SYLLABUS OF SEMESTER-I, MCA (Artificial Intelligence and Machine Learning)

Course Code: 24CS60TH1177 Course: Data Structures

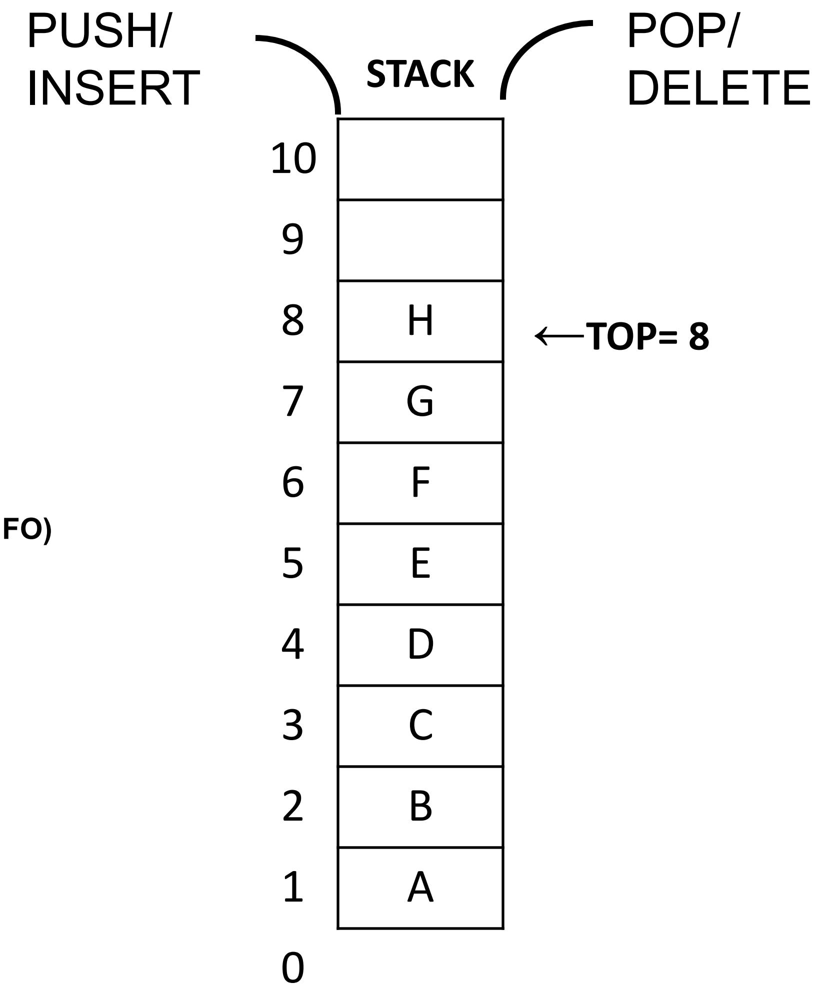
UNIT-III

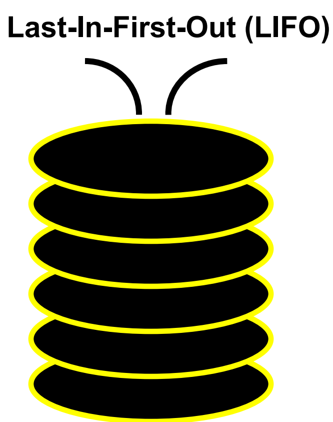
Stacks: Definition and example, primitive operations on Stacks, Arithmetic expressions (Infix, Postfix and Prefix), Evaluating postfix expression, converting an expression from infix to postfix. Applications of stacks: Tower of Hanoi Problem, Recursion, etc.

STACKS

- A ***stack*** is a list of elements in which an element may be inserted or deleted only at one end, called the ***top*** of the stack.
- Inserting an element into a stack, called ***push***.
- Deleting an element into a stack, called ***pop***.

