# *CSE-281: Data Structures and Algorithms*

# Stacks, Recursion and Queue
# (Chapter-6)

*Ref: Schaum's Outline Series, Theory and problems of Data Structures*
*By Seymour Lipschutz*

*And Online Resource*

***Eftekhar Hossain***
***Assistant Professor***
***Dept. of ETE, CUET***

# Topics to be Covered

- Stacks
- Stack Implementation
- Stack Applications
- Recursion
- Recursion Applications
- Tower of Hanoi
- Queue
- Priority Queue

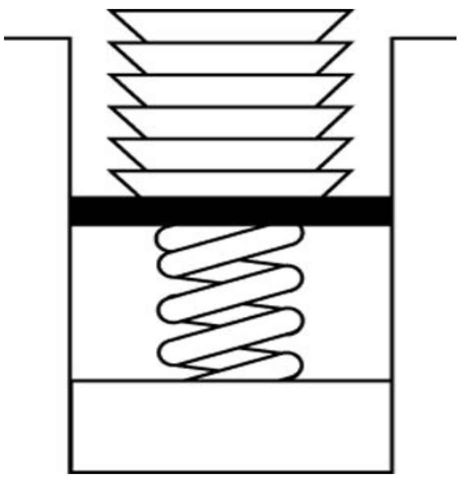
# What is Stack?

- A Stack: Stores a set of elements in a particular order

- LAST IN, FIRST OUT (LIFO) property
  - The last item placed on the stack will be the first item removed

  - Analogy
    - A stack of coins
    - A stack of dishes in a cafeteria

(a)



(b)

***Figure 1:*** Stack of (a) Coins, (b) Cafeteria Dishes

# What is Stack?

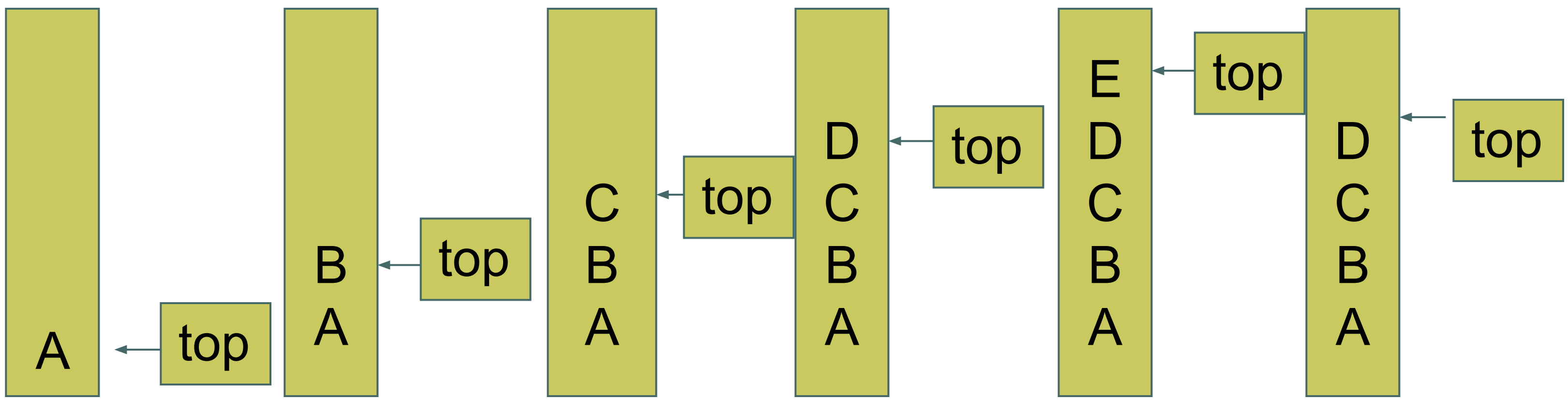



**Figure 2:** Diagrams of Stack

# Stack Operations

- **Stack Operations**
  - Create an empty stack
  - Destroy a stack
  - Determine whether a stack is empty
  - Add a new item
  - Remove the item that was added most recently
  - Retrieve the item that was added most recently

# Stack Applications

- **Real life**
  - Pile of books
  - Plate trays
- More applications related to computer science
  - Program execution stack
  - Evaluating expressions

# Array-Based Stack Implementation

- ALGORITHM OF INSERTION IN STACK: (PUSH)

```
Insertion(a[], top, item, max):
 If top=max then
    print 'STACK OVERFLOW'
 exit
 else
    top=top+1
 end if
 a[top]=item
 Exit
```

# Array-Based Stack Implementation

- ALGORITHM OF DELETION IN STACK:(POP)

```
Deletion(a,top,item):

If top=0 then
    print 'STACK UNDERFLOW'
exit
else
    item=a[top]
end if
top = top-1
Exit
```

# Array-Based Stack Implementation

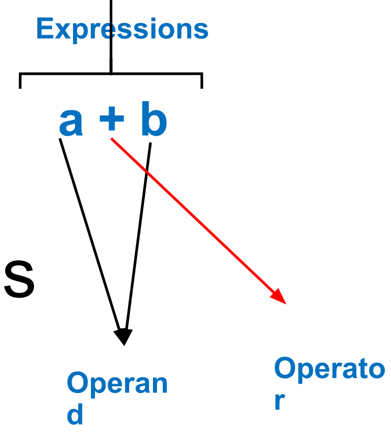
- ALGORITHM OF DISPLAY IN STACK

```
Display(top,i,a[i]):

If top=0 then
    Print 'STACK EMPTY'
exit
else
for i= top to 0
    print a[i]
end for
exit
```

# Algebraic Expressions

- **Infix Expressions**
  - An operator appears between its operands
    - Example: **a + b**




- **Prefix Expressions**
  - An operator appears before its operands
    - Example: **+ a b**
- **Postfix Expressions**
  - An operator appears after its operands
    - Example: **a b +**

# Algebric Expressions

- **Infix Expressions (operand | operator |operand)**
  - **a * b + c**
- **Prefix Expressions (operator |operand| operand)**
  - **+ *abc**
- **Postfix Expressions(operand|operand|operator)**
  - **ab *c+**

# Level of Precedence

| | |
|---|---|
| **Highest** | ↑ |
| **Next Highest** | *, / |
| **Lowes<br>t** | +, - |

# Evaluation of a Postfix Expressions

- Algorithm: To find the VALUE of a postfix expression P using STACK. 5, 6, 2, +, *, 12, 4, /, -

1. Add a right parenthesis ")" at the end of P. (Sentinel)
2. Scan P from left to right and repeat steps 3 and 4 for each element of 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 equal to the top element on STACK.
6. Exit.

# Evaluation of a Postfix Expressions (Example)

- **Postfix Expression P:** 5, 6, 2, +, *, 12, 4, /, -
- **Infix Expression Q:** 5 * ( 6 + 2 ) – 12 / 4

| | Symbol Scanned | STACK |
|---|---|---|
| (1) | 5 | 5 |
| (2) | 6 | 5, 6 |
| (3) | 2 | 5, 6, 2 |
| (4) | + | 5, 8 |
| (5) | * | 40 |
| (6) | 12 | 40, 12 |
| (7) | 4 | 40 |
| (8) | / | 40, 3 , 12 |
| (9) | - | 37 , 4 |
| (10) | ) | |

6 +2 = 8

5 * 8 = 40

12 / 4 = 3

40 - 3 = 37

# Transforming Infix into Postfix Expressions

- Algorithm: POLISH(Q, P)

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 $\otimes$ is encountered, then:
   a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than $\otimes$.
   b) Add $\otimes$ to STACK.

