CSE-2321
Data Structure
Part-5

Presented by

Asmaul Hosna Sadika
Adjunct Faculty
Dept of CSE, IIUC

Contents

- Queue – its representation
- Application of queue
- Types of queue
- Operations in queue
- Recursion [Factorial function, Fibonacci sequence, Towers of Hanoi]

What is Queue?

- A **queue** is a linear data structure where elements are stored in the FIFO (First In First Out) principle where the first element inserted would be the first element to be accessed.
- A queue is an Abstract Data Type (ADT) similar to stack, the thing that makes queue different from stack is that a queue is open at both its ends.
- The data is inserted into the queue through one end and deleted from it using the other end.
- Queue is very frequently used in most programming languages.
- A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Application of queue

- CPU scheduling, Disk Scheduling
- When data is transferred asynchronously between two processes. The queue is used for synchronization. For example: IO Buffers, pipes, file IO, etc.
- Handling of interrupts in real-time systems.
- Call Center phone systems use Queues to hold people calling them in order.

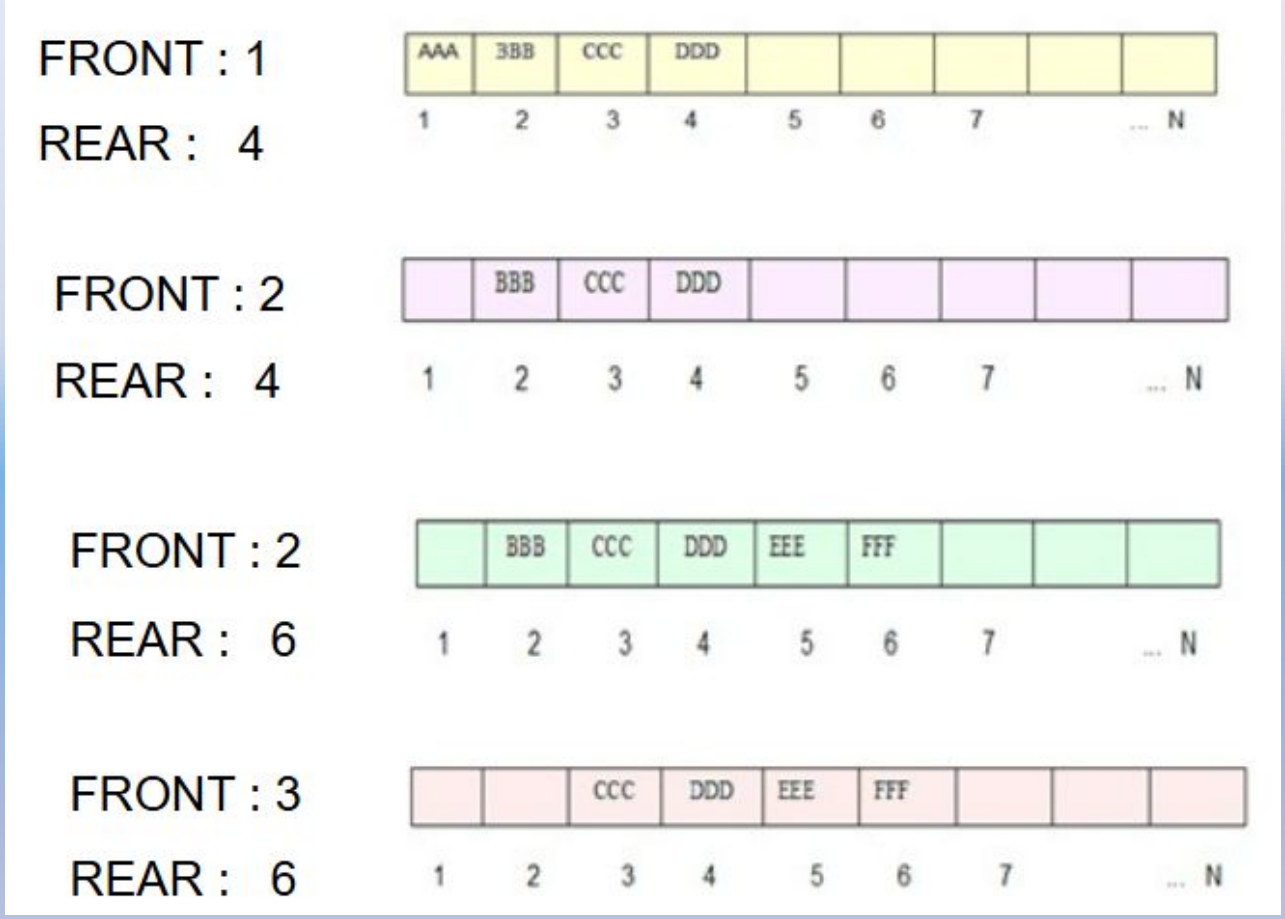
Representation of Queues

- Similar to the stack ADT, a queue ADT can also be implemented using arrays, linked lists, or pointers. We mostly implement queues using a one-dimensional array.

- Queue may be represented in the computer in various ways, usually by means of one-way lists or linear arrays.
- Each of our queues will be maintained by a linear array QUEUE and two pointer variables:
- FRONT, containing the location of the front element of the queue;
- and REAR, containing the location of the rear element of the queue.
- The condition FRONT = NULL will indicate that the queue is empty.

