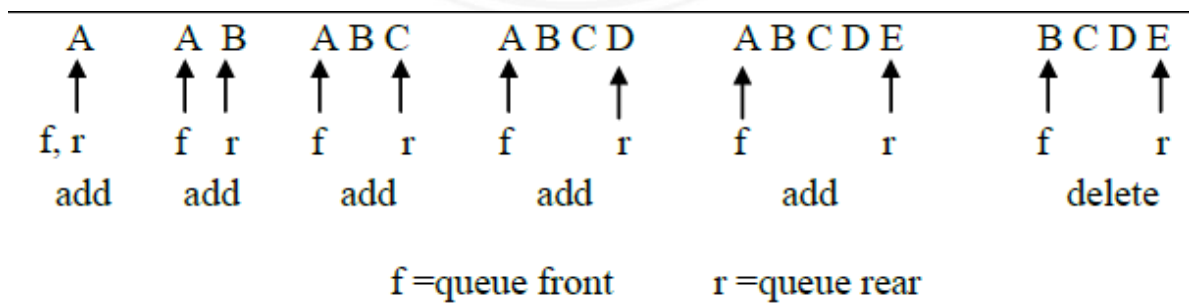
DEFINITION

- “A queue is an **ordered list** in which insertions (additions, pushes) and deletions (removals and pops) take place at different ends.”
- The end at which new elements are added is called the **rear**, and that from which old elements are deleted is called the **front**.

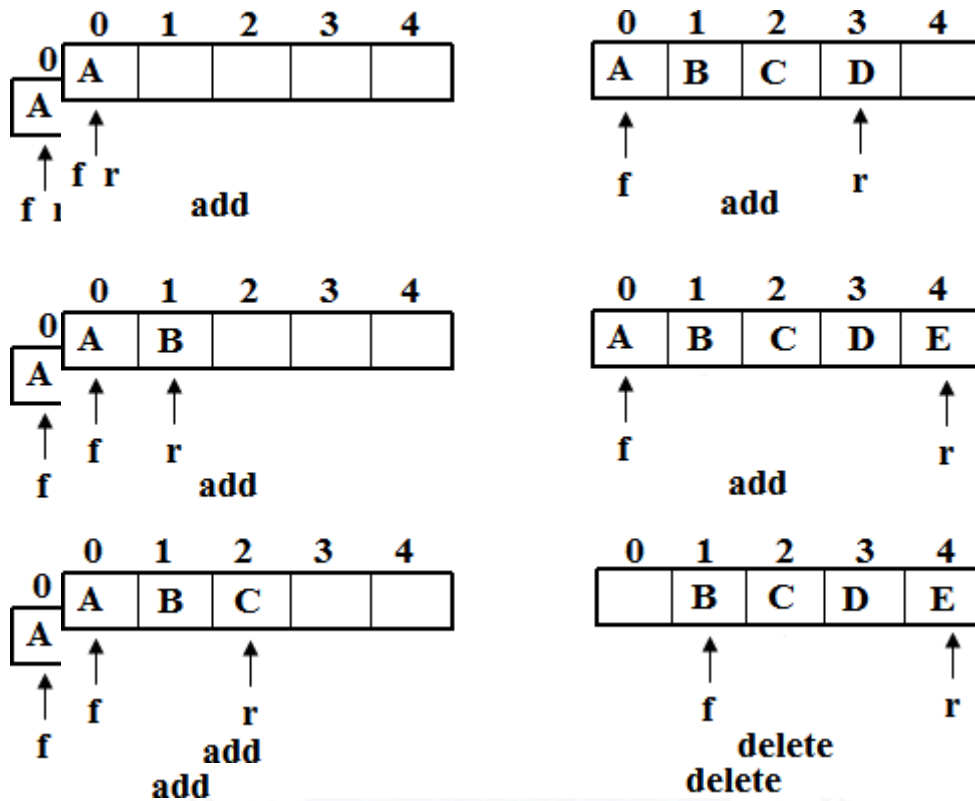
If the elements are inserted A, B, C, D and E in this order, then A is the first element deleted from the queue. Since the first element inserted into a queue is the first element removed, queues are also known as **First-In-First-Out (FIFO) lists**

Figure indicates the way elements will be deleted from the queue and the way new elements will be added to the queue.

