CSE-213
(Data Structure)

Lecture on
Stacks and Queues

Md. Jalal Uddin

Lecturer

Department of CSE

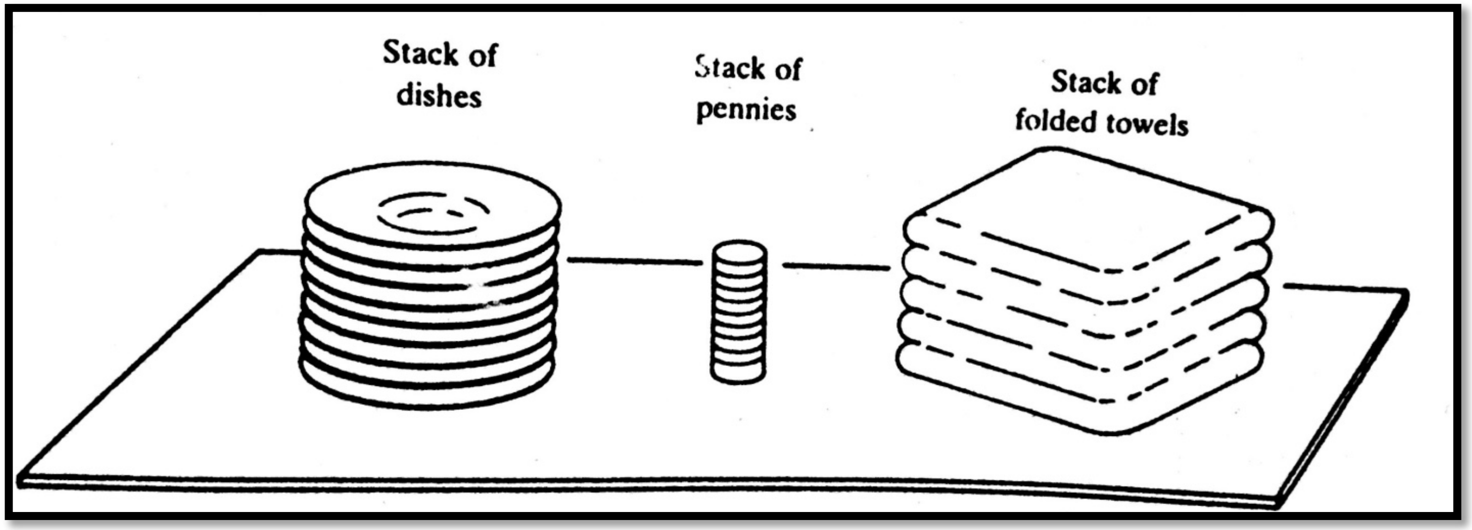
City University

Email: jalalruice@gmail.com

No: 01717011128 (Emergency Call)

Department of Computer Science & Engineering (CSE)
City University, Khagan, Birulia, Savar, Dhaka-1216, Bangladesh

STACKS: Introduction



- A stack is a linear structure in which items may be added or removed only at one ends
- Stacks are also called last-in-first-out (LIFO) list.
- Other names:
    - Piles
    - Push-down lists.
- Although the stack may be a very restricted type of data structure, it has many important applications in computer science.

STACKS: Special Terminology

Special terminology is used for two basic operations associated with stacks:

(a) “Push”-used to insert an element into a stack

(b) “Pop”-used to delete an element from a stack.

Standard Stack Operations:

The following are some common operations implemented on the stack:

push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.

pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.

isEmpty(): It determines whether the stack is empty or not.

isFull(): It determines whether the stack is full or not.'

peek(): It returns the element at the given position.

count(): It returns the total number of elements available in a stack.

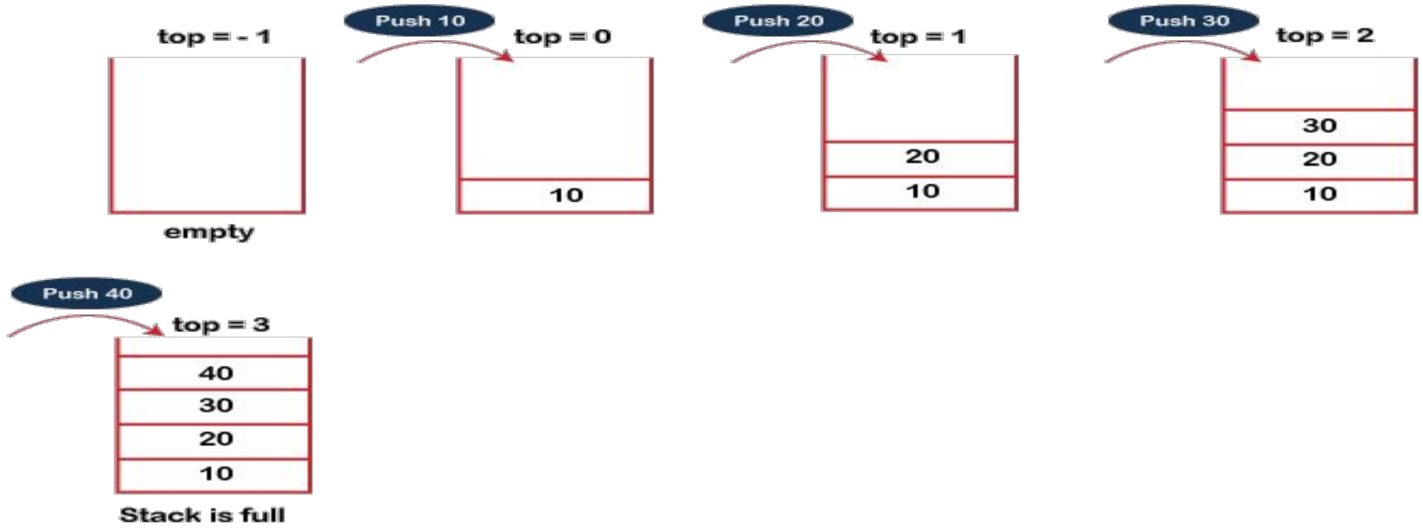
change(): It changes the element at the given position.

display(): It prints all the elements available in the stack.