Queue Representation

Array representation of Queue



Array representation of Queue

when an element is deleted from the queue, the value of FRONT is increased by 1; this can be implemented by the assignment

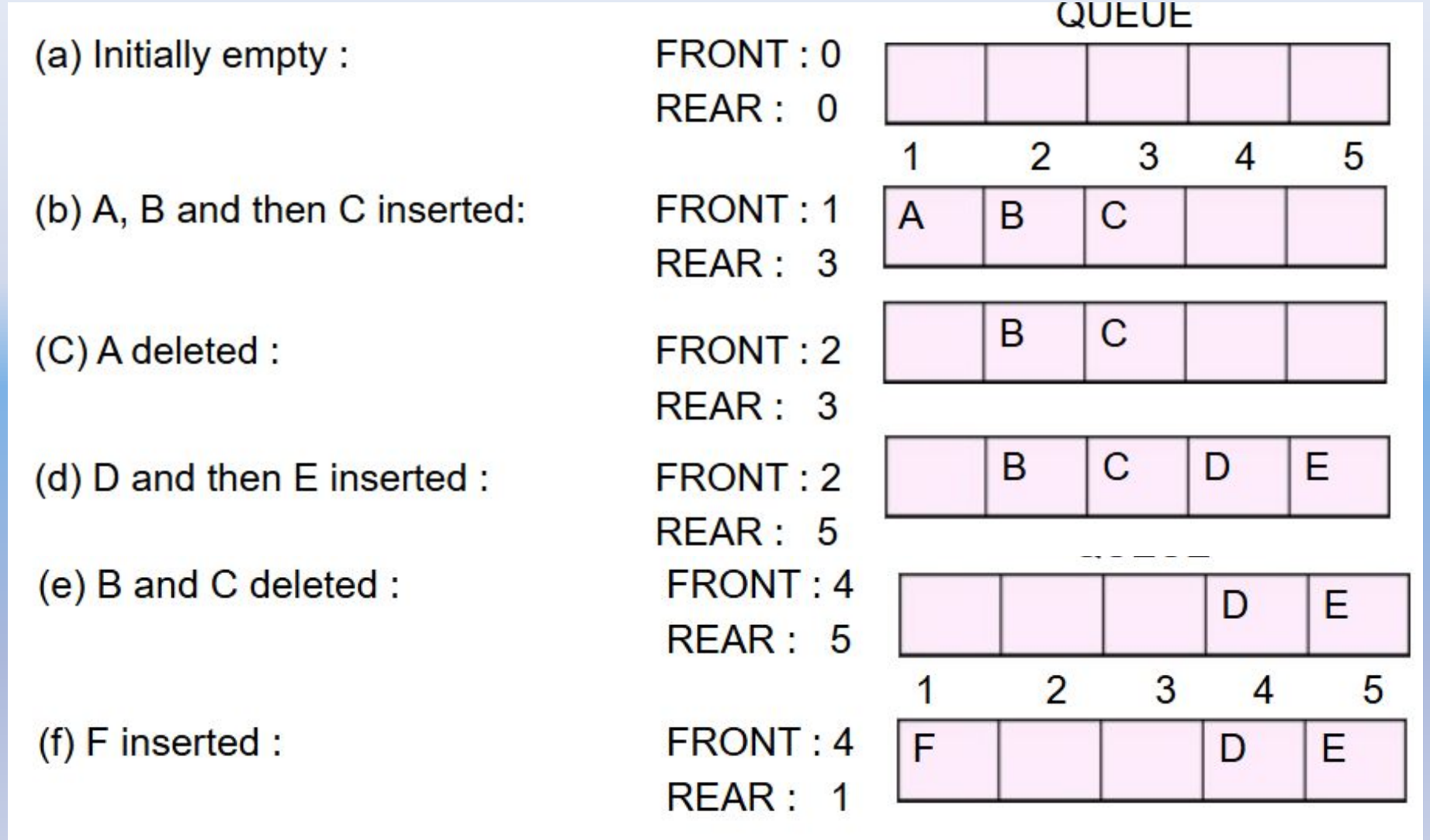
FRONT := FRONT + 1

Similarly, whenever 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

This means that after N insertions, the rear element of the queue will occupy QUEUE[N] or, in other words, eventually the queue will occupy the last part of the array. This occurs even through the queue itself may not contain many elements.

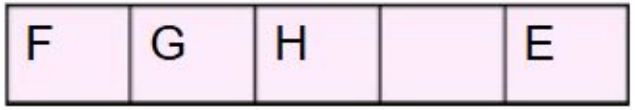
Array representation of Queue



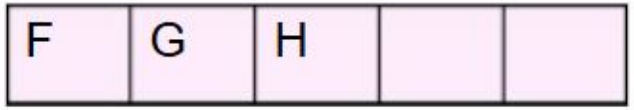
Array representation of Queue

(g) D deleted : FRONT : 5, REAR : 1

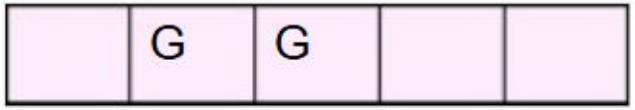
(h) G and then H inserted : FRONT : 5, REAR : 3



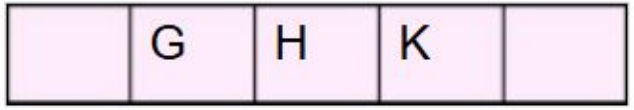
(i) E deleted : FRONT : 1, REAR : 3



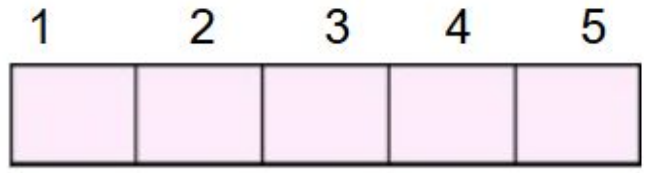
(j) F deleted : FRONT : 2, REAR : 3



(k) K inserted : FRONT : 2, REAR : 4



(l) G and H deleted : FRONT : 4, REAR : 5



(m) K deleted, QUEUE is empty : FRONT : 4, REAR : 1

Basic Operations of queue

- Queue operations also include initialization of a queue, usage and permanently deleting the data from the memory.
- The most fundamental operations in the queue ADT include:

1. **enqueue()** - Add an element to the end of the queue
2. **dequeue()** - Remove an element from the front of the queue
3. **peek()** - Get the value of the front of the queue without removing
4. **isFull()** - Check if the queue is full
5. **isEmpty()** - itCheck if the queue is empty

- These are all built-in operations to carry out data manipulation and to check the status of the queue.
- Queue uses two pointers − **front** and **rear**. The front pointer accesses the data from the front end (helping in enqueueing) while the rear pointer accesses data from the rear end (helping in dequeuing).

Types of Queue

A queue is a useful data structure in programming. It is similar to the ticket queue outside a cinema hall, where the first person entering the queue is the first person who gets the ticket.