# Continued

- Algorithm: POLISH(Q, P)

6. If a right parenthesis is encountered, then:
   a) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   b) Remove the left parenthesis. [Do not add the left parenthesis to P.]
7. Exit

# Example (Infix to Postfix)

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

| Symbol Scanned | | STACK | Expression P |
|---|---|---|---|
| (1) | A | ( | A |
| (2) | + | ( + | A |
| (3) | ( | ( + ( | A |
| (4) | B | ( + ( | AB |
| (5) | * | ( + ( * | A B |
| (6) | C | ( + ( * | ABC |
| (7) | - | ( + ( - | ABC * |
| (8) | ( | ( + ( - ( | ABC * |
| (9) | D | ( + ( - ( | ABC * D |
| (10) | / | ( + ( - ( / | ABC * D |

# Example (Infix to Postfix)

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

| Symbol Scanned | | STACK | Expression P |
|---|---|---|---|
| (11) | E | ( + ( - ( / | ABC * DE |
| (12) | ↑ | ( + ( - ( / ↑ | ABC * DE |
| (13) | F | ( + ( - ( / ↑ | ABC * DEF |
| (14) | ) | ( + ( - | ABC * DEF ↑ / |
| (15) | * | ( + ( - * | ABC * DEF ↑ / |
| (16) | G | ( + ( - * | ABC * DEF ↑ / G |
| (17) | ) | ( + | ABC * DEF ↑ / G * - |
| (18) | * | ( + * | ABC * DEF ↑ / G * - |
| (19) | H | ( + * | ABC * DEF ↑ / G * - H |
| (20) | ) | | ABC * DEF ↑ / G * - H * + |

# Recursion

- A function that calls itself is known as recursive function and the process of calling function itself is known as recursion.
- Every recursive function must be provided with a way to end the recursion.
- 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 closer to a base value.
- A récursive function with these following two properties is said to be well-defined.

# Factorial Function

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

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

- This function of $n!$ is recursive, since it refers to itself when it uses $(n - 1)!$
  1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value)
  2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

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

- This function of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$
  1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value)
  2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

(1) a) If $n = 0$, then $n! = 1$ b) If $n > 0$, then $n! = n * (n-1)!$ This function of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$ 1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value) 2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

(2) a) If $n = 0$, then $n! = 1$ b) If $n > 0$, then $n! = n * (n-1)!$ This function of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$ 1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value) 2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

(3) a) If $n = 0$, then $n! = 1$ b) If $n > 0$, then $n! = n * (n-1)!$ This function of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$ 1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value) 2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

(4) a) If $n = 0$, then $n! = 1$ b) If $n > 0$, then $n! = n * (n-1)!$ This function of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$ 1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value) 2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

(5) a) If $n = 0$, then $n! = 1$ b) If $n > 0$, then $n! = n * (n-1)!$ This function of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$ 1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value) 2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

(6) a) If $n = 0$, then $n! = 1$ b) If $n > 0$, then $n! = n * (n-1)!$ This function of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$ 1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value) 2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

(7) a) If $n = 0$, then $n! = 1$ b) If $n > 0$, then $n! = n * (n-1)!$ This function of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$ 1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value) 2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

(8) a) If $n = 0$, then $n! = 1$ b) If $n > 0$, then $n! = n * (n-1)!$ This function of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$ 1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value) 2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

(9) a) If $n = 0$, then $n! = 1$ b) If $n > 0$, then $n! = n * (n-1)!$ This function of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$ 1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value) 2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

# Factorial Recursive Definition in C

// Computes the factorial of a nonnegative integer.
// ***Precondition***: n must be greater than or equal to 0.
// ***Postcondition***: Returns the factorial of n; n is unchanged.

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

**** Self**

| |
|---|
| Write the algorithm for calculating factorial of a number using Recursion or Iteration. |

# Fibonacci Sequence

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

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

- This function of $n!$ is recursive, since it refers to itself when it uses $(n - 1)!$
  1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value)
  2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

**** Self**

**Write the algorithm for finding the Fibonacci number using Recursion or Iteration.**

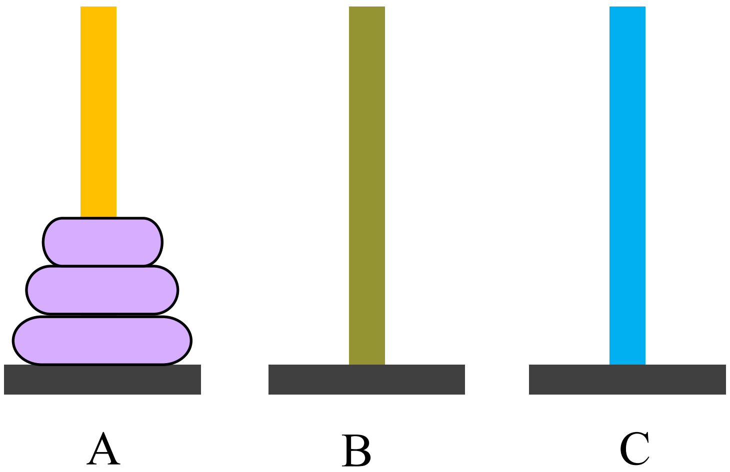
# Basic Recursion Problems

**Try to solve these problems by hand and for verification implement the code.**

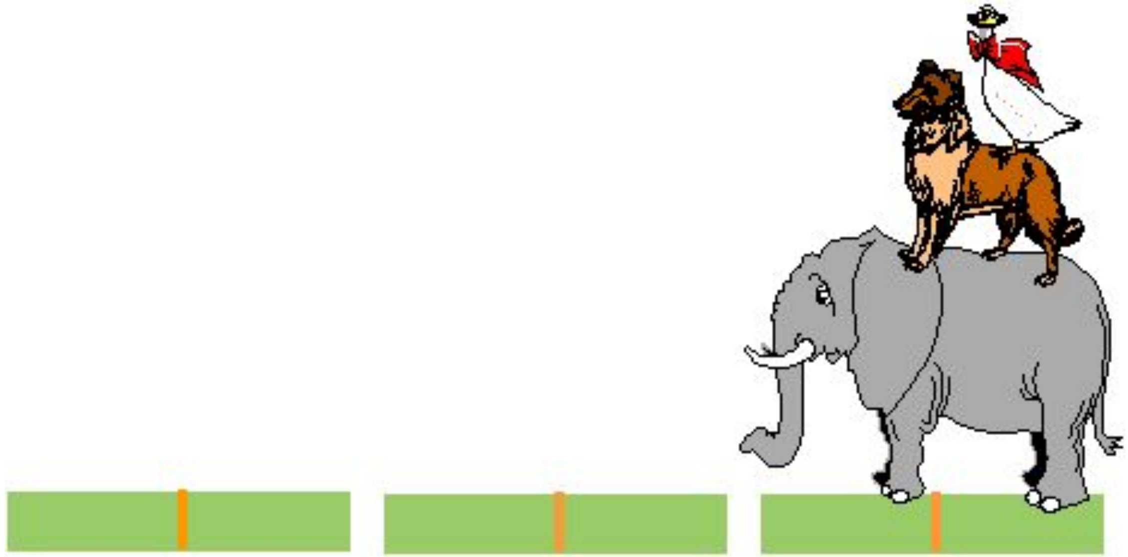
1. **Print Name 5 Times**
2. **Print Linearly from 1 to N.**
3. **Print from N to 1.**
4. **Print Linearly from 1 to N (By Backtrack).**
5. **Print from N to 1 (By Backtrack).**

# The Towers of Hanoi

- Given
  - three poles
  - a set of discs on the first pole, discs of different sizes, the smallest discs at the top
- Goal
  - move all the discs from the left pole to the right one.
- Condition
  - only one disc may be moved at a time.
  - A disc can be placed either on an empty pole or on top of a larger disc.