PUSH operation:

The steps involved in the PUSH operation is given below:

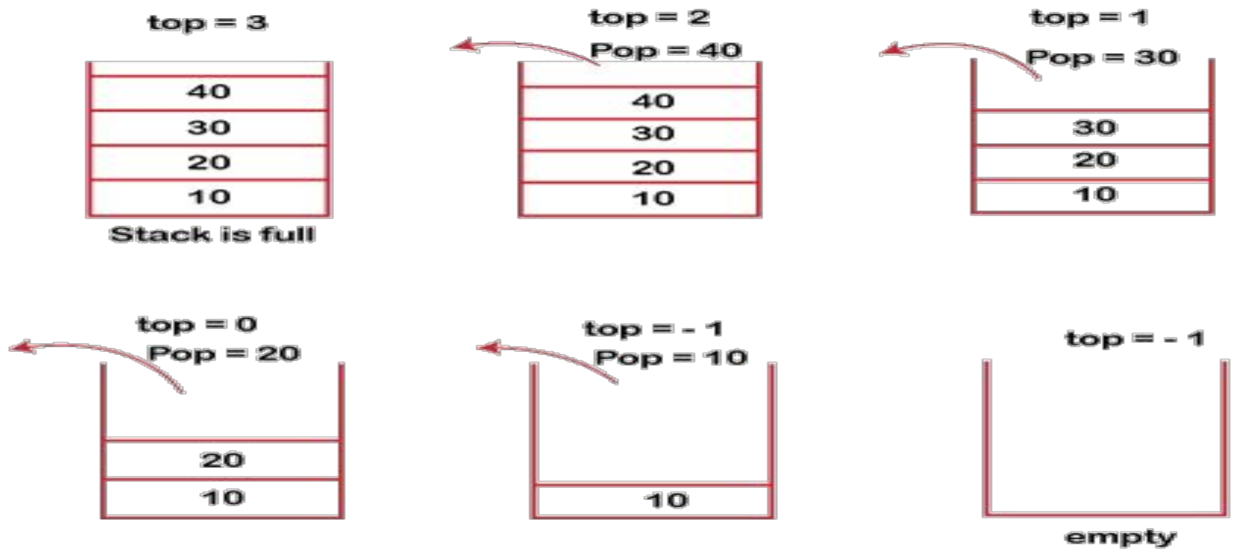
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., **top=top+1,** and the element will be placed at the new position of the **top**.
- The elements will be inserted until we reach the ***max*** size of the stack.



POP operation:

The steps involved in the POP operation is given below:

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the ***underflow*** condition occurs.
- If the stack is not empty, we first access the element which is pointed by the ***top***
- Once the pop operation is performed, the top is decremented by 1, i.e., **top=top-1**.



STACKS: Array Representation

- Stack may be represented in the computer in various ways,
  - ✔ usually by means of a one-way list, or
  - ✔ A linear array
- Usually Stacks will be maintained by……
  - A linear array, **STACK**,**(**main lists**)**
  - A pointer variable, **TOP**, **(**contains the locations of the top element of the stack**)**
  - A variable **MAXSTK**, (gives the maximum number of elements that can be held by the stack)

The condition TOP=0 or **TOP=NULL** of a **STACK** indicates…..?

EMPTY

STACKS: Array Representation Example

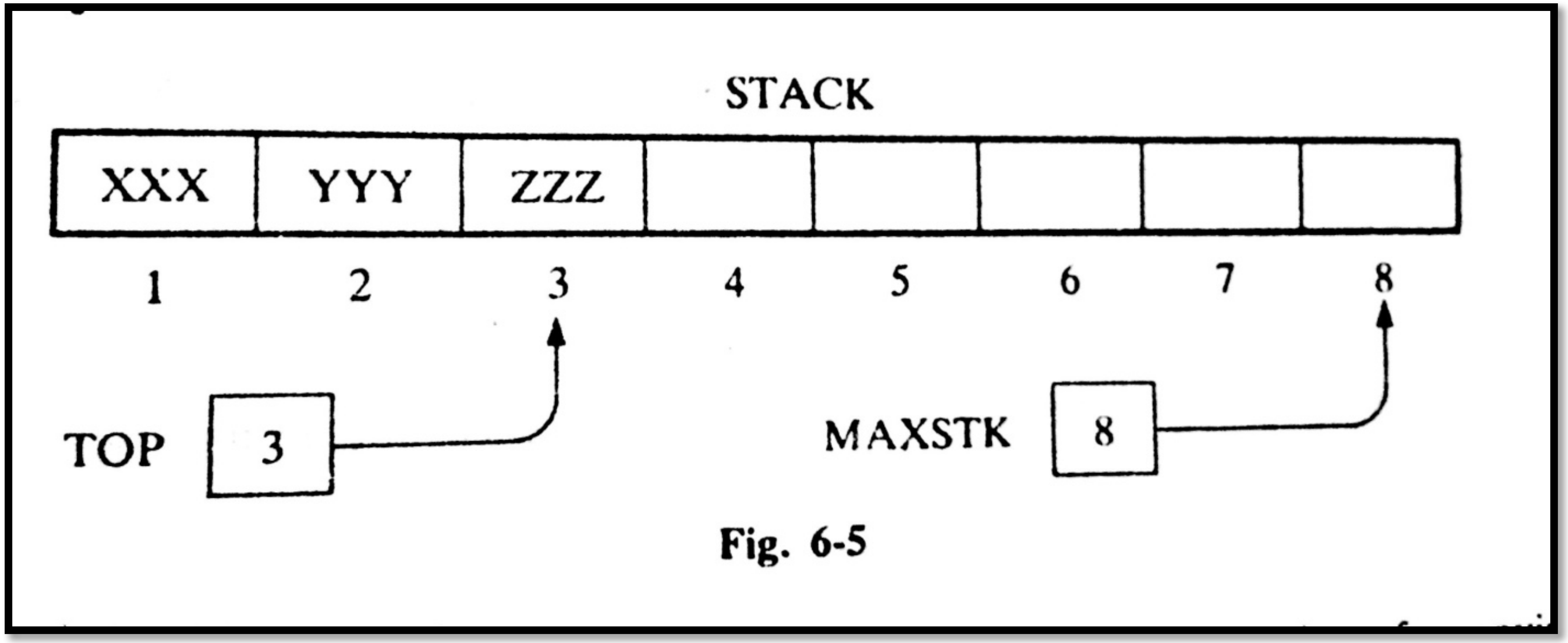


Fig. 6-5

Present Status: **TOP=3;**

MAXSTK=8;

There is room for 5 more items in the stack

Future Implementation:

Pushing an item onto a stack

Popping an item from a stack

STACKS: Pushing Algorithm

```
PUSH (STACK, TOP, MAXSTK, ITEM)
Step 1. [Stack already filled?]
    If TOP=MAXSTK, then print: OVERFLOW, and Return.
Step 2. Set TOP=TOP+1. [Increase TOP by 1.]
Step 3. Set STACK[TOP]:=ITEM. [Insert ITEM in new TOP position]
Step 4. Return
```