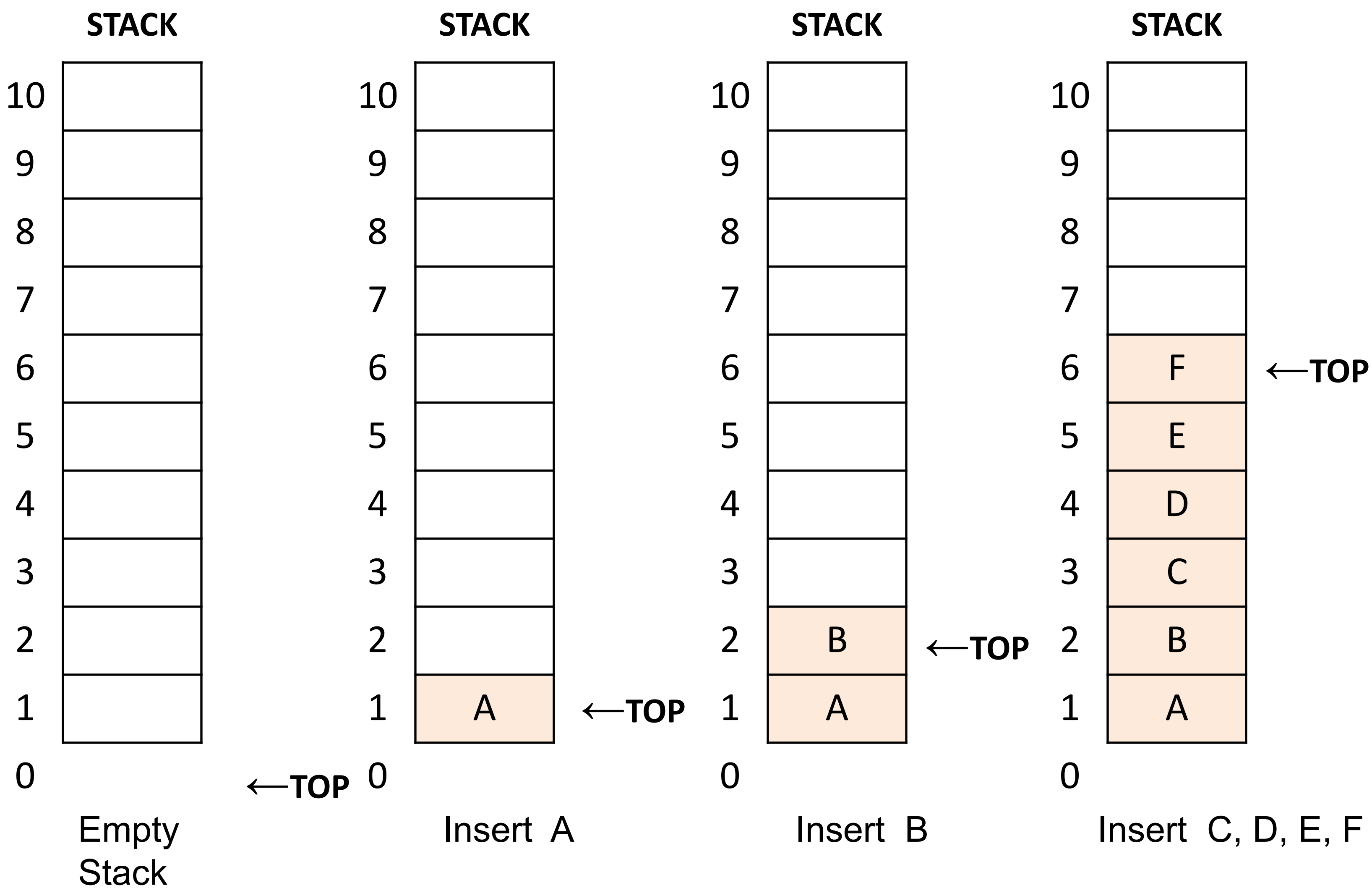
Last-In-First-Out (LIFO)