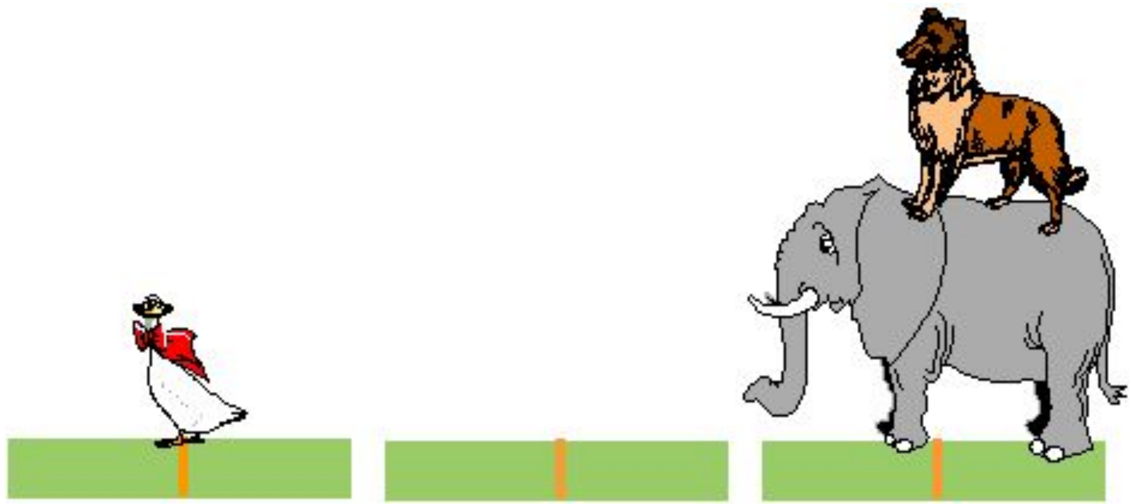


# The Towers of Hanoi



A B C

# The Towers of Hanoi

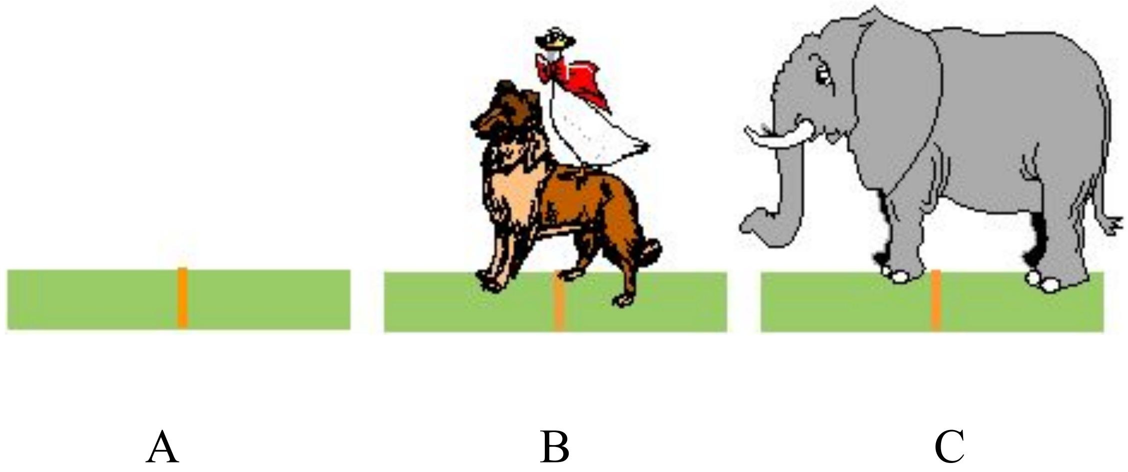


A B C

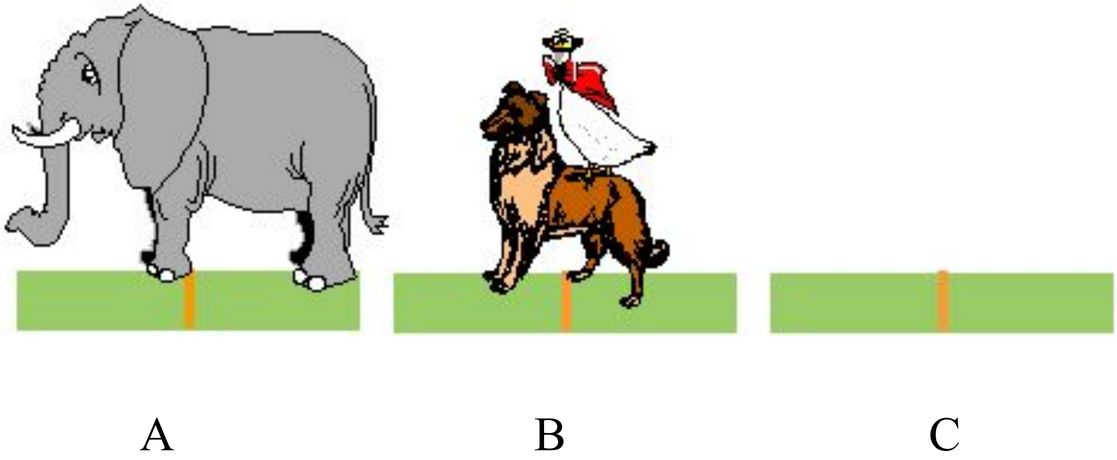
# The Towers of Hanoi

A B C

# The Towers of Hanoi



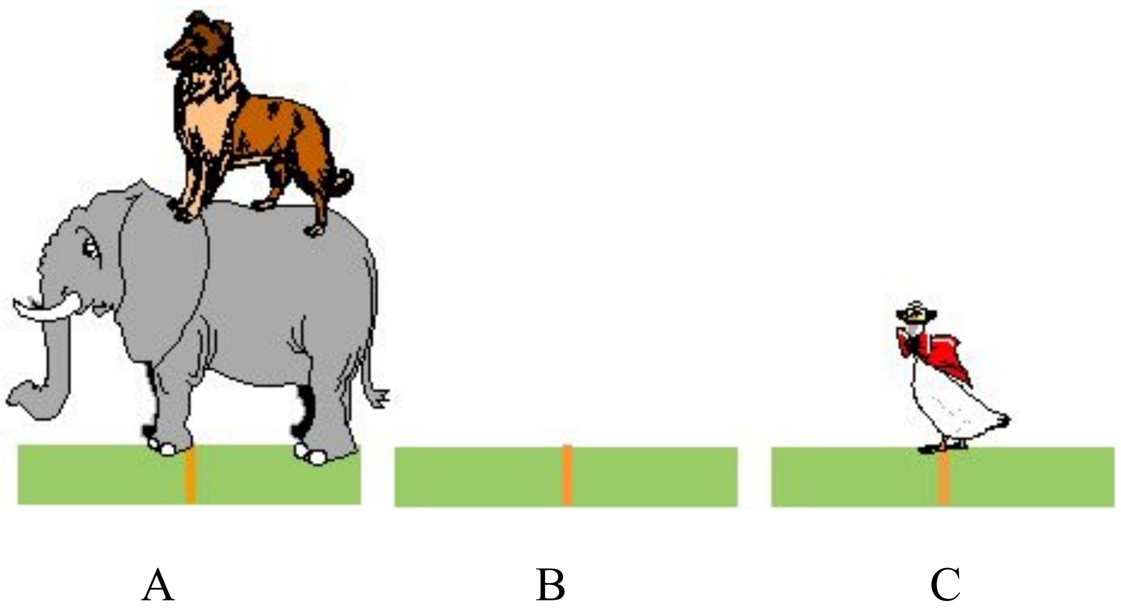
# The Towers of Hanoi



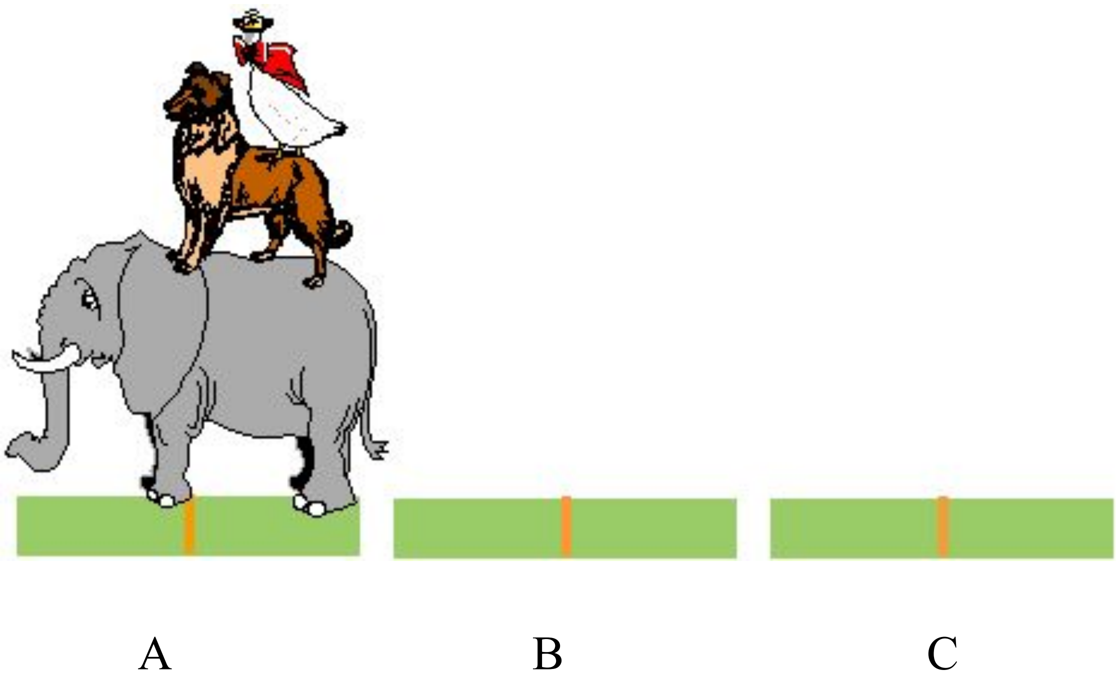
# The Towers of Hanoi

A B C

# The Towers of Hanoi

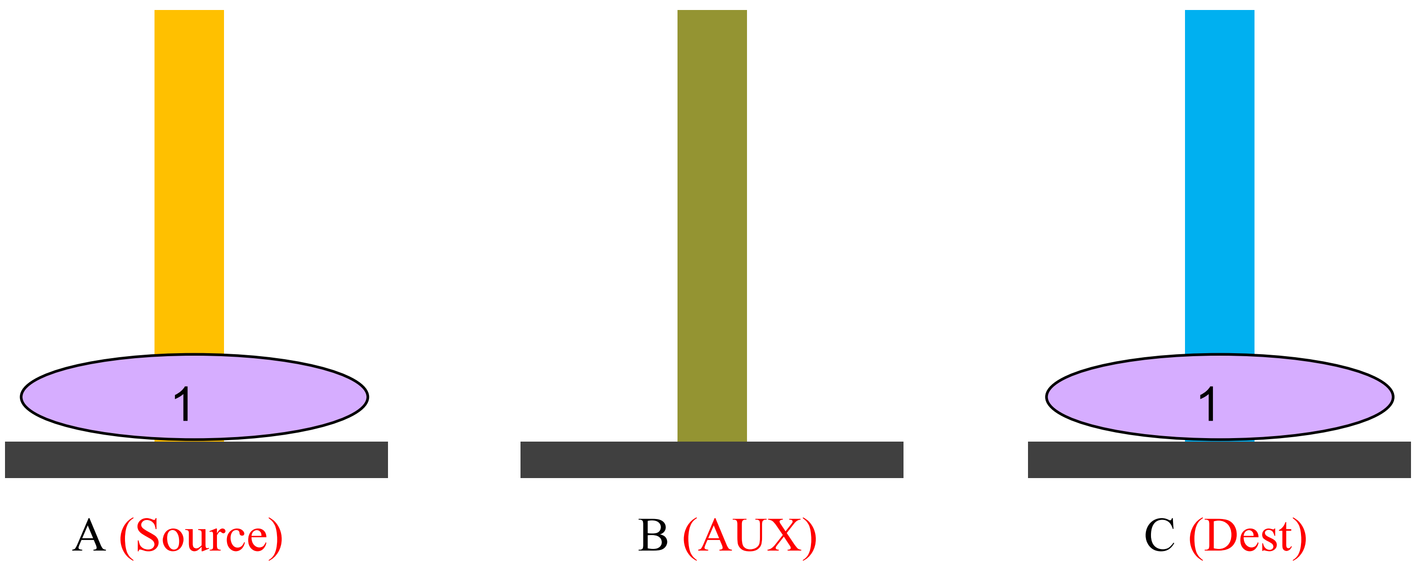


# The Towers of Hanoi



# Solution for Single Disc

**Move a disc from A to C**

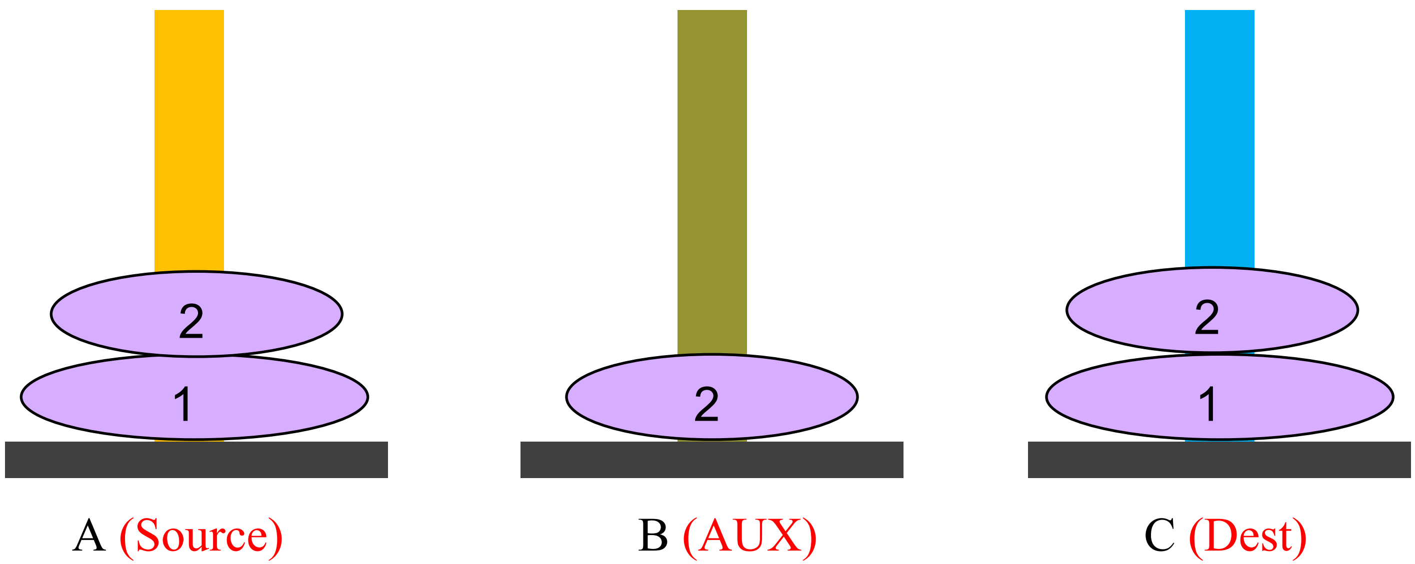