There are four different types of queues:

1. Simple Queue
2. Circular Queue
3. Priority Queue
4. Double Ended Queue

Simple Queue

- In the simple queue, the beginning of the array will become the front for the queue and the last location of the array will act as rear for the queue.
- The following relation gives the total number of elements present in the queue, when implemented using arrays:

Total no of elem.=front-rear+1

- Also, note that if rear<front then there will be no element in the queue or queue is always be empty. Bellow show a figure a empty simple queue Q[5] which can accommodate five elements.

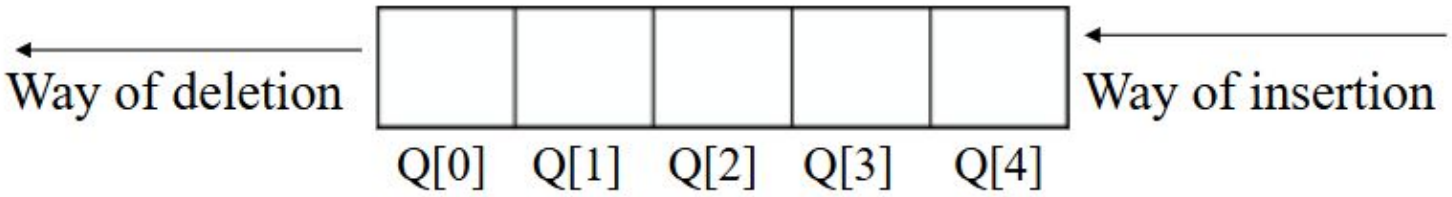


Fig: Simple Queue

Simple queue

Limitation of simple queue:

- There are certain problems associated with a simple queue when queue is implemented using array.
- Consider an example of a simple queue Q[5], which is initially empty.
- We can only five elements insertion in queue.
- If we attempt to add more elements, the elements can’t be inserted because in a simple queue rear increment up to size-1.
- At that time our program will flash the message “Queue is full”.
- But if the queue is too large of say 5000 elements it will be a tedious job to do and time consuming too.
- To remove this problem we use circular queue.

Circular queue

- A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location of the queue is full.
- In other words if we have a queue Q of say n elements, then after inserting an element last (n-1th) location of the array the next elements will be inserted at the very first location (0) of the array.
- It is possible to insert new elements, if and only if those location (slots) are empty.
- We can say that a circular queue is one in which the first element comes just after the last element.
- It can be viewed as a mesh or loop of wire, in which the two ends of the wire are connected together.
- A circular queue overcomes the problem of unutilized space in linear queues implemented as arrays.
- A circular queue also have a Front and Rear to keep the track of the elements to be deleted and inserted and therefore to maintain the unique characteristic of the queue.

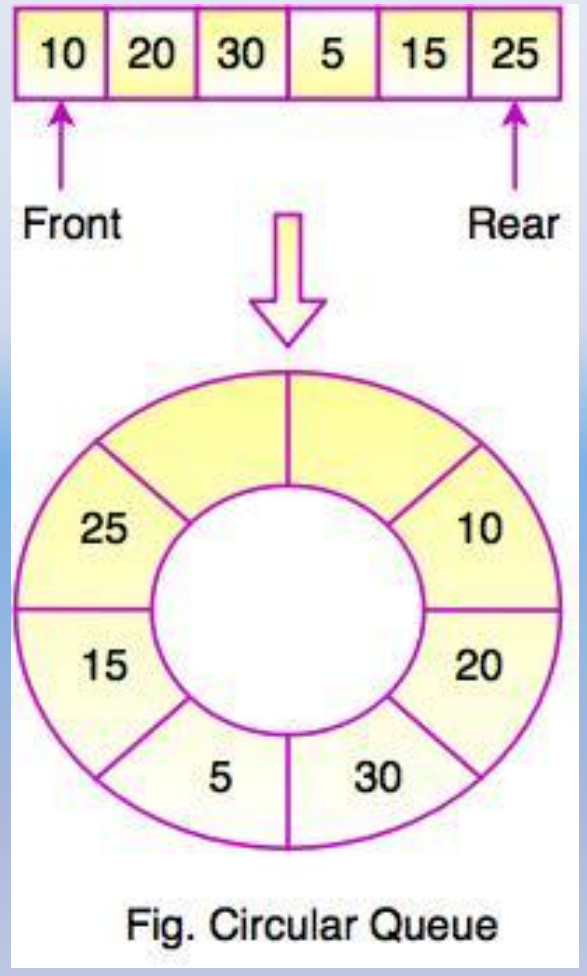
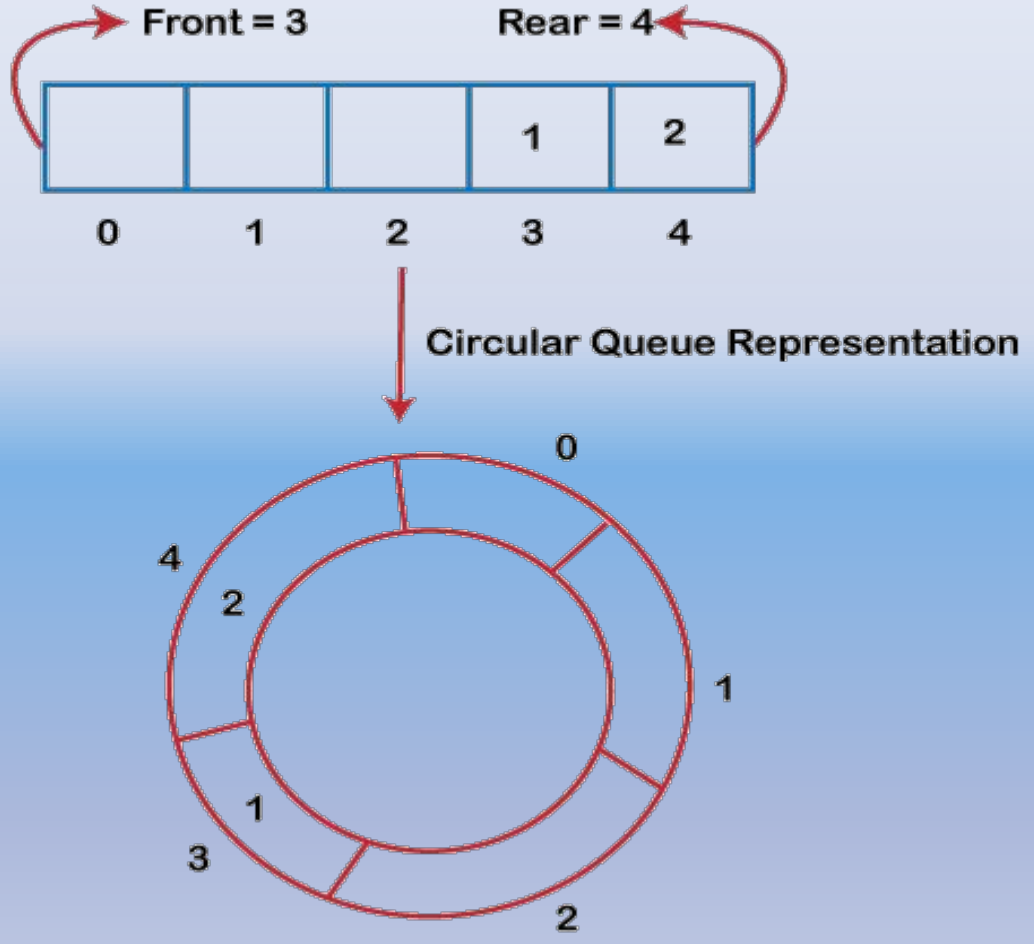