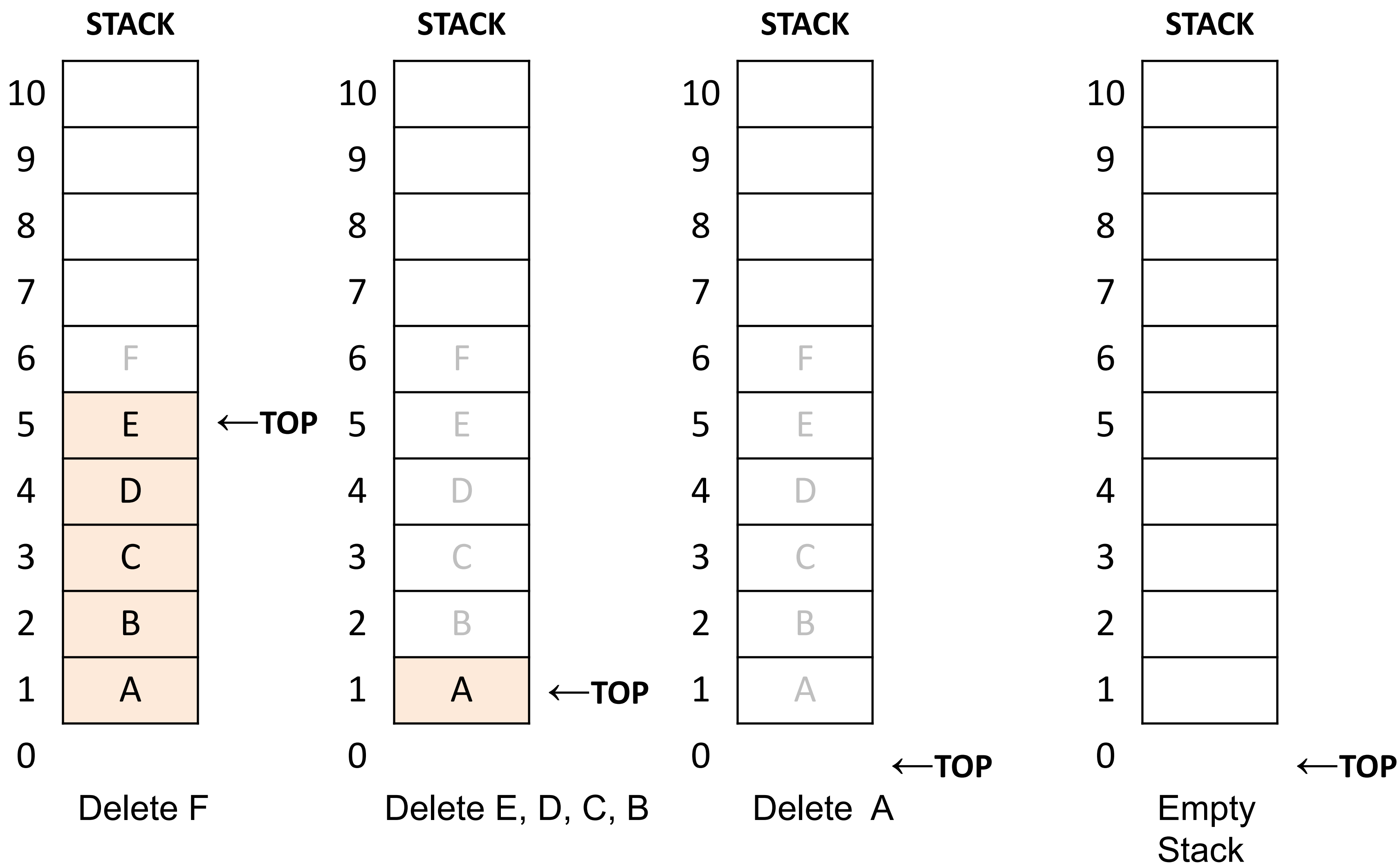


Stack of Dishes

Inserting Elements onto STACK

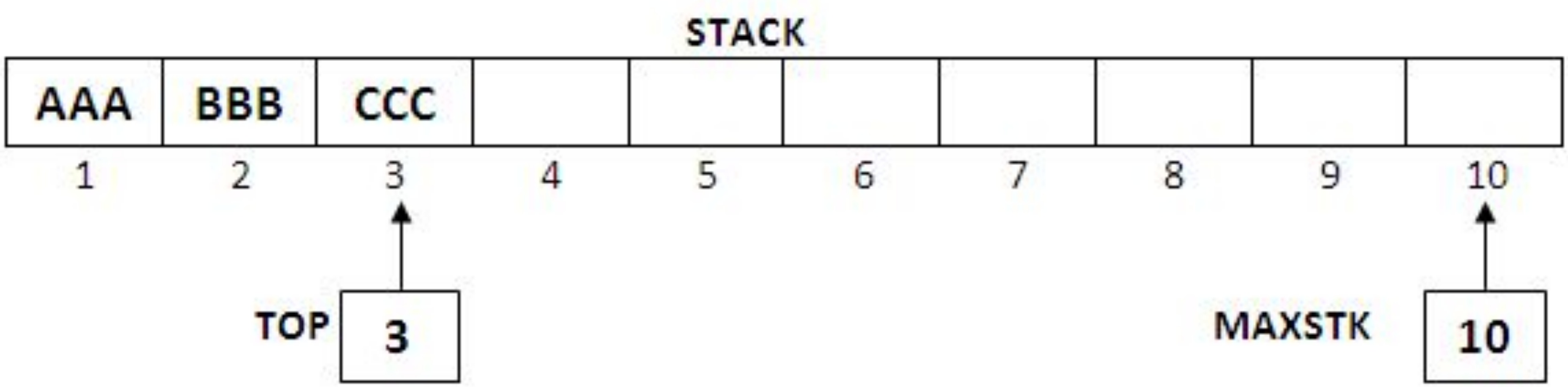


Deleting Elements from STACK



Array Representation of Stacks

- Stack will be maintained by a linear array **STACK** in memory.
- A pointer variable **TOP** points to the top element.
- **MAXSTK** gives the maximum number of elements.