# Solution for 2 Discs

***Move a disc from A to B Using C***

***Move a disc from A to C***

***Move a disc from B to C Using A***

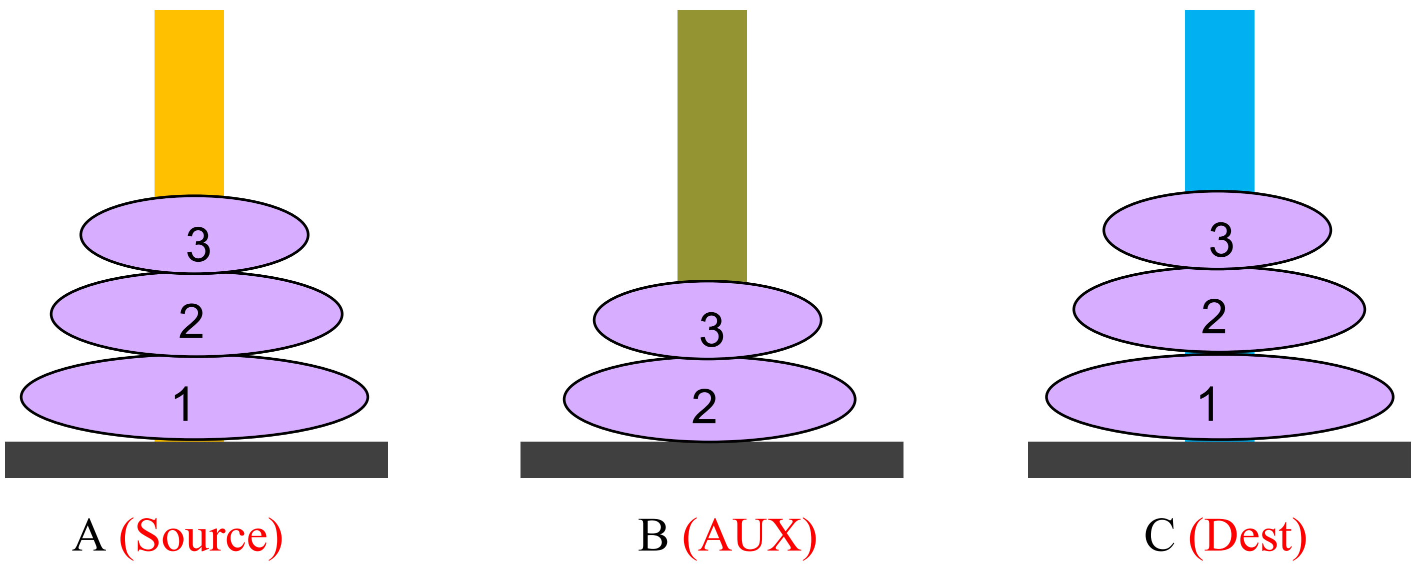


# Solution for 3 Discs

*Move 2 discs from A to B Using C*

*Move a disc from A to C*

*Move 2 disc from B to C Using A*

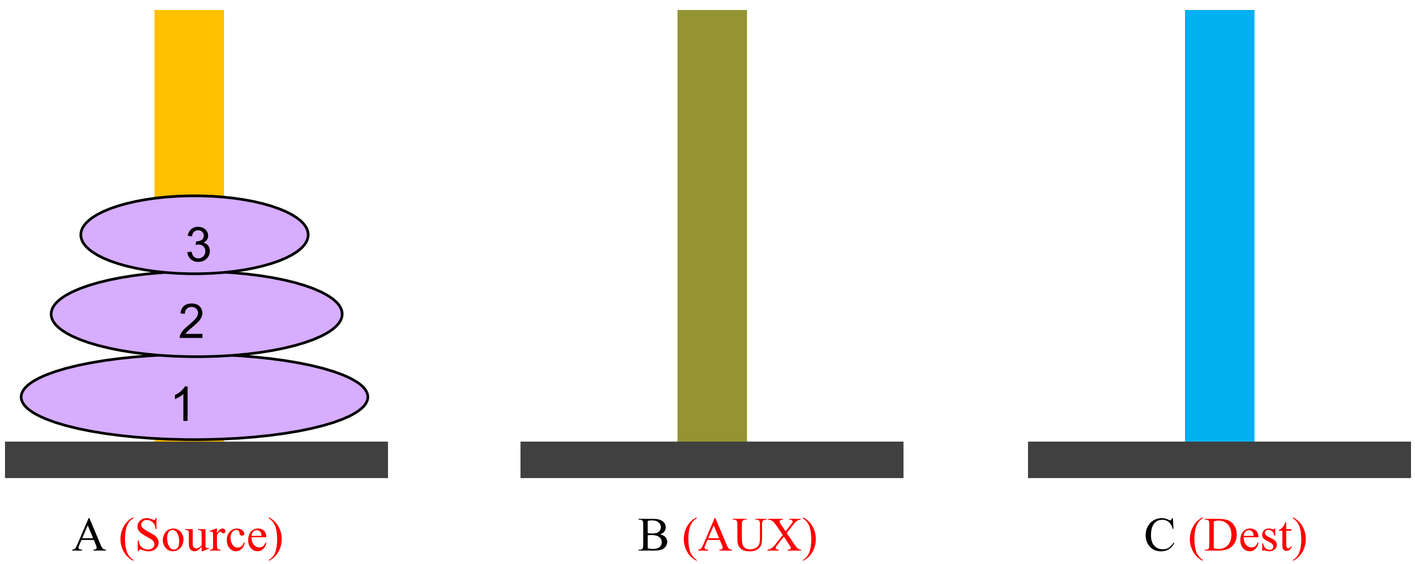


# Solution for n Discs

***Move n-1 discs from A to B Using C***

***Move a disc from A to C***

***Move n-1 disc from B to C Using A***



# Solution for n Discs

*Move n-1 discs from A to B Using C*