Fig. Circular Queue



Double ended queue

It is also a homogeneous list of elements in which insertion and deletion operations are performed from both the ends. That is, we can insert elements from the rear end or from the front ends.

Hence it is called double-ended queue. It is commonly referred as a Deque.

There are two types of Deque. These two types are due to the restrictions put to perform either the insertions or deletions only at one end.

Input-restricted deque An input restricted deque is a deque which allows insertion at only one end of the list but allows deletion a both end of the list.

Output-restricted deque An output restricted deque is a deque which allows deletion at only one end of the list but allows insertion a both end of the list

Priority queue

- A priority queue is a collection of elements such that each element has been assigned a priority, such that the order in which the elements are deleted and processed comes from the following rules

  –An element with higher priority will be processed before any element with lower priority.

  –Two elements with the same priority will be processed in order in which they are add to the queue.

- Insertion occurs based on the arrival of the values and removal occurs based on priority.

Queue operations

- Before inserting an element in the queue we must check for overflow conditions.
- An overflow occurs when we try to insert an element into a queue that is already full, i.e. when rear = MAX – 1, where MAX specifies the maximum number of elements that the queue can hold.
- Similarly, before deleting an element from the queue, we must check for underflow condition.
- An underflow occurs when we try to delete an element from a queue that is already empty. If front = -1 and rear = -1, this means there is no element in the queue.

Queue Insertion Operation: Enqueue()

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

Algorithm for Insertion Operation

```
Algorithm to insert an element in a queue

Step 1: IF REAR=MAX-1, then;
          Write OVERFLOW
               Goto Step 4
         [END OF IF]
Step 2: IF FRONT == -1 and REAR = -1, then
          SET FRONT = REAR = 0
         ELSE
          SET REAR = REAR + 1
         [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: Exit
```

Queue Deletion Operation: dequeue()

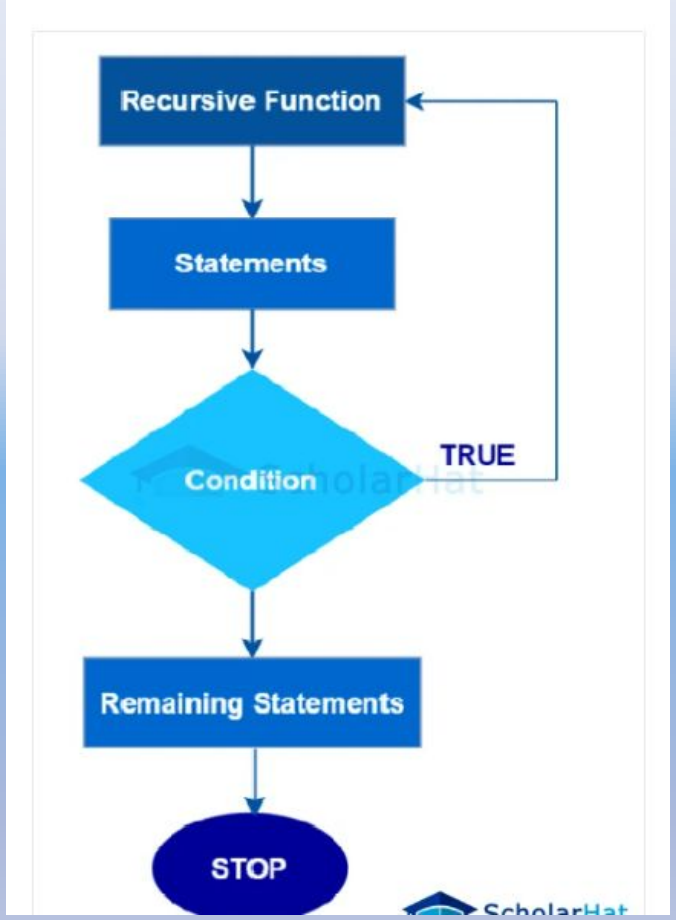
Algorithm for Deletion Operation

```
Algorithm to delete an element from a queue

Step 1: IF FRONT = -1 OR FRONT > REAR, then
          Write UNDERFLOW
                Goto Step 3
          [END OF IF]

Step 2:SET VAL = QUEUE[FRONT]
       SET FRONT = FRONT + 1

Step 3: Exit
```

1. START
2. Check if the queue is empty.
3. If the queue is empty, produce underflow error and exit.
4. If the queue is not empty, access the data where front is pointing.
5. Increment front pointer to point to the next available data element.
6. Return success.
7. END

Recursion

- Recursion is the process in which a function calls itself again and again.
- It entails decomposing a challenging issue into more manageable issues and then solving each one again.
- There must be a terminating condition to stop such recursive calls.
- Recursion may also be called the alternative to iteration.
- Recursion provides us with an elegant way to solve complex problems, by breaking them down into smaller problems and with fewer lines of code than iteration.