Procedure: PUSH(STACK, TOP, MAXSTK, ITEM)

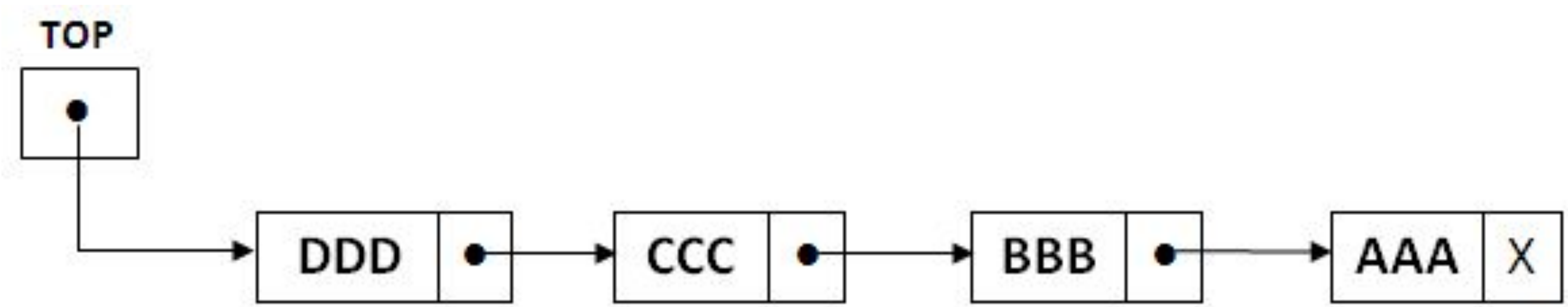
1. If TOP=MAXSTK, then: Print “Overflow” and return.
2. Set TOP := TOP + 1
3. Set STACK[TOP] := ITEM
4. Return.

Procedure: POP(STACK, TOP, ITEM)

1. If TOP = 0, then: Print “Underflow” and return.
2. Set ITEM := STACK[TOP]
3. Set TOP := TOP - 1
4. Return.

Linked Representation of Stacks

- Stack elements can be represented in memory using a linked list.
- Data elements are called nodes.
- Node divided in two parts: first part; ***info*** and second; ***link*** field.
- **TOP** is a pointer to starting node.




```
struct Node
{
   int info;
   struct Node *link;
};
```

Algorithm: PUSH_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)

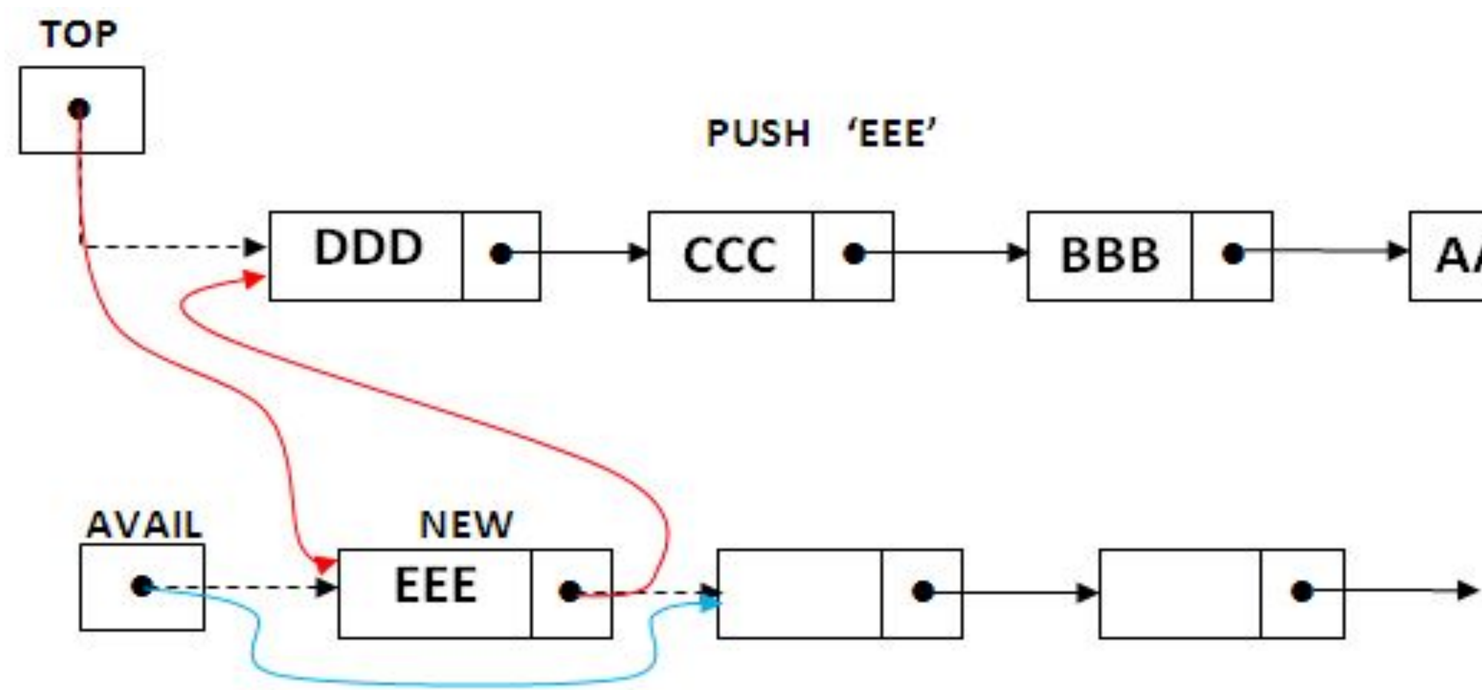
1.If AVAIL = NULL, then: Print “Overflow” and Exit.