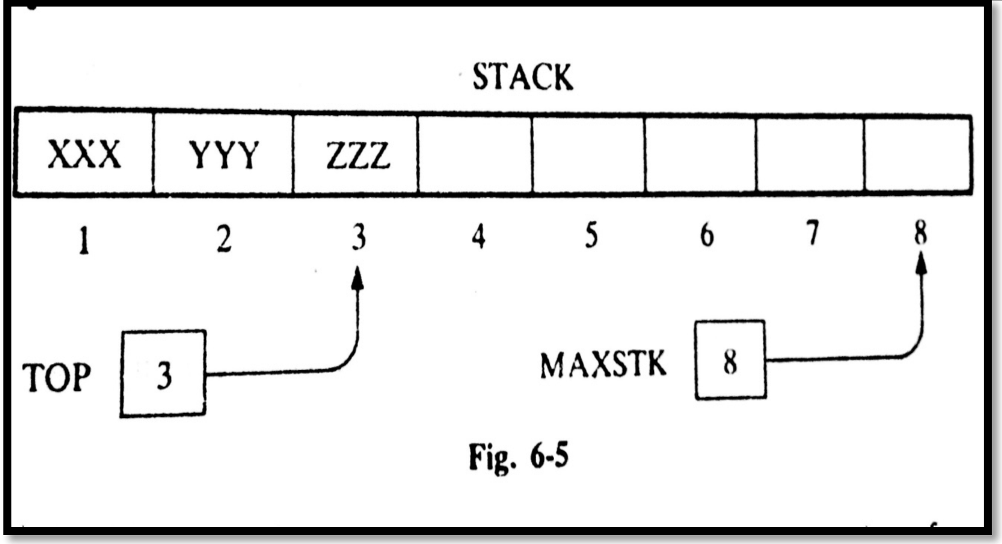


Fig. 6-5

1. Since TOP=3, go to Step 2
2. TOP=3+1=4

3. STACK [TOP]=STACK[4]=

WWW

4. Return

STACKS: Popping Algorithm

```
POP (STACK, TOP, ITEM)
Step 1. [Stack has an item to be removed?]
        If TOP=0, then print: Underflow, and Return.
Step 2. Set ITEM= STACK[TOP] [Assign TOP element to ITEM.]
Step 3. Set TOP:= TOP-1. [Decreases TOP by 1]
Step 4. Return
```

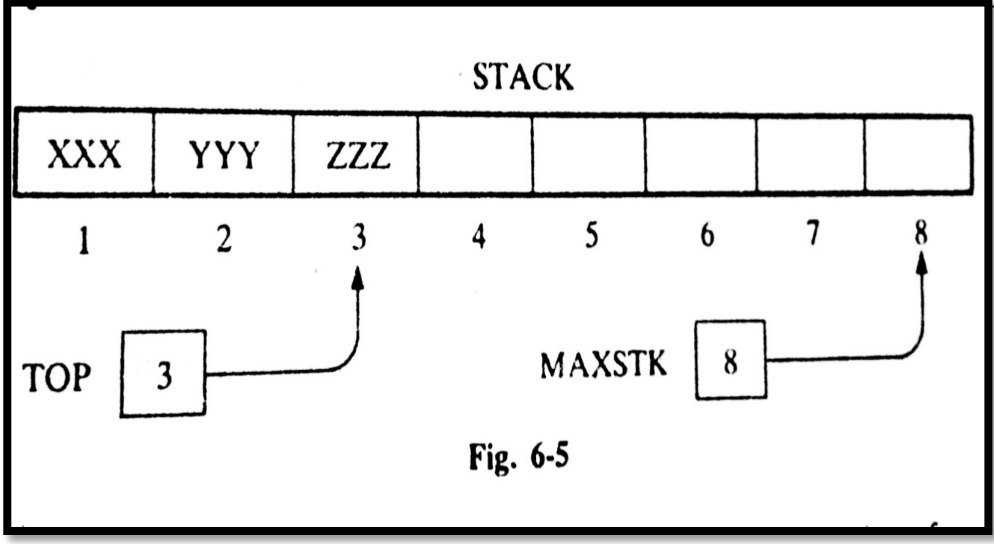


Fig. 6-5

1. Since TOP=3, go to Step 2
2. ITEM=ZZZ.

3. TOP=3-1=2.

4. Return.

STACKS:

PUSH
1. Since TOP=3, go to Step 2
2. TOP=3+1=4

3. STACK [TOP]=STACK[4]=

WWW

4. Return

POP
1. Since TOP=3, go to Step 2
2. ITEM=ZZZ.

3. TOP=3-1=2.

4. Return.

- ✔ **TOP** is changed before the insertion in PUSH, whereas
- ✔ **TOP** is changed after the deletion in POP

PUSH Syntex:

```
void push (int val,int n) //n is size of the stack
{
    if (top == n )
    printf("\n Overflow");
    else
    {
    top = top +1;
    stack[top] = val;
    }
}
```

POP Sysntex

```
int pop ()
{
   if(top == -1)
   {
      printf("Underflow");
      return 0;
   }
   else
   {
      return stack[top - - ];
   }
}
```

Satck Full Operation:

```
#include <stdio.h>

int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{

    printf("Enter the number of elements in the stack ");
    scanf("%d",&n);
    printf("Stack operations using array);

    while(choice != 4)
    {
        printf("Chose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
```

```
switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                show();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
```

```
void push ()
{
    int val;
    if (top == n )
    printf("\n Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}
```

```
void pop ()
{
    if(top == -1)
    printf("Underflow");
    else
    top = top -1;
}
void show()
{
    for (i=top;i>=0;i--)
    {
        printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
        printf("Stack is empty");
    }
}
```

Queue

- A Queue is a linear list of elements in which deletions can take place only at one end, called the front, and insertions can take place only at the other end, called the rear. The terms “front” and “rear” are used in describing a linear list only when it implemented as a queue.
- Queues are also called **first-in first-out** (FIFO) lists, since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enter a queue is the order in which they leave. This contrasts with stacks, which are Last-in First-out (LIFO) lists.