Properties of Recursion

- It solves a problem by breaking it down into smaller sub-problems, each of which can be solved in the same way.
- A recursive function must have a base case or stopping criteria to avoid infinite recursion.
- Recursion involves calling the same function within itself, which leads to a call stack.
- Recursive functions may be less efficient than iterative solutions in terms of memory and performance.

Applications of Recursion

- Recursive solutions are best suited to some problems like:
  - Tree Traversals: InOrder, PreOrder, PostOrder
  - Graph Traversals: DFS [Depth First Search] and BFS [Breadth First Search]
  - Tower of Hanoi
  - Backtracking Algorithms
  - Divide and Conquer Algorithms
  - Dynamic Programming Problems
  - Merge Sort, Quick Sort
  - Binary Search
  - Fibonacci Series, Factorial, etc.
- Recursion is used in the design of compilers to parse and analyze programming languages.
- Many computer graphics algorithms, such as fractals and the Mandelbrot set, use recursion to generate complex patterns.

Recursive Function

- A recursive function is a function that calls itself one or more times within its body.
- A recursive function solves a particular problem by calling a copy of itself and solving smaller subproblems of the original problems.
- Many more recursive calls can be generated as and when required.
- Syntax to Declare a Recursive Function

```
recursionfunction()
{
recursionfunction(); //calling self function
}
```

Format of Recursive Function

- A recursive function consists of two things:

1. A **base case**, in which the recursion can terminate and return the result immediately.
2. A **recursive case**, in which the function is supposed to call itself, to break the current problem down into smaller problems.

Similarly, the input size gets smaller and smaller with each recursive call, and we get the solution to that smaller problem. Later, we combine those results to find the solution to the entire problem.

Calculate Factorial of a Number Using Recursion

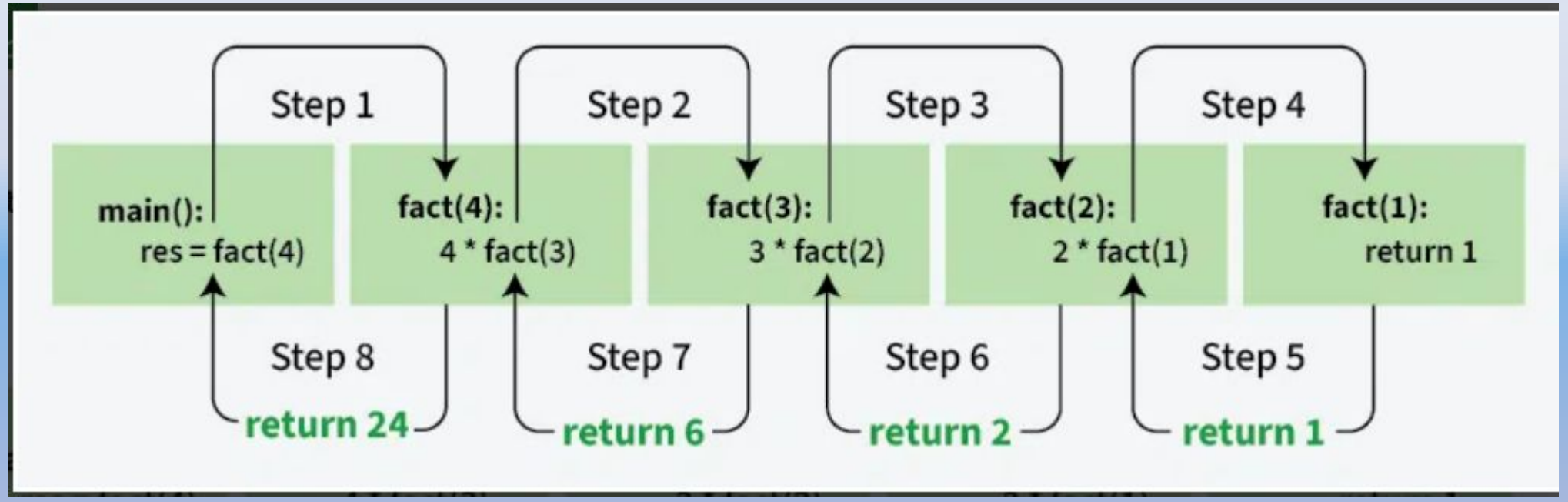
Let us first see how we can break factorial(n) into smaller problem and then define recurrence.

- ***n!*** *= n * (n – 1) * (n – 2) .... 2 * 1*
- ***(n – 1)!*** *= (n – 1) * (n – 2) ... 2 * 1*
- *From the above two equations, we can say that n!* ***= n * (n – 1)!***

Since the problem can be broken down into The idea is to define a recursive function, say ***factorial(n)*** *to calculate the factorial of number* ***n****. According to the value of n, we can have two cases:*

if *n* ***= 0 or n = 1 :***
factorial(n) = 1
Else *:*
*factorial(n) = n * factorial(n – 1).*

Illustration



Factorial algorithm

FACTORIAL(FATC,N)

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

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

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

Fibonacci series

In case of Fibonacci series, next number is the sum of previous two numbers.

F(2) = F(0) + F(1)

for example 0, 1, 1, 2, 3, 5, 8, 13, 21 etc.

The first two numbers of Fibonacci series are 0 and 1.

We can use recursion to solve this problem because any Fibonacci number **n** depends on previous two Fibonacci numbers. Therefore, this approach repeatedly breaks down the problem until it reaches the base cases.