2.Set NEW := AVAIL, AVAIL := LINK[AVAIL].

3.Set INFO[NEW] := ITEM

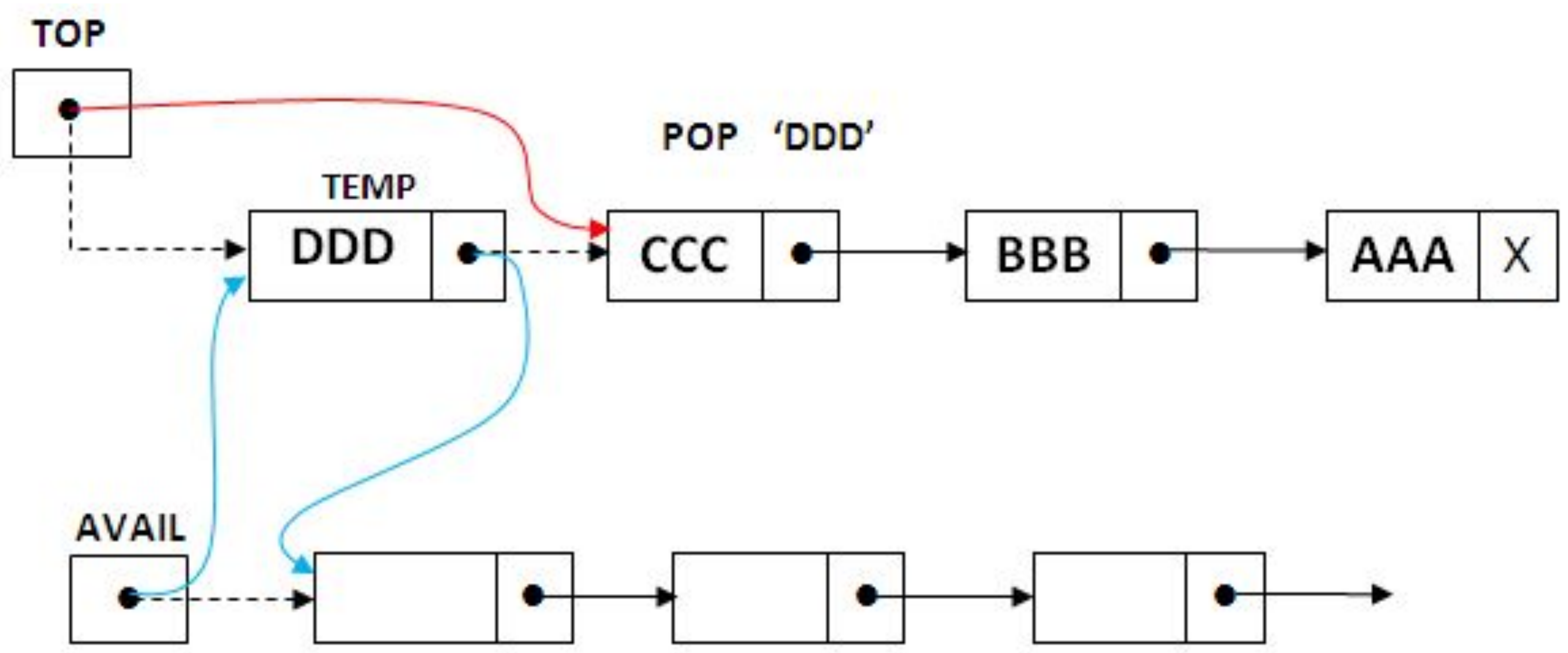
4.Set LINK[NEW] := TOP and TOP := NEW

5.Exit.



Algorithm: POP_LINKSTACK(INFO, LINK, TOP, AVAIL, ITEM)

1. If TOP = NULL, then: Print “Underflow” and Exit.
2. Set ITEM := INFO[TOP]
3. Set TEMP := TOP, TOP := LINK[TOP].
4. Set LINK[TEMP] := AVAIL and AVAIL:= TEMP
5. Exit.



Arithmetic Expressions; Polish Notation

- Precedence of five binary operations:

  Highest : Exponentiation (↑ or **^**)

  Next highest : Multiplication (*****) and Division (**/**)

  Lowest : Addition (**+**) and Subtraction (**-**)

  evaluate: 2 ↑ 3 + 5 *2 ↑ 2 – 12 / 6

- The operator symbol is placed between its two operands, called ***infix notation***.

  e.g. A + B C - D E * F G/H J ↑ K

- The notation in which the operator symbol is placed before its two operands, called ***polish notation***. (named after Polish mathematician **Jan Lukasiewicz**)

- These notations also called **prefix notation**.

  e.g. +A B -CD *EF /GH ↑ JK

- The notation in which the operator symbol is placed after its two operands, called ***reverse polish notation***.