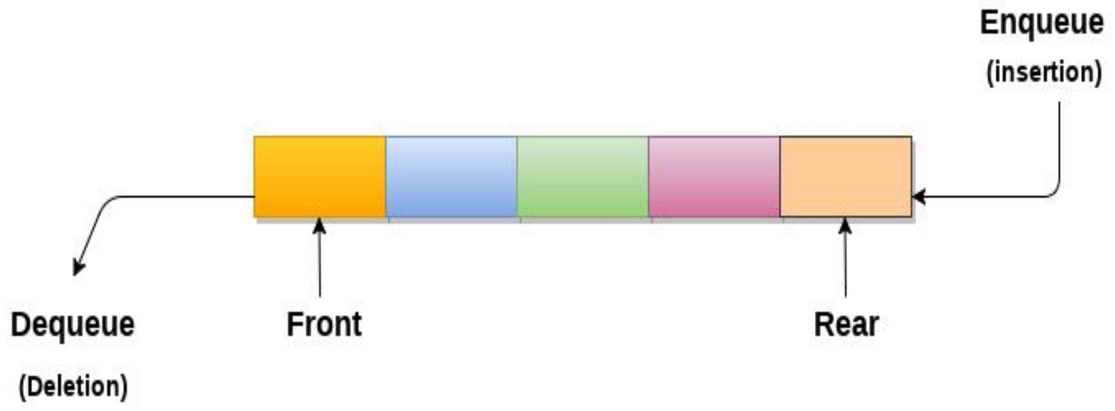
Queues abound in everyday life. For example:

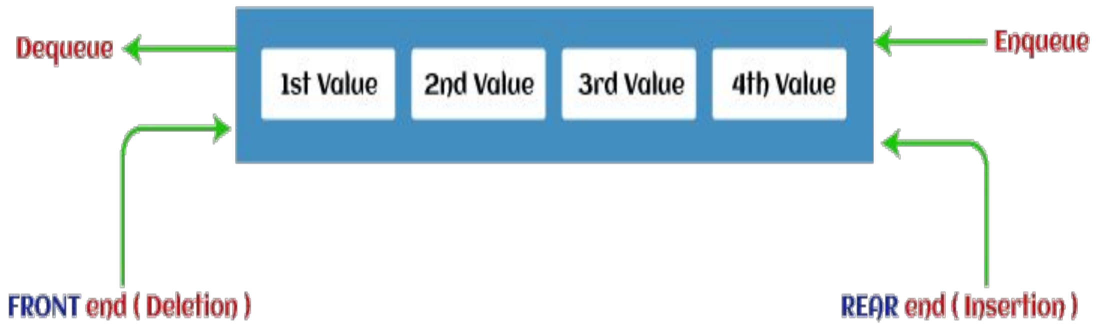
- The automobiles waiting to pass through an intersection form a queue, in which the first car in line is the first car through;
- the people waiting in line at a bank form a queue, where the first person in line is the first person to be waited on; and so on.
- An important example of a queue in computer science occurs in a timesharing system, in which programs with the same priority form a queue while waiting to be executed.
- Queue for printing purposes
- Collection of documents sent to a shared printer.

Queue:

A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.

- Queue is referred to be as First In First Out list.
- For example, people waiting in line for a rail ticket form a queue.

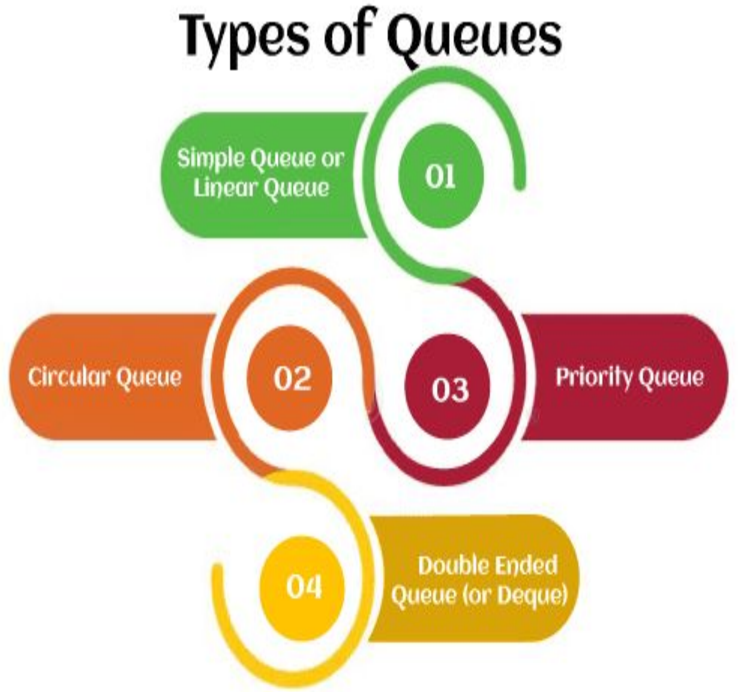




Types of Queue

There are four different types of queue that are listed as follows –

- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue



Operations performed on queue:

The fundamental operations that can be performed on queue are listed as follows -

Enqueue: The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.

Dequeue: It performs the deletion from the front-end of the queue.

Peek: This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.

Queue overflow (isfull): It shows the overflow condition when the queue is completely full.

Queue underflow (isempty): It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue

Enqueue Operation:

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success

Dequeue Operation:

Accessing data from the queue is a process of two tasks −access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation

Step 1 − Check if the queue is empty.

Step 2 − If the queue is empty, produce underflow error and exit.

Step 3 − If the queue is not empty, access the data where **front** is pointing.

Step 4 − Increment **front** pointer to point to the next available data element.

Step 5 − Return success.

Enqueue Syntex:

```
int enqueue(int data)
if(isfull())
return 0;
rear = rear + 1;
queue[rear] = data;
return 1;
end procedure
```

Dequeu Syntex:

```
int dequeue()
{
if(isempty())
return 0;
int data = queue[front];
front = front + 1;
return data;
}
```

REPRESENTATION OF QUEUE

- Suppose we want to insert an element ITEM into a queue at the time the queue does occupy the last part of the array,
- i.e. when REAR = N. One way to do this is to simply move the entire queue to the beginning of the array, changing FRONT and REAR accordingly, and then inserting ITEM as above.
- Array QUEUE is circular, that is, that QUEUE[1] comes after QUEUE[N] in the array. With this assumption, we insert ITEM into the queue by assigning ITEM to QUEUE[1]. Specifically, instead of increasing REAR to N+1, we reset REAR=1 and then assign

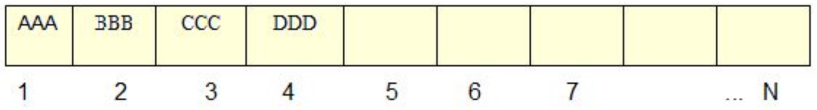
  QUEUE[REAR] := ITEM
- Similarly, if FRONT = N and an element of QUEUE is deleted, we reset FRONT = 1 instead of increasing FRONT to N + 1.
- Suppose that our queue contains only one element, i.e., suppose that
- FRONT = REAR = NULL /
- And suppose that the element is deleted. Then we assign
- FRONT := NULL and REAR := NULL To indicate that the queue is empty.