*Move a disc from A to C*

*Move n-1 disc from B to C Using A*

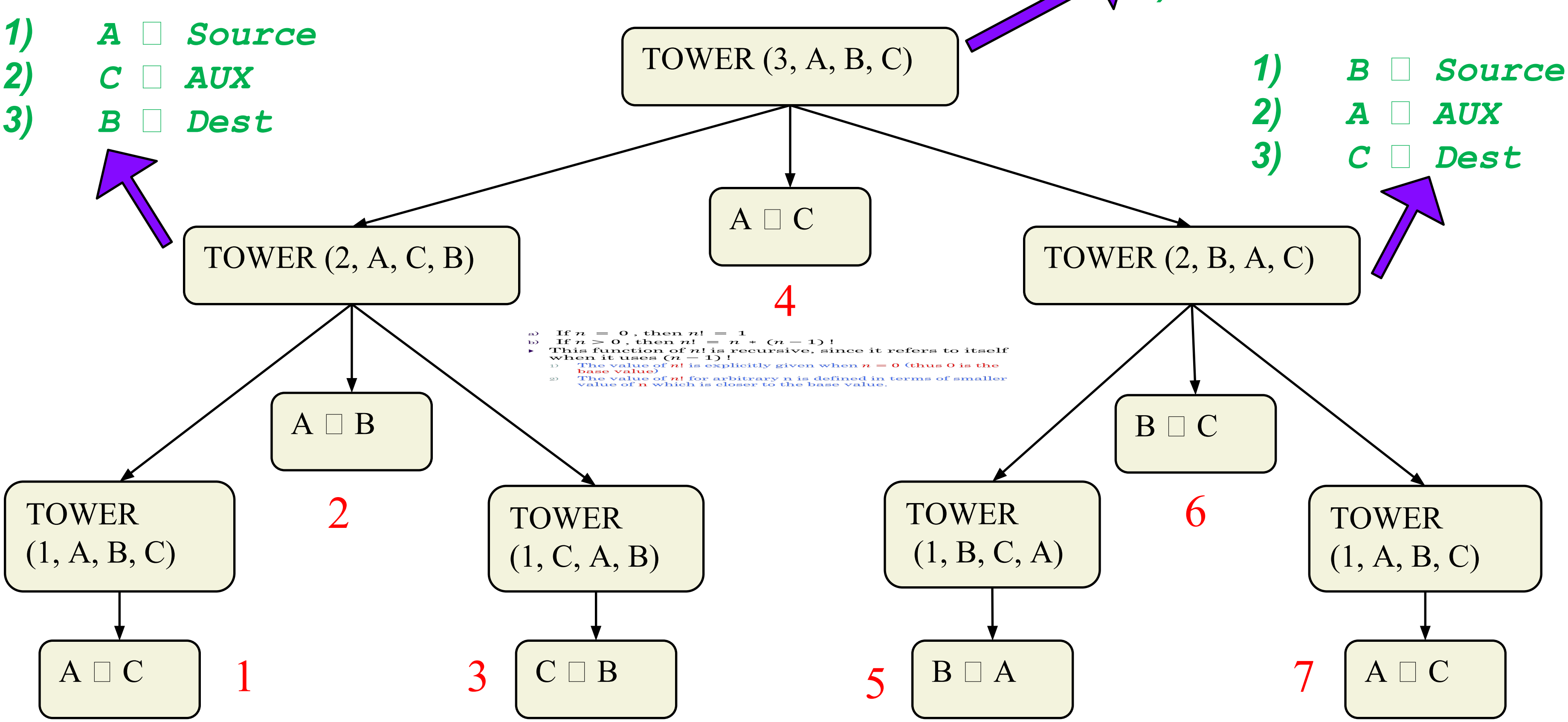
*Program*

1) *TOWER (n-1, A, C, B)*
2) *Printf (Move a disc from A to C)*
3) *TOWER (n-1, B, A, C)*

# Tracing for 3 Discs

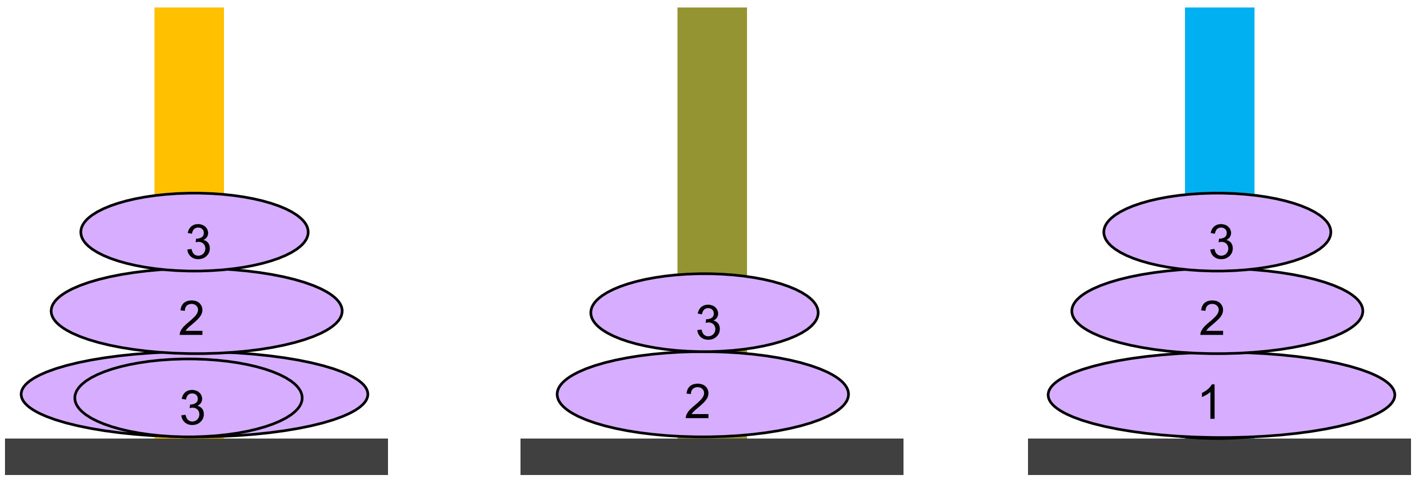
```
Program
1)TOWER (n-1, Source, Dest, AUX)
2)Printf (Move a disc from Source to Dest)
3)TOWER (n-1, AUX, Source, Dest)
```

1) A □ Source
2) B □ AUX
3) C □ Dest




a) If $n = 0$, then $n! = 1$
b) If $n > 0$, then $n! = n * (n-1)!$