- These notations also called **postfix notation**.

  e.g. A B + CD- EF* GH/ JK ↑

Convert following infix expression to prefix and postfix expressions.

1) $(A + B) * C$
2) $(A \uparrow B * C)/ D * (E - F)$
3) $(A * B \wedge C) * (D/E) + (F - G)$
4) $2 + (3 * 5) / (7 - 4)$
5) $6 \wedge 3 + (8 * 2) + 5$

Evaluation of Postfix Expressions

❖ Evaluation of postfix expression (P) uses the STACK to hold operands.

1. Add a right parenthesis "**)**" at the end of **P**. (This acts as a sentinel)
2. Scan **P** from left to right and repeat Steps 3 and 4 for each element in **P** until the sentinel "**)**" is encountered.
3. If an operand is encountered, put it on **STACK**.
4. If an operator ⊗ is encountered, then:

   (a) Remove the two top elements of **STACK**, where **A** is the top element and **B** is the next-to-top element.

   (b) Evaluate **B** ⊗ **A**.

   (c) Place the result of (b) back on **STACK**.

5. Set **VALUE** equals to the top element on **STACK**.
6. Exit.

Expression : 2 * (3 + 5) – 2

Postfix expression: 2, 3, 5, +, *, 2, -

Add sentinel ")" to right

	Symbol scanned	**STACK**
1	**2**	**2**
2	**3**	**2, 3**
3	**5**	**2, 3, 5**
4	**+**	**2, 8**
5	*****	**16**
6	**2**	**16, 2**
7	**-**	**14**
8	**)**	**VALUE =14**

Transforming Infix Expressions into Postfix Expressions

- Transforming an arithmetic expression **Q** into postfix expression **P** uses the **STACK** to hold operators.

1. Push “**(**“ onto STACK, and add “**)**” to the end of **Q**.
2. Scan **Q** from left to right and repeat Steps 3 to 6 for each element of **Q** until the STACK is empty.
3. If an operand is encountered, add it to **P**.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator ⊗ is encountered, then:
   (a) Repeatedly pop from **STACK** and add to P each operator which has the same precedence as or higher precedence then ⊗.
   (b) Add ⊗ to STACK.
6. If the right parenthesis is encountered, then:
   (a) Repeatedly pop from **STACK** and add to P each operator until a left parenthesis is encountered.
   (b) Remove the left parenthesis.
7. Exit

Expression Q : (A * B ↑ C) * (D/E + F - G)

Add ")" to expression and "(" to STACK

Symbol scanned		STACK	Expression P
1	**(**	**((**	
2	**A**	**((**	**A**
3	*****	**((***	**A**
4	**B**	**((***	**A B**
5	↑	**((* ↑**	**A B**
6	**C**	**((* ↑**	**A B C**
7	**)**	**(**	**A B C ↑ ***
8	*****	**(***	**A B C ↑ ***
9	**(**	**(* (**	**A B C ↑ ***
10	**D**	**(* (**	**A B C ↑ * D**
11	**/**	**(* (/**	**A B C ↑ * D**
12	**E**	**(* (/**	**A B C ↑ * D E**
13	**+**	**(* (+**	**A B C ↑ * D E /**
14	**F**	**(* (+**	**A B C ↑ * D E / F**
15	**-**	**(* (-**	**A B C ↑ * D E / F +**
16	**G**	**(* (-**	**A B C ↑ * D E / F + G**
17	**)**	**(***	**A B C ↑ * D E / F + G -**
18	**)**		**A B C ↑ * D E / F + G - ***

Recursion

- 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***.
- 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.

Factorial Function:

Definition:

(a) If n = 0, then n! = 1

(b) If n > 0, then n! = n * (n-1)!

Procedure: **FACTORIAL(FACT, N)**

1. If N = 0, then: Set FACT := 1 and Return.
2. Call FACTORIAL(FACT, N-1)
3. Set FACT := N * FACT.
4. Return.

Recursion …

Fibonacci Sequence:

Definition:

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

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

Procedure: **FIBONACCI(FIB, N)**

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.

Divide-and-Conquer Algorithms

- Suppose **A** is an algorithm which partitions **S** into smaller sets such that the solution of the problem **P** for **S** is reduced to the solution of **P** for one or more of the smaller sets. Then **A** is called a ***divide-and-conquer*** algorithm.
- **A divide-and-conquer algorithm may be viewed as a recursive procedure.**

Ackermann Function:

Definition:

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

Divide-and-Conquer Algorithms . . .

Towers of Hanoi

❖ Suppose three pegs, A, B, and C are given. The object of the game is to move the disks from peg A to peg C using peg B as an auxiliary.