REPRESENTATION OF QUEUE

QUEUE

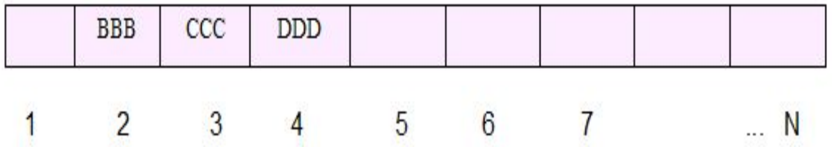
FRONT : 1

REAR : 4



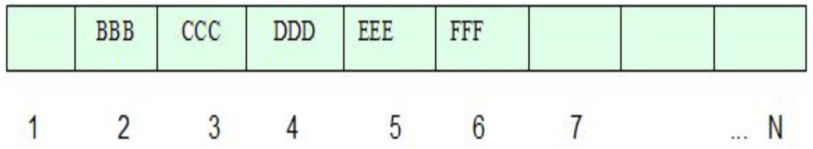
FRONT : 2

REAR : 4



FRONT : 2

REAR : 6



FRONT : 3

REAR : 6

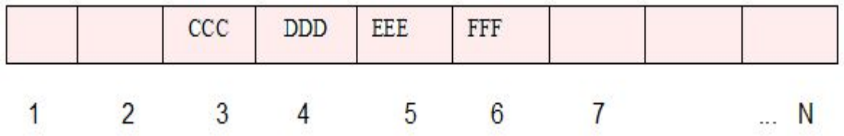


Fig : 6.16 Array representation of a queue

Example

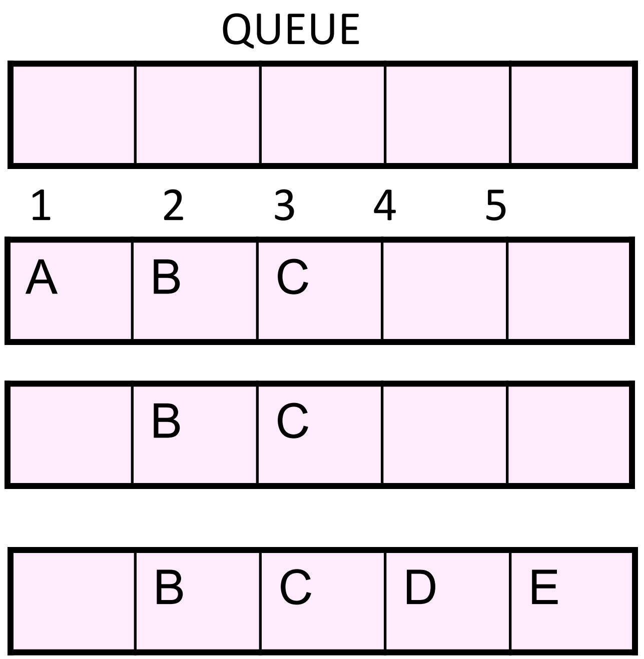
Figure 6-17 shows how a queue may be maintained by a circular array QUEUE with N = 5 memory locations. Observe that the queue always occupies consecutive locations except when it occupies locations at the beginning and at the end of the array. If the queue is viewed as a circular array, this means that is still occupies consecutive locations. Also, as indicated by fig. 6-17(m), the queue will be empty only when FRONT = REAR and an elements is deleted. For this reason, NULL is assigned to FRONT and REAR in fig. 6-17(m).

(a) Initially empty : FRONT : 0
REAR : 0

(b) A, B and then C inserted: FRONT : 1
REAR : 3

(C) A deleted : FRONT : 2
REAR : 3

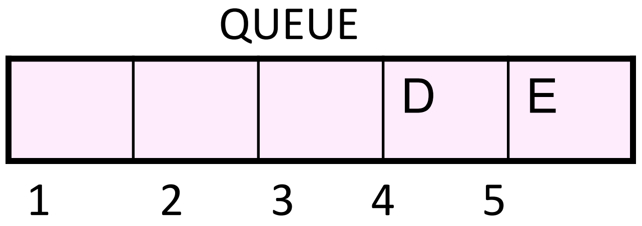
(d) D and then E inserted : FRONT : 2
REAR : 5



Example

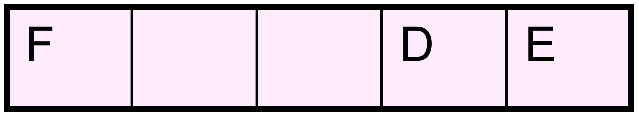
(e) B and C deleted : FRONT : 4

REAR : 5



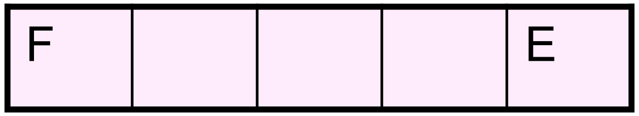
(f) F inserted : FRONT : 4

REAR : 1



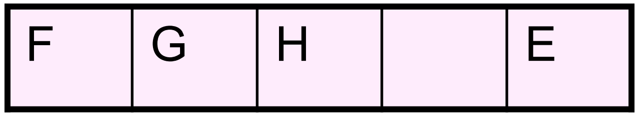
(g) D deleted : FRONT : 5

REAR : 1



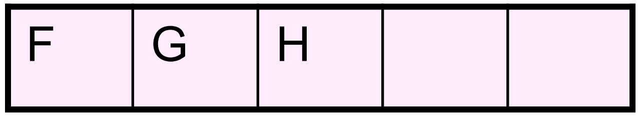
(h) G and then H inserted : FRONT : 5

REAR : 3



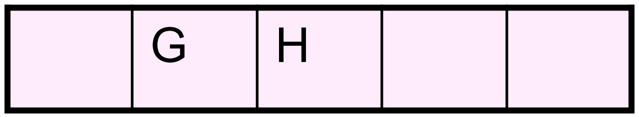
(i) E deleted : FRONT : 1

REAR : 3



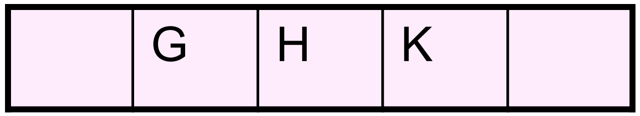
(j) F deleted : FRONT : 2

REAR : 3



(k) K inserted : FRONT : 2

REAR : 4



Example

(l) G and H deleted : FRONT : 4
REAR : 4

(m) K deleted, QUEUE is empty : FRONT : 0
REAR : 0

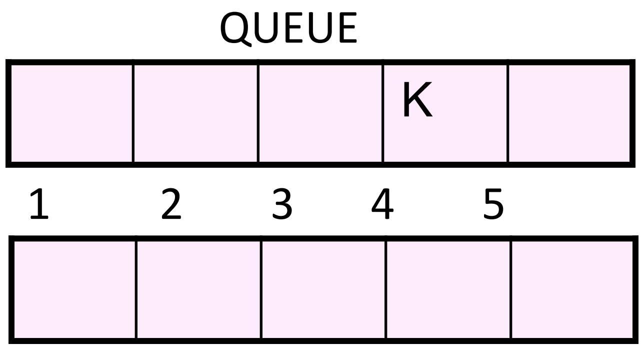


Fig :6.17

TOWER of HANOI Problem

The preceding section gave examples of some recursive definition and procedures. Recursion may be used as a tool in developing an algorithm to solve a particular problem. The problem we pick is known as Tower of Hanoi problem.