- Whenever an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment $FRONT := FRONT + 1$
- When an element is added to the queue, the value of REAR is increased by 1; this can be implemented by the assignment $REAR := REAR + 1$



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML



QUEUE OPERATIONS

Implementation of the queue operations as follows.

1.Queue Create

```
#define MAX_QUEUE_SIZE 100 /* Maximum queue size */
```

```
typedef struct
```

```
{
    int key;          /* Other fields can be added */
} element;
```

```
element queue[MAX_QUEUE_SIZE];
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
int rear = -1;
```

```
int front = -1;
```

```
2.Boolean IsEmptyQ(queue) ::= front==rear
```

```
3.Boolean IsFullQ(queue) ::= rear == MAX_QUEUE_SIZE-1
```

In the queue, two variables are used which are **front** and **rear**. The queue increments **rear** in addq() and **front** in delete(). The function calls would be **addq (item);** and **item =delete();**

```
4.addq(item)
```

```
void addq(element item)
```

```
{
```

```
/* Add an item to the queue */
```

```
if (rear == MAX_QUEUE_SIZE - 1)
```

```
    queueFull();
```

```
    queue[++rear] = item;
```

```
}
```

```
5.deletq( )
```

```
element deletq()
```

```
{
```

```
/* Remove element at the front of the queue */
```

```
if (front == rear)
```

```
    return queueEmpty(); /* Return an error key */
```

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

```
return queue[++front];  
}
```

7.queueFull()

The **queueFull** function which prints an error message and terminates execution

```
void queueFull()  
{  
    fprintf(stderr, "Queue is full, cannot add element");  
    exit(EXIT_FAILURE);  
}
```

Example: Job scheduling

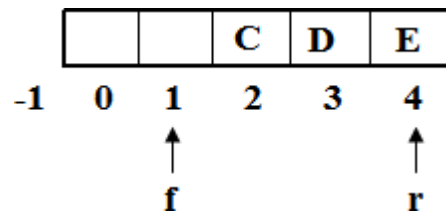
- Queues are frequently used in creation of a **job queue** by an operating system. If the operating system does not use priorities, then the jobs are processed in the order they enter the system.
- Figure illustrates how an operating system process jobs using a sequential representation for its queue.

front	rear	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					Queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Drawback of Queue

When item enters and deleted from the queue, the queue gradually shifts to the right as shown in figure.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML



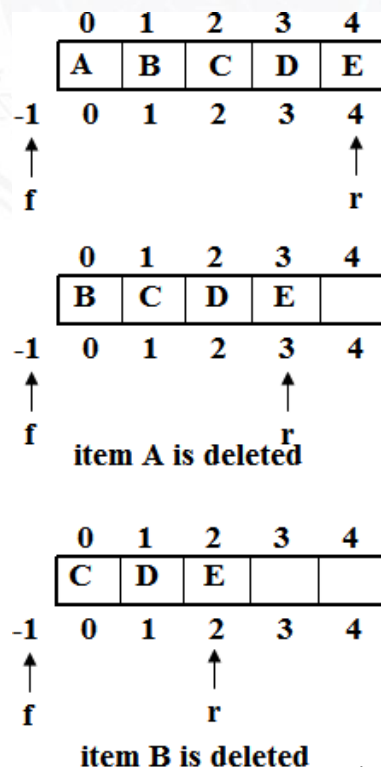
In this above situation, when we try to insert another item, which shows that the **queue is full**. This means that the **rear** index equals to $\text{MAX_QUEUE_SIZE} - 1$. But even if the space is available at the front end, rear insertion cannot be done.

Overcome of Drawback using different methods

Method 1:

- When an item is deleted from the queue, move the entire queue to the **left** so that the first element is again at `queue[0]` and front is at `-1`. It should also recalculate **rear** so that it is correctly positioned.

Shifting an array is very time-consuming when there are many elements in queue & `queueFull` has worst case complexity of $O(\text{MAX_QUEUE_SIZE})$.





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML





ATME

College of Engineering



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING - AI & ML