Recurrence relation:

- ***Base case:*** *F(n) = n, when n = 0 or n = 1*
- ***Recursive case:*** *F(n) = F(n-1) + F(n-2) for n>1*

Fibonacci series with recursion

FIBONACCI (FIB, N)

This procedure calculates FN 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(FIBA, N-1)
4. Set FIB := FIBA+ FIBB
5. Return

Tower of Hanoi

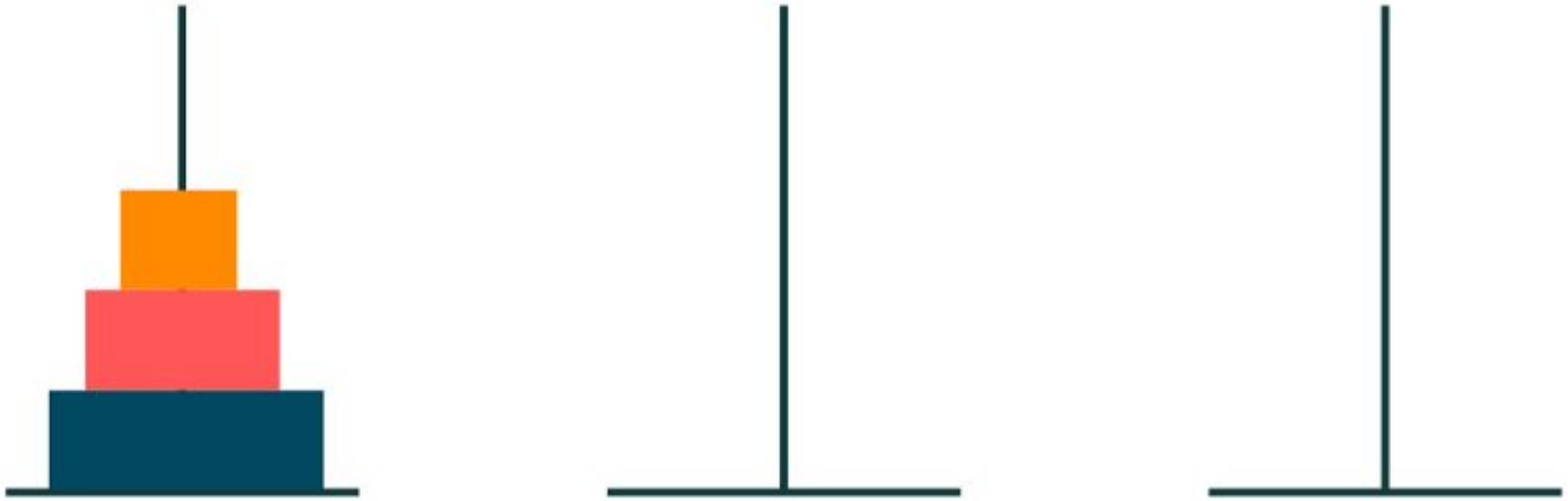
The Tower of Hanoi puzzle consists of three pegs (often represented by rods) and a set of disks of different sizes. The disks can be stacked onto the pegs, with the largest disk at the bottom and the smallest disk at the top. The objective is to move the entire stack from one peg, known as the source peg, to another peg, known as the target peg, using the third peg, called the auxiliary peg, as temporary storage.

The puzzle follows three simple rules:

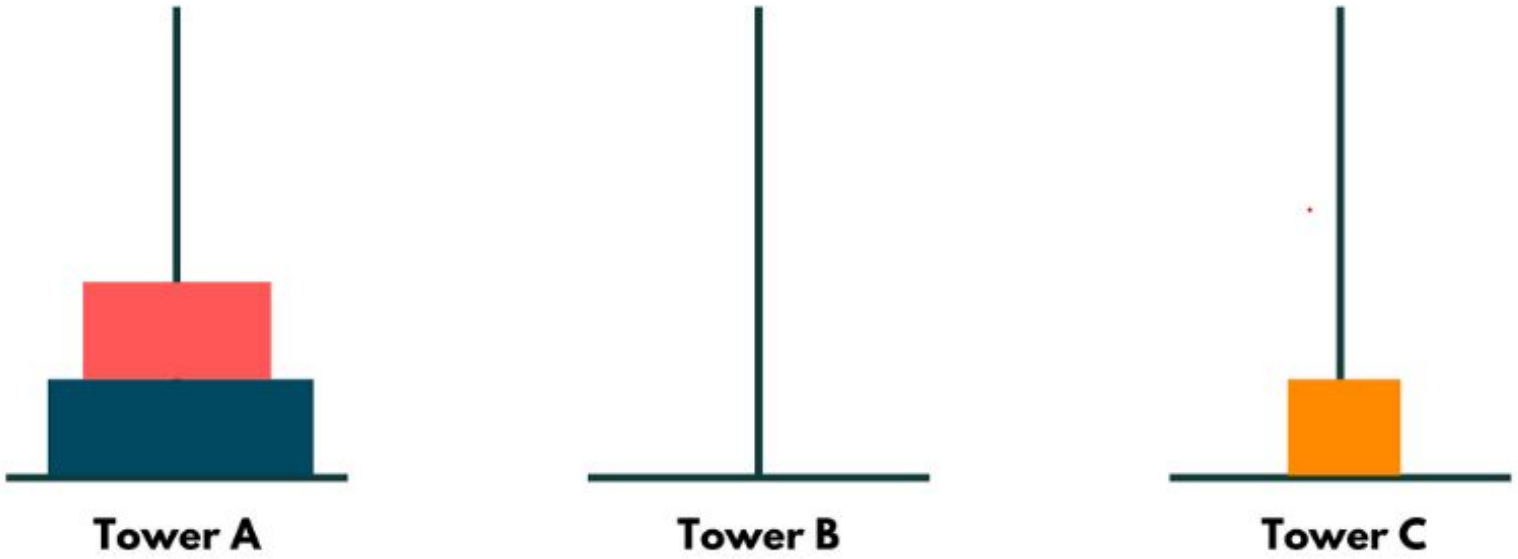
- Only one disk can be moved at a time: You can only move the topmost disk from any stack of disks on a peg.
- Larger disks cannot be placed on top of smaller disks: At no point during the puzzle can you place a larger disk on top of a smaller one
- Use the auxiliary peg for temporary storage: You can utilize the auxiliary peg to temporarily store disks while moving them from the source peg to the target peg.

Let's consider an illustrative example of the Tower of Hanoi problem with 3 disks.