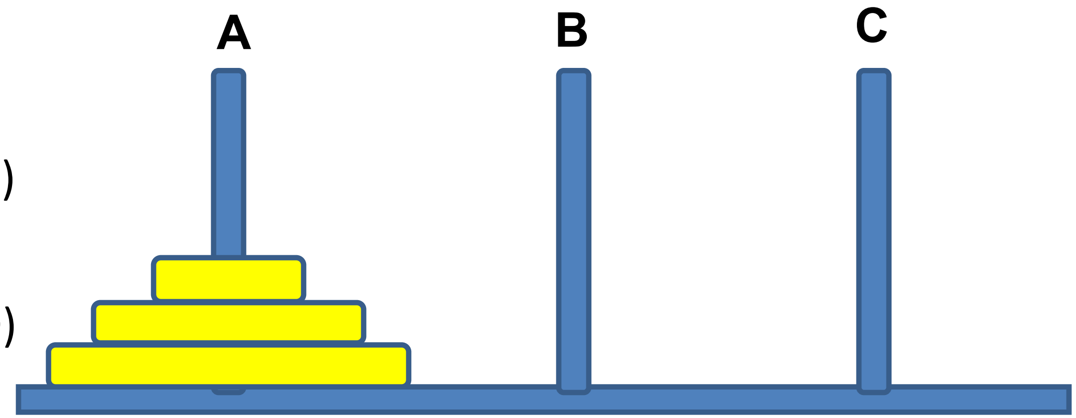
❖ The rules of the game are as follows:

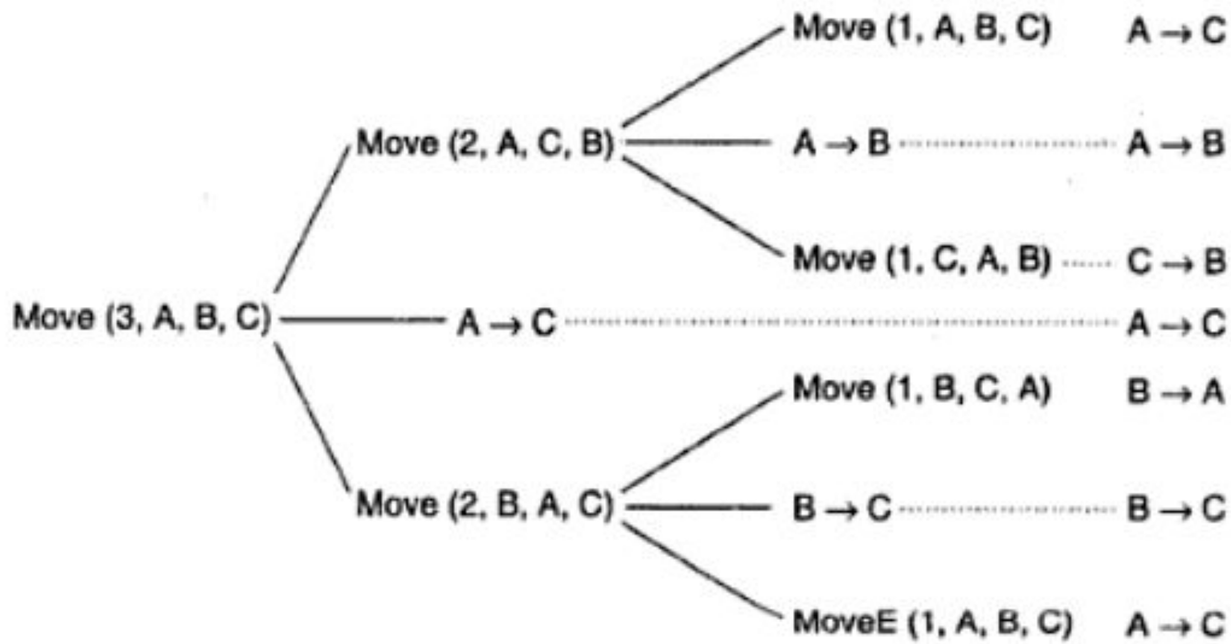
(a) Only one disk (top disk) may be moved at a time.

(b) Only smaller disk place on a larger disk.

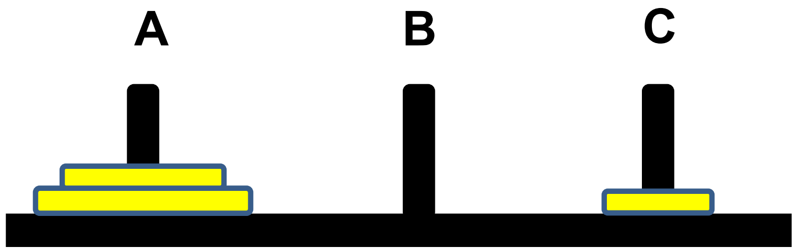
Procedure: **TOWER(N, BEG, AUX, END)**

1. If N = 1, then:
   a. Write: BEG → END
   b. Return.
2. Call TOWER(N – 1, BEG, END, AUX)
3. Write: BEG → END
4. Call TOWER(N – 1, AUX, BEG, END)
5. Return