TOWER of HANOI Problem

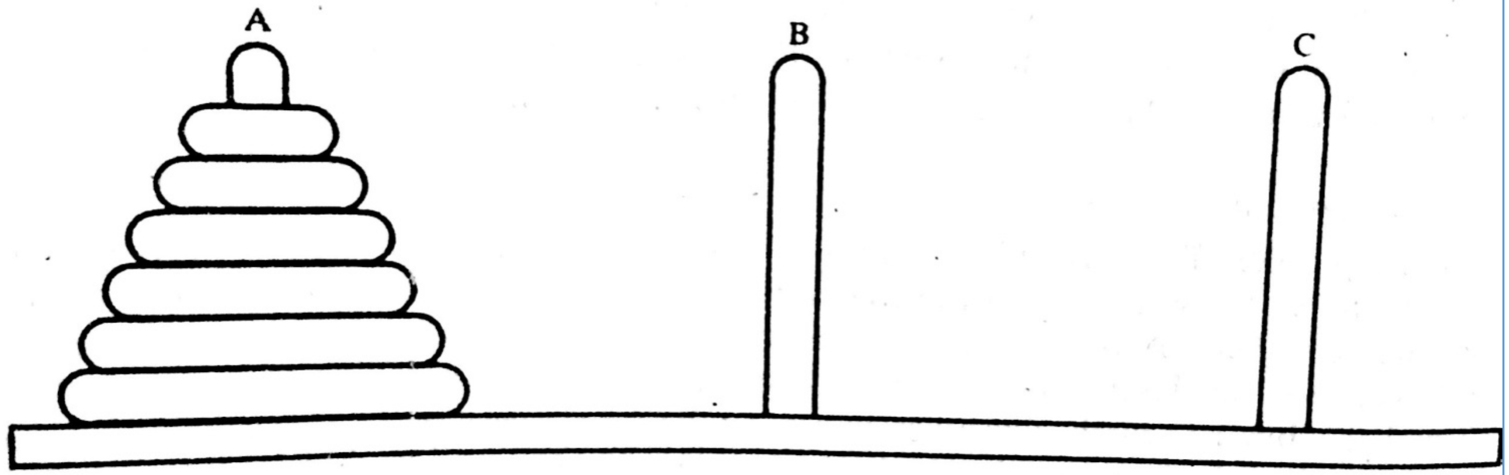


Fig. 6-10 Initial setup of Towers of Hanoi with $n = 6$.

TOWER of HANOI Problem

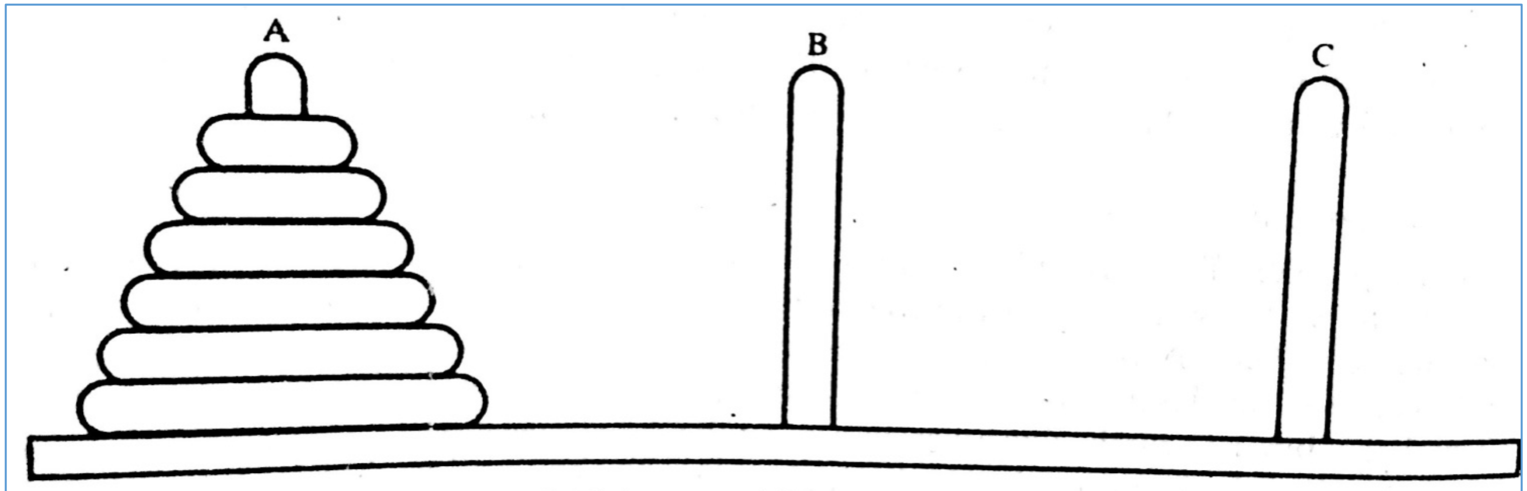
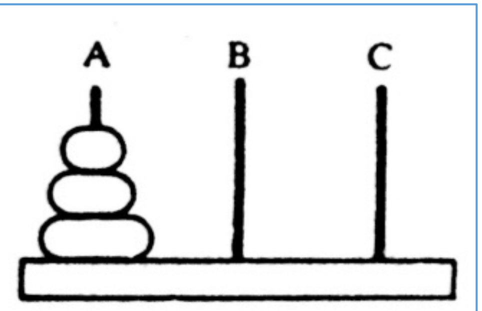


Fig. 6-10 Initial setup of Towers of Hanoi with $n = 6$.