- This function of $n!$ is recursive, since it refers to itself when it uses $(n-1)!$
  1) The value of $n!$ is explicitly given when $n = 0$ (thus 0 is the base value)
  2) The value of $n!$ for arbitrary n is defined in terms of smaller value of n which is closer to the base value.

# Solution for 3 Discs




A (Source) B (AUX) C (Dest)

A □ C  A □ B  C □ B  A □ C  B □ A  B □ C  A □ C

# Solution for 4 Discs

****Self:***

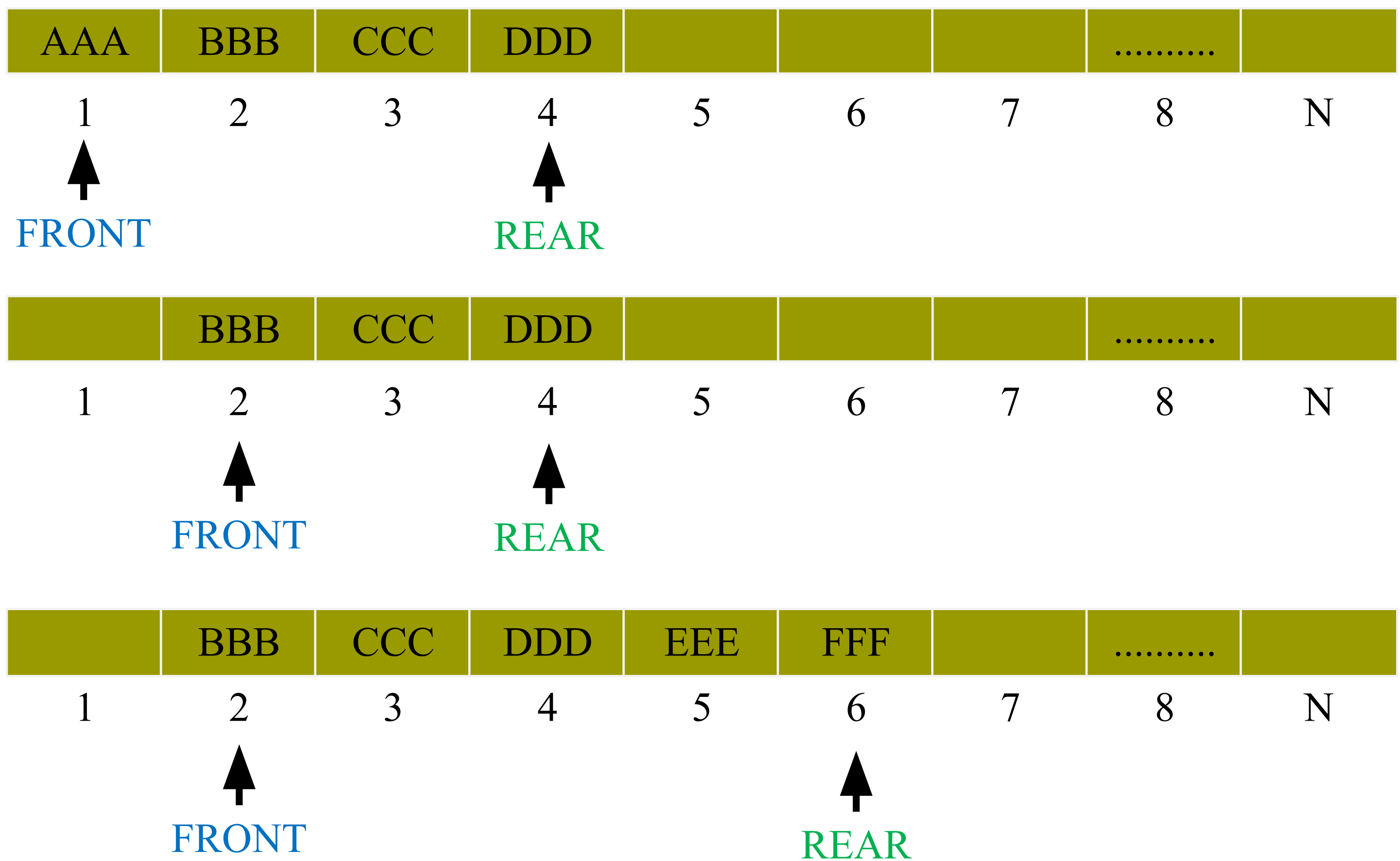
***Recursive Solution to Tower of Hanoi problem for n = 4***