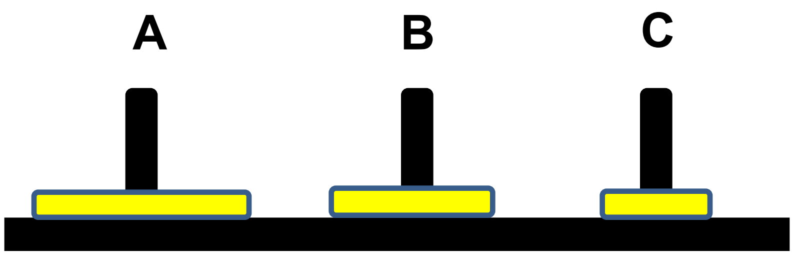




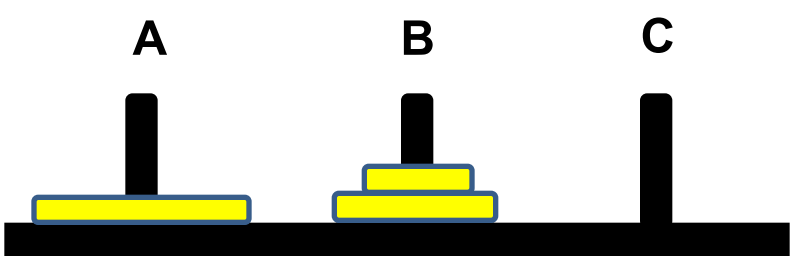
Tower of Hanoi (with N = 3) solution with recursion.



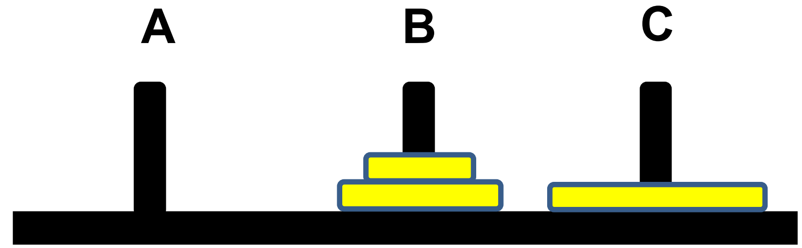
1. Move top disk from **peg A → peg C**



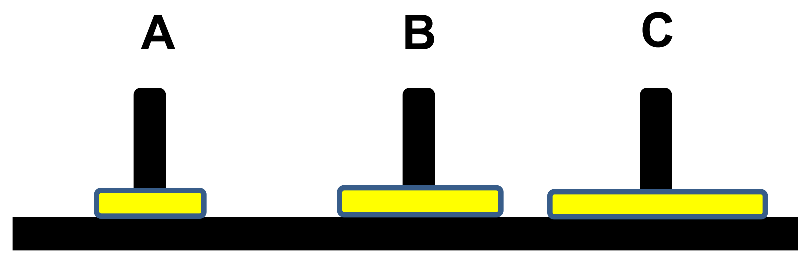
2. Move top disk from **peg A → peg B**



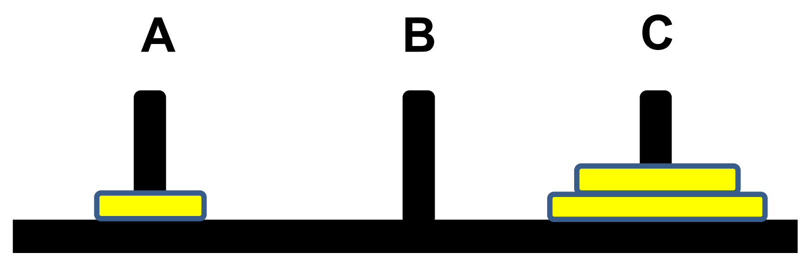
3. Move top disk from **peg C → peg B**



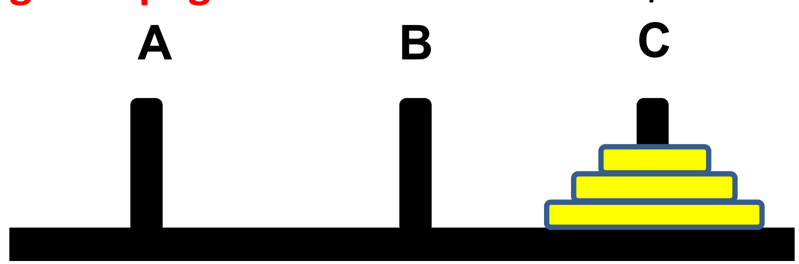
4. Move top disk from **peg A → peg C**



5. Move top disk from **peg B → peg A**



6. Move top disk from **peg B → peg C**



7. Move top disk from **peg A → peg C**

Total No. of moves : $2^n - 1$