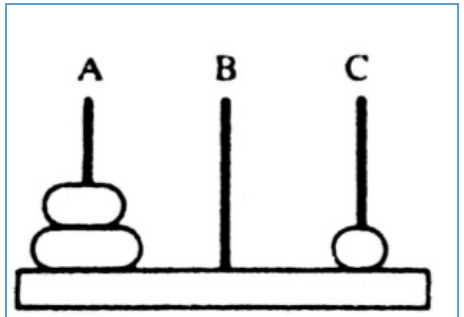
- ✔ Only one disc could be moved at a time
- ✔ A larger disc must never be stacked above a smaller one
- ✔ One and only one extra needle could be used for intermediate storage of discs

TOWER of HANOI Problem Solution

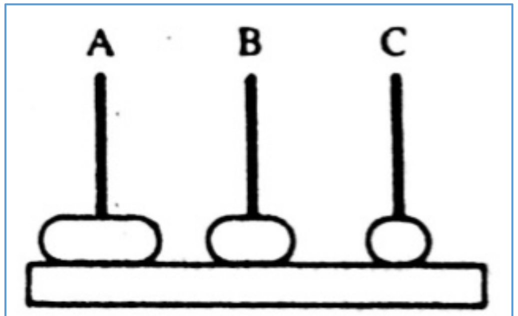
The solution to the Tower of Hanoi problem for n=3 appears



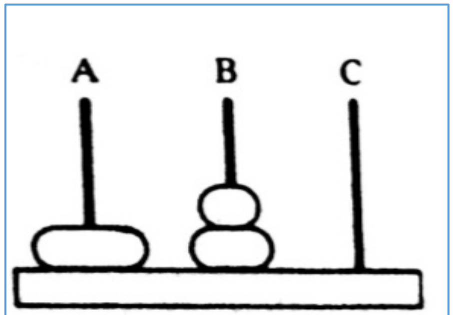
(a) Initial.



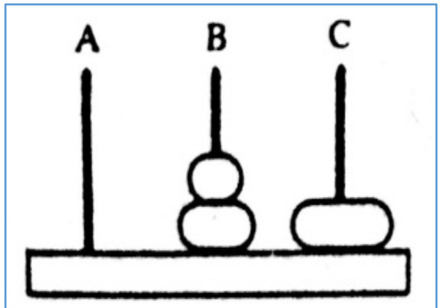
(1) A→C.



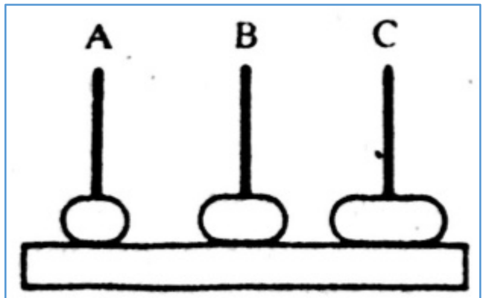
(2) A→B.



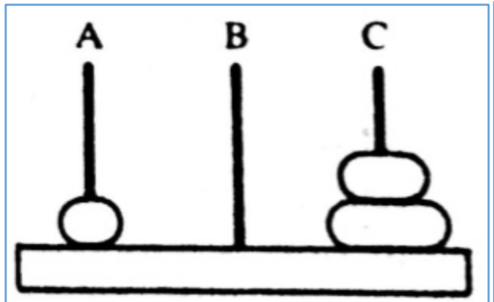
(3) C→B.



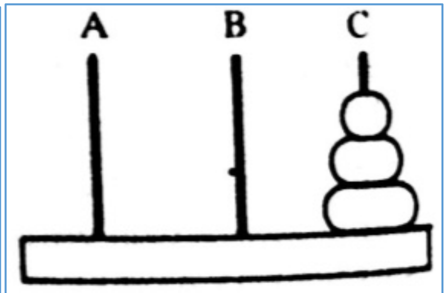
(4) A→C.



(5) B→A.



(6) B→C.



(7) A→C.

N=3: A→C, A→B, C→B, A→C, B→A, B→C, A→C,

TOWER of HANOI Problem Solution

The **SEPARATE SOLUTION** to the Tower of Hanoi problem

for **n=1**

Solution: A→C

n=2

Solution: A→B, A→C, B→C

n=3

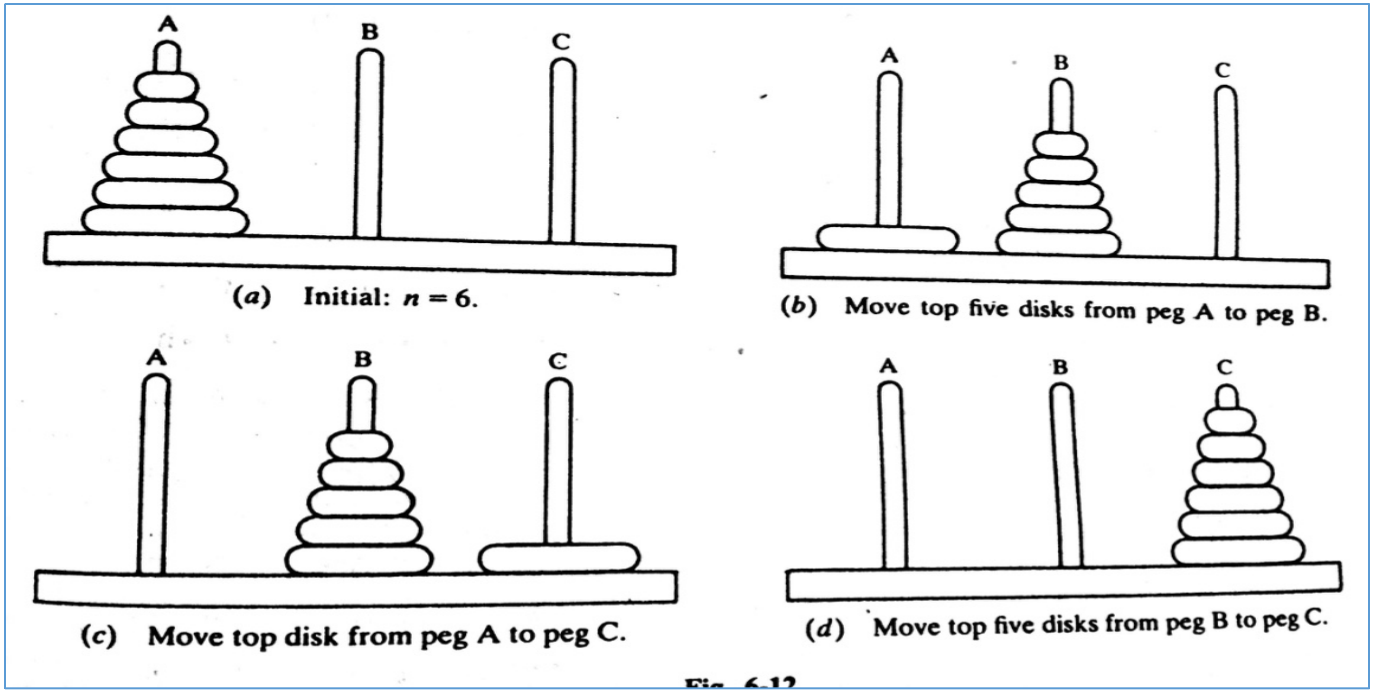
Solution: A→C, A→B, C→B, A→C, B→A, B→C, A→C

- ✔ **General Solution to the Tower of Hanoi for any # of Disk is PREFERRED.**
- ✔ **Which can be done in RECURSIVE way.**

Solution of TOWER of HANOI in **RECURSIVE WAY**

✔ The solution to the Tower of Hanoi problem for n>1 disks may be reduced to the following sub-problems:

- Move the top n-1 disks from peg A to peg B.
- Move the top disk from A to peg C: **A→C**
- Move the top n-1 disks from peg B to peg C



(a) Initial: $n = 6$.