# What is Queue?

- A Queue: is a linear list of elements in which deletions can take place only at one end, called FRONT, and
- Insertions can take place only at other end, called the REAR.

- FIRST IN, FIRST OUT (FIFO) property
  - The first element in a queue will be the first element out of the queue.
    - Analogy
      - Automobiles waiting
      - People waiting in a bank
      - Timesharing system in CS

# Array Representation of Queue



# Array Representation of Queue

# Circular Queue

Initially Empty

| | | | | |
|---|---|---|---|---|
| | | | | |
| 1 | 2 | 3 | 4 | 5 |

Insert (A, B, C )

| | | | | |
|---|---|---|---|---|
| A | B | C | | |
| 1 | 2 | 3 | 4 | 5 |
| FRONT | | REAR | | |

Delete ( A )

| | | | | |
|---|---|---|---|---|
| | B | C | | |
| 1 | 2 | 3 | 4 | 5 |
| | FRONT | REAR | | |

Insert ( D, E )

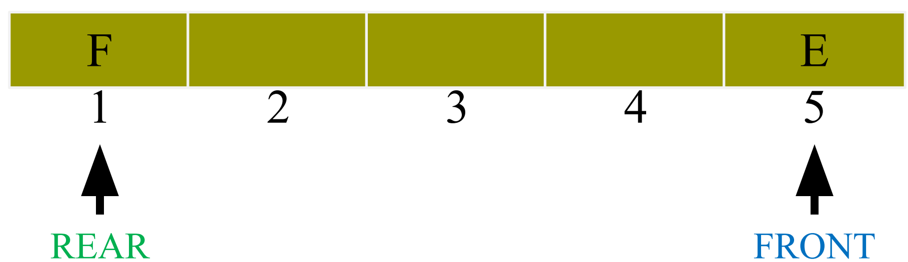
| | | | | |
|---|---|---|---|---|
| | B | C | D | E |
| 1 | 2 | 3 | 4 | 5 |
| | FRONT | | | REAR |

# Circular Queue

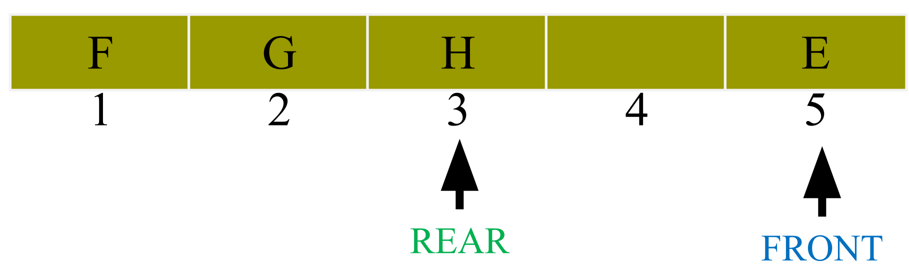
Delete ( B, C )


Insert ( F )


Delete ( D )

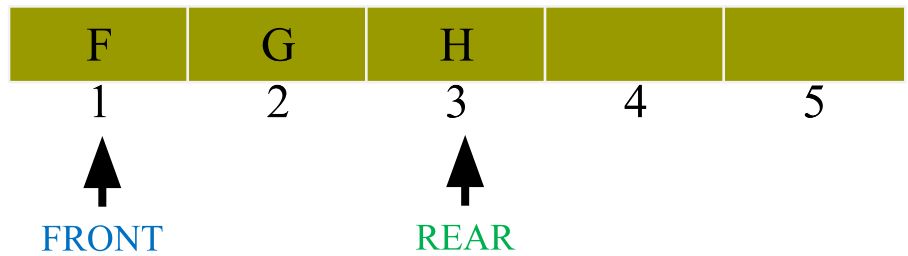



Insert ( G, H )

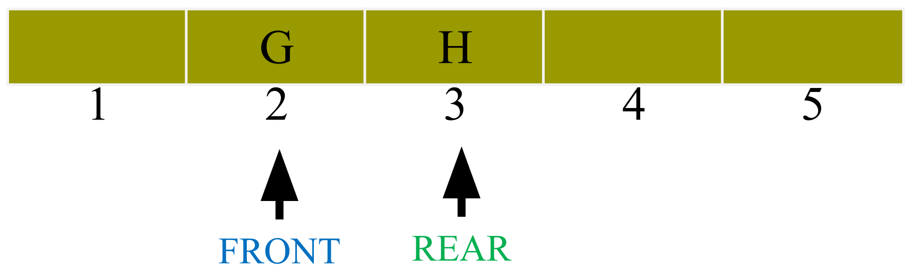


# Circular Queue

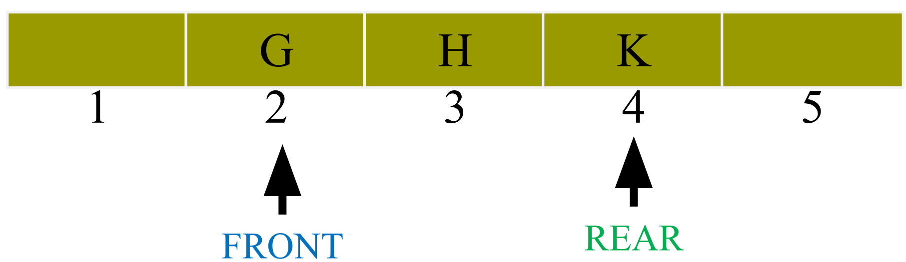
Delete ( E )



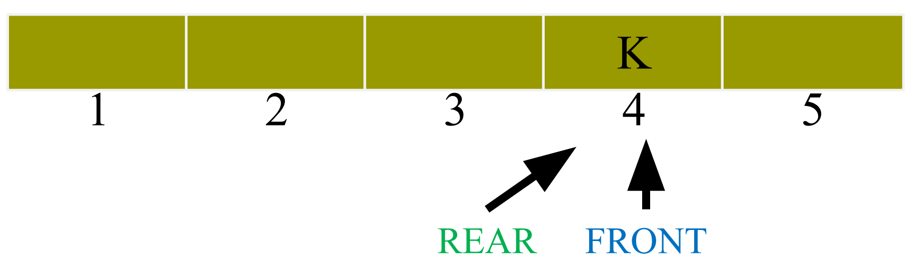

Delete ( F )




Insert ( K )




Delete ( G, H )



# Algorithm 1 (Enqueue)

QINSERT (QUEUE, N, FRONT, REAR, ITEM)