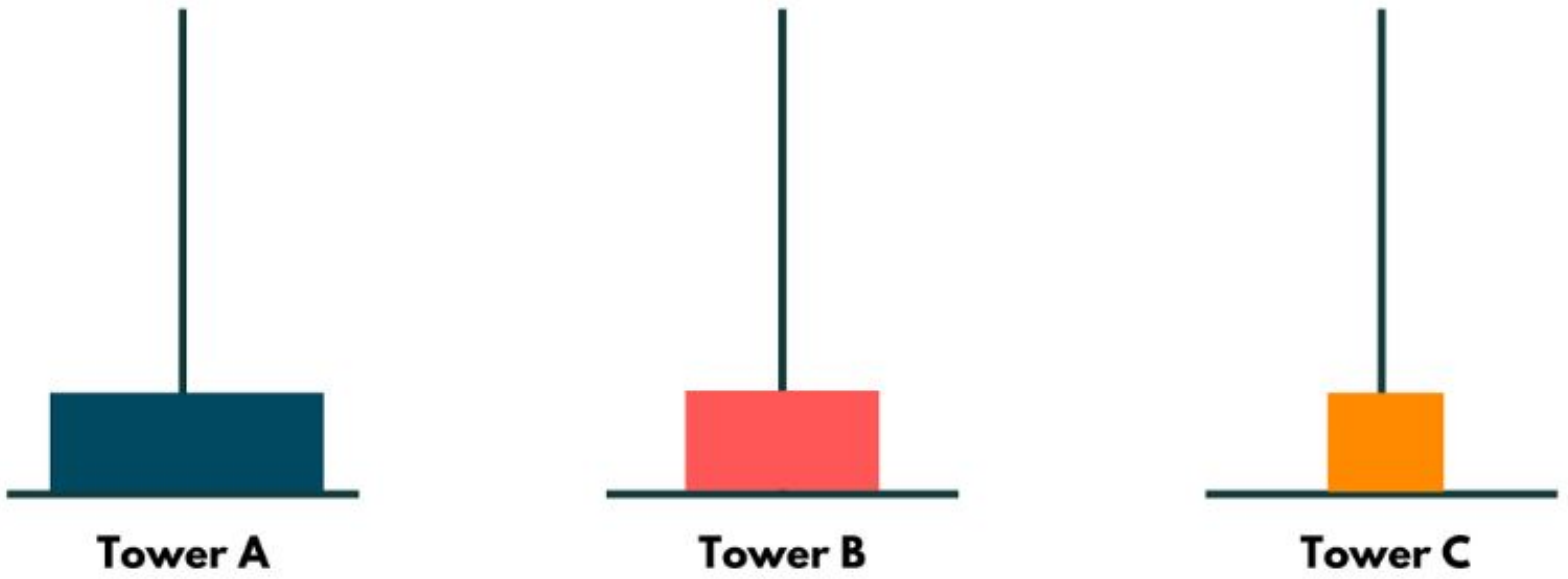
We have three pegs: A, B, and C. The initial configuration of the disks is as follows:



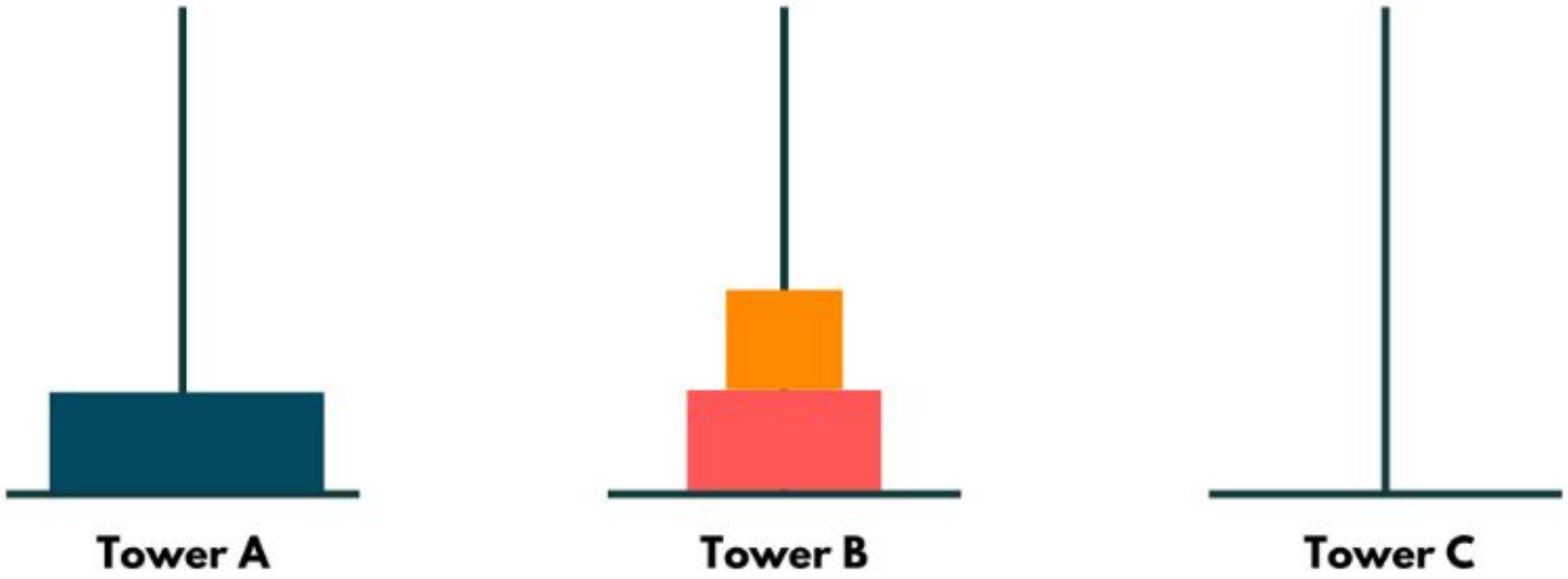
The smallest block should be moved to Tower C first. In this stage, the number of blocks is reduced from N = 3 to N = 2, illustrating how the issue has been divided. We may now imagine that our secondary issue is traveling from Tower A to Tower C over a distance of two blocks.