(b) Move top five disks from peg A to peg B.

(c) Move top disk from peg A to peg C.

(d) Move top five disks from peg B to peg C.

Solution of TOWER of HANOI in **RECURSIVE WAY**

Its to denote a procedure which moves the top n disk from the initial peg BEG to the final peg END using the peg AUX as an auxiliary. When n=1, we

The general notation of the solution for any # of disk:

TOWER(N, BEG, AUX, END)

When **n=1** then

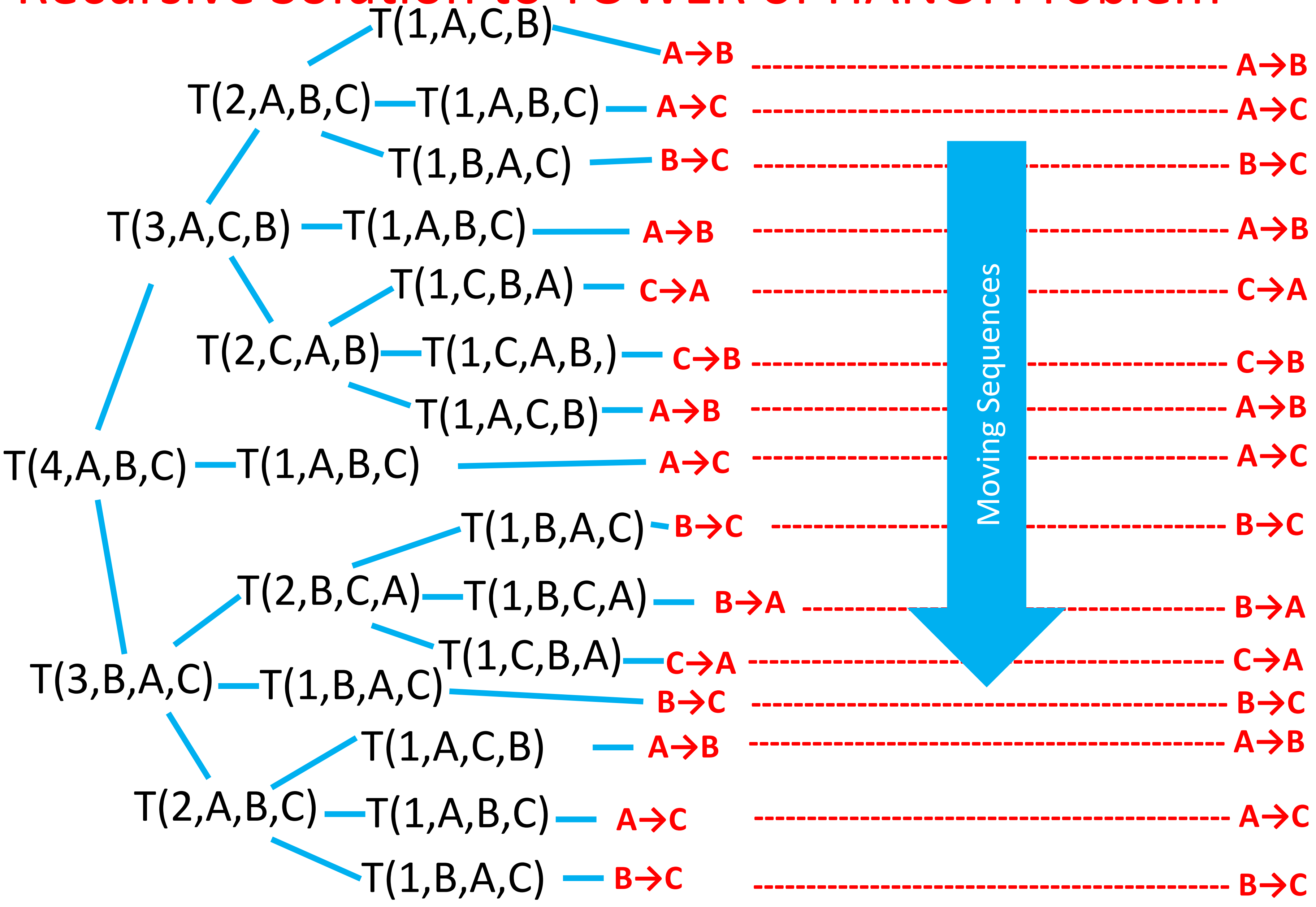
TOWER(1, BEG, AUX, END)

BEG $\rightarrow$ END, But

When **n> 1** then then solution may be reduced to the solution of the following three sub-problem:

(1) TOWER (N-1, BEG, END, AUX)
(2) TOWER (1, BEG, AUX, END)
(3) TOWER (N-1, AUX, BEG, END)

Recursive Solution to TOWER of HANOI Problem


T(1,A,C,B) A→B A→B
T(2,A,B,C) T(1,A,B,C) A→C A→C
T(1,B,A,C) B→C B→C
T(3,A,C,B) T(1,A,B,C) A→B A→B
T(1,C,B,A) C→A C→A
T(2,C,A,B) T(1,C,A,B,) C→B C→B
T(1,A,C,B) A→B A→B
T(4,A,B,C) T(1,A,B,C) A→C A→C
T(1,B,A,C) B→C B→C
T(2,B,C,A) T(1,B,C,A) B→A B→A
T(1,C,B,A) C→A C→A
T(3,B,A,C) T(1,B,A,C) B→C B→C
T(1,A,C,B) A→B A→B
T(2,A,B,C) T(1,A,B,C) A→C A→C
T(1,B,A,C) B→C B→C
Moving Sequences

Formal TOWER of HANOI Problem Solving Procedure

```
TOWER (N, BEG, AUX, END)

Step1. If N=1, then:

      (a) Write: BEG→END.

          (b) Return.

        [End of If Structure]

Step2. [Move N-1 disks from peg BEG to peg AUX.]

          Call TOWER (N-1, BEG, END, AUX).

Step3. Write: BEG → END.

Step4. [Move N-1 disks from peg AUX to peg END.]

          Call TOWER (N-1), AUG, BEG, END).

Step5. Return
```