1. If FRONT = 1 and REAR = N, or if FRONT = REAR + 1, then;
    write: Overflow, and Return
2. [Find new value of REAR]
    If FRONT : = NULL, then [Queue initially Empty]
        Set FRONT : = 1 and REAR := 1
    Else if REAR = N, then :
        Set REAR : = 1
    Else:
        Set REAR : = REAR + 1

3. Set QUEUE [REAR] : = ITEM [This inserts new element]
4. Return

# Algorithm 2 (Dequeue)

QDELETE (QUEUE, N, FRONT, REAR, ITEM)

1. [Queue already empty ?]
   If FRONT = NULL , then; Write: UNDERFLOW, and Return
2. Set ITEM = QUEUE [FRONT]
3. [Find new value of FRONT]
   If FRONT = REAR, then: [Queue has only one element to start]
      Set FRONT : = NULL and REAR := NULL
   Else if FRONT = N, then :
      Set FRONT = 1
   Else:
      Set FRONT = FRONT + 1
4. Return

# Priority Queue

- Collection of elements  such that each element assigned a priority and
- The order in which elements are deleted and processed comes from the following rules :
  1) An element of higher priority is processed before any element of lower priority
  2) Two elements with same priority are processed according to the order in which they were added to the queue
- A prototype of a priority queue is time sharing system:
- Programs of high priority are processed first, and Programs with same priority form a standard queue

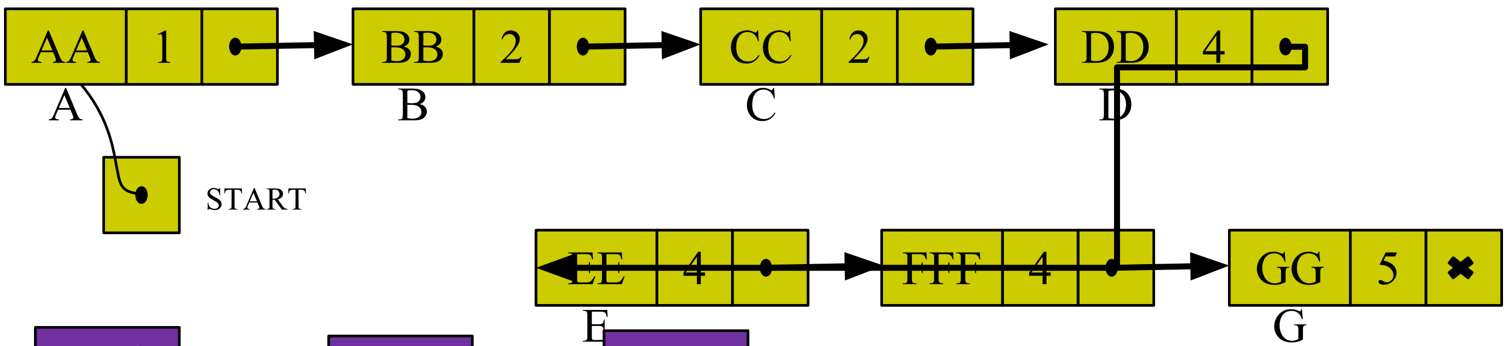
# One Way List Representation

- Each node in the list will contain three items of information: an information field Info, a priority number PRN and a link number LINK.

- A node X precedes a node Y in the list (1) when X has higher priority than Y or (2) when both have the same priority but X was added to the list before Y.

- This means that the order in the one way list corresponds to the order of the priority queue.

# One Way List Representation




| | INFO | PRN | LINK |
|---|---|---|---|
| 1 | BBB | 2 | 6 |
| 2 | | | 7 |
| 3 | DDD | 4 | 4 |
| 4 | EEE | 4 | 9 |
| 5 | AAA | 1 | 1 |
| 6 | CCC | 2 | 3 |
| 7 | | | 10 |
| 8 | GGG | 5 | 0 |
| 9 | FFF | 4 | 8 |
| 10 | | | 11 |
| 11 | | | 12 |
| 12 | | | 0 |

One Way List Representation of Queue

# Adding Element in PQ

a) Traverse the one way-list until finding a node X whose priority number exceeds N. Insert ITEM in front of node X.

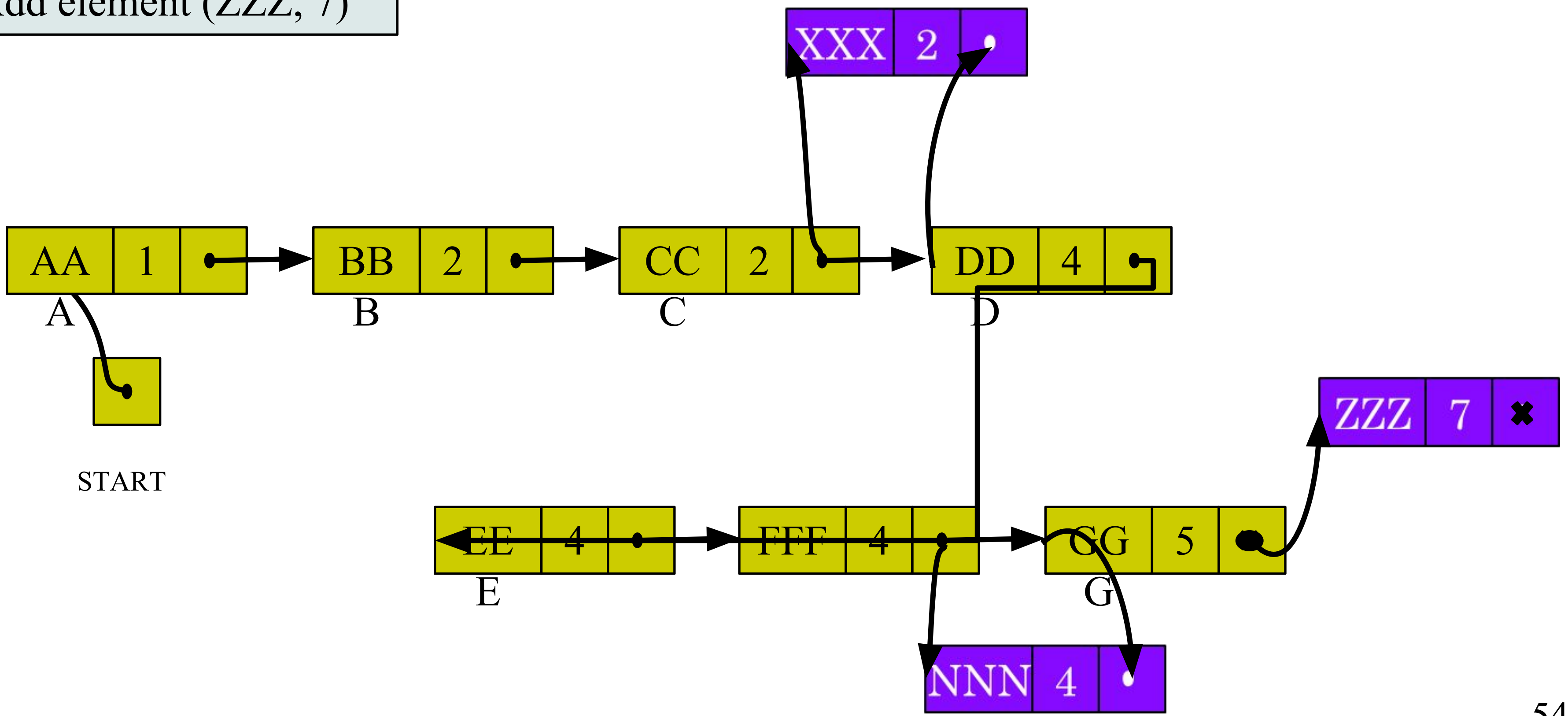
b) If no such node is found, insert ITEM as the last element of the list.

# Adding Elements in PQ

Add element (XXX, 2)

Add element (NNN, 4)

Add element (ZZZ, 7)



Thank You