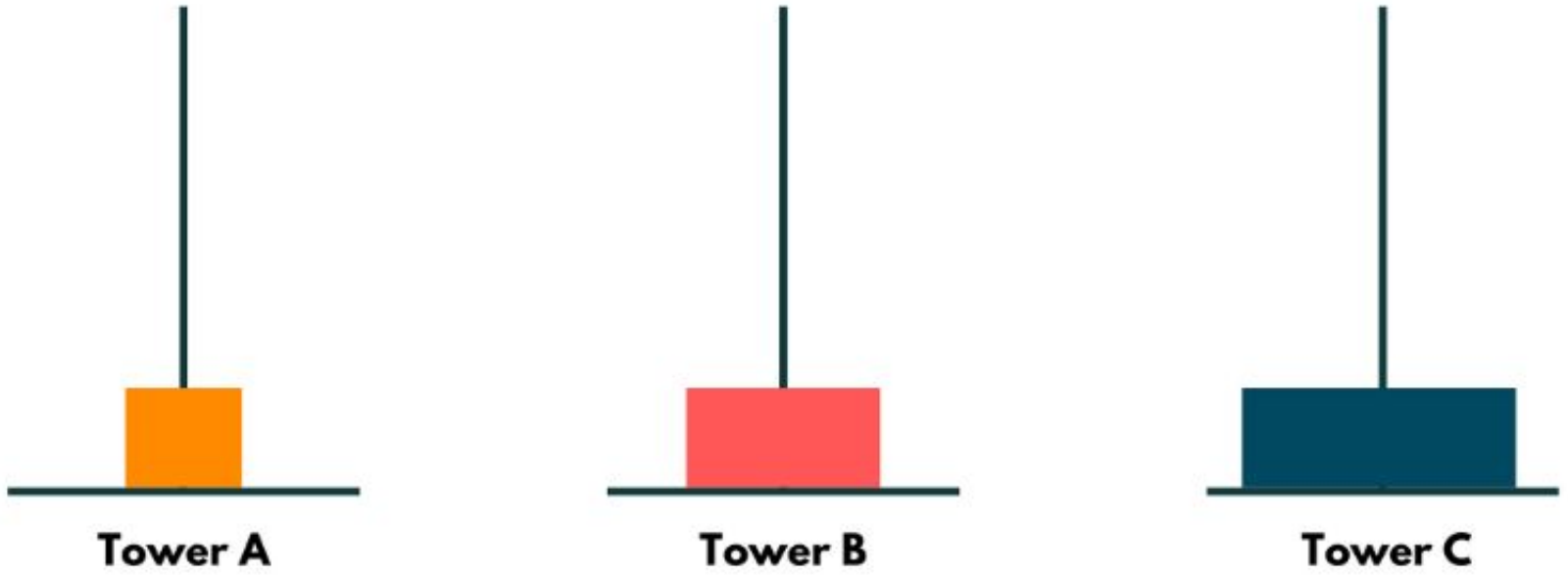
Then, as seen in the diagram below, transfer the center block to Tower B. At the source point, we further separated the sub-problem from N=2 to N=1 in a manner similar to the previous phase.



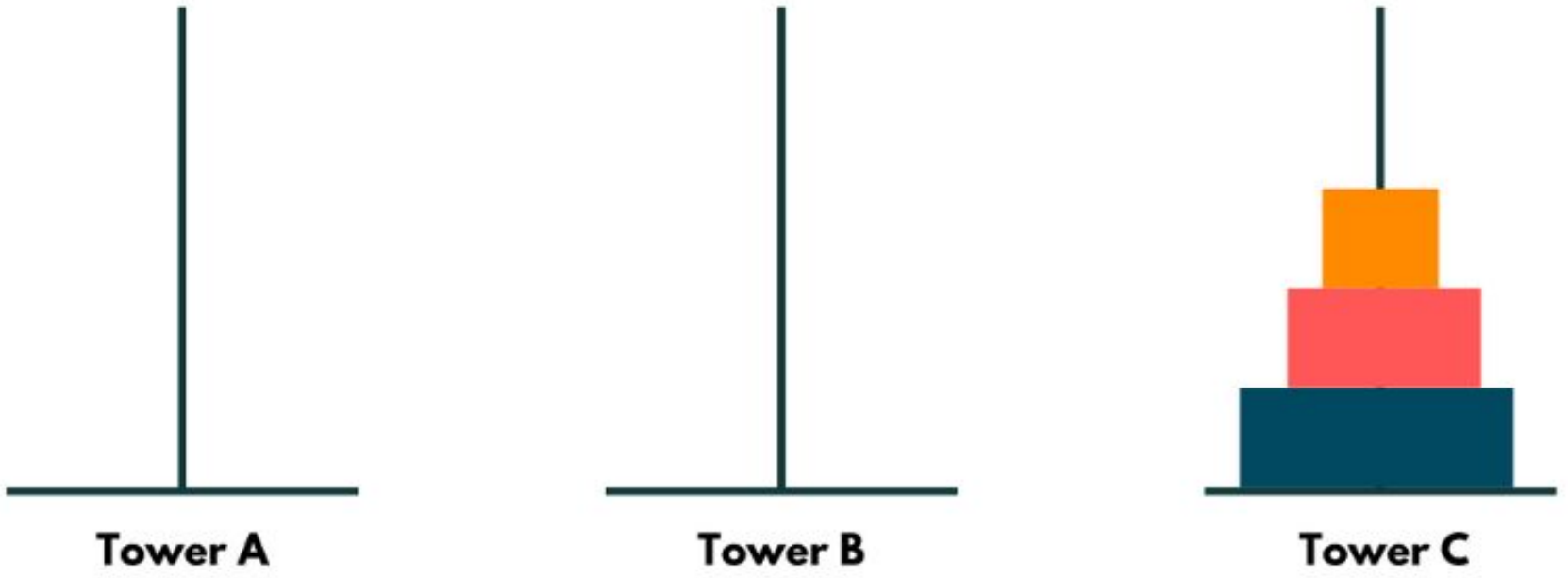
By moving the smallest block in Tower B over the middle block, we may begin to solve the subproblems in this phase, as shown below:



Later, transfer the largest block from Tower A to Tower C and the smallest block from Tower B to the vacant Tower A.



Then, as illustrated in the following diagram, we combine the solution to arrive at the ultimate outcome by moving the middle block over the largest block from Tower B to Tower C and transferring the smallest block from Tower A to Tower C.



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 □ END
    b) Return
2. [Move N-1 disks from peg BEG to peg AUX.]
   Call TOWER(N-1, BEG, END, AUX)
3. Write: BEG□END
4. [Move N-1 disks from peg AUX to peg END.]
    Call TOWER(N-1, AUX, BEG, END)
5. Return

THANK YOU

Any Question??