**MODULE – 2**

**Syllabus: -**

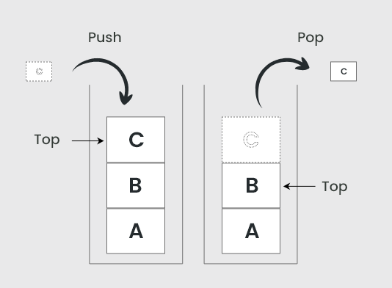
**Stack and Queue**:Stack and its implementations (using array, using linked list), applications. Queue, circular queue, dequeue. Implementation of queue- both linear and circular (using array, using linked list), applications. Implementation of dequeue with input and output restriction.

**Recursion**:Principles of recursion – use of stack, differences between recursion and iteration, tail recursion. Applications -

The Tower of Hanoi, ~~Eight Queens Puzzle~~ (Concept of Backtracking).

**STACK**

Stack is a linear data structure that follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). LIFO implies that the element that is inserted last, comes out first and FILO implies that the element that is inserted first, comes out last.



Operations in Stack -

1. **push()** - to insert an element into the stack
2. **pop() -** to remove an element from the stack
3. **top()** - Returns the top element of the stack.
4. **isempty() -**returns true if stack is empty else false.
5. **size()** - returns the size of stack.

**Algorithm for push**

Step 1: BEGIN

Step 2: IF stack is full

Step 3: RETURN “Stack is Full”

Step 4: ENDIF

Step 5: ELSE

Step 6: INCREMENT top

Step 7: stack[top] = value

Step 8: END ELSE

Step 9: END

**Algorithm for pop**

Step 1: BEGIN

Step 2: IF stack is empty

Step 3: RETURN “Stack is empty”

Step 4: ENDIF

Step 5: ELSE

Step 6: value = stack[top]

Step 7: DECREMENT top

Step 8: RETURN value

Step 9: END ELSE

Step 10:END

**Algorithm for Top**

Step 1: BEGIN

Step 2: RETURN stack[top]

Step 3: END

**Algorithm for isEmpty**

Step 1: BEGIN

Step 2: IF top < 1

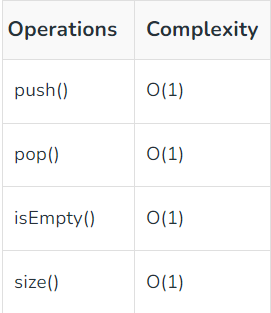
Step 3: RETURN TRUE

Step 4: ELSE

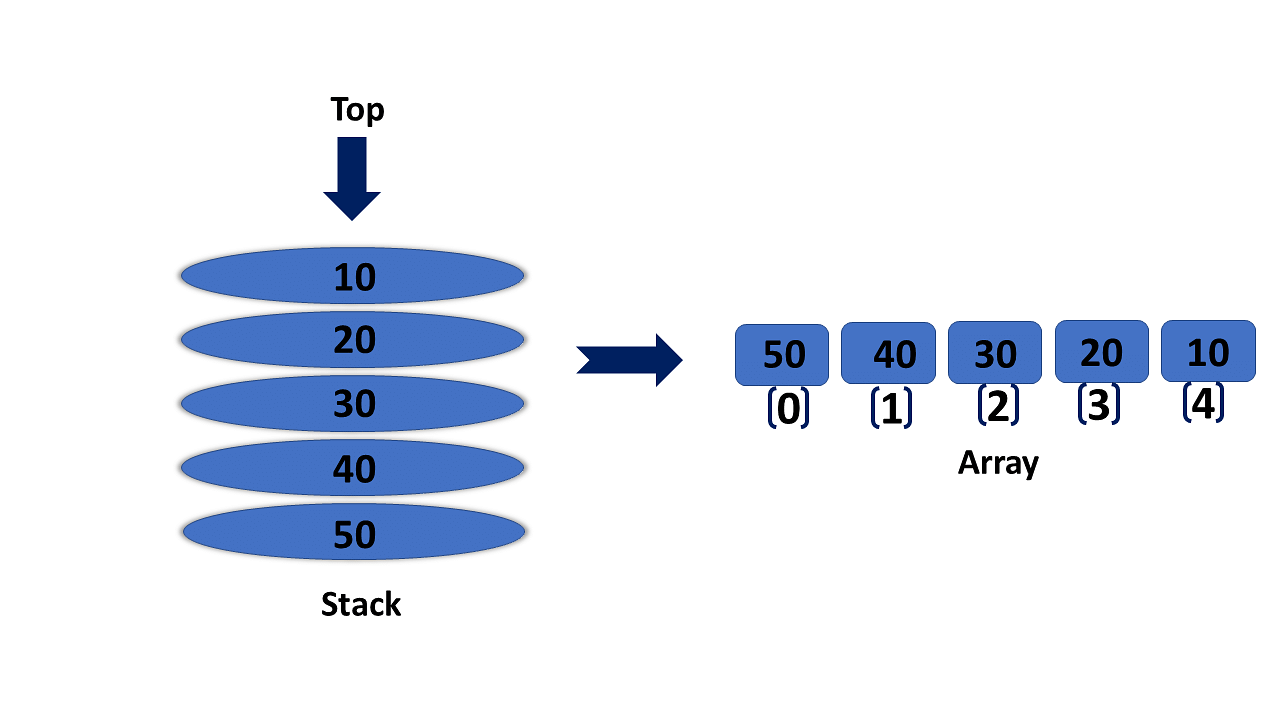
Step 5: RETURN FALSE

Step 6: END

Time Complexity: -



Stack implementation using Array



**CODE**

# include <stdio.h>

int MAX = 8;

int stack[8];

int top = -1;

int isempty()

{

if (top==-1)

return 1;

else;

return 0;

}

int isfull()

{

if (top==MAX)

return 1;

else

return 0;

}

int push(int a)

{

if(!isfull())

{

top = top + 1;

stack[top] = a;

}

else

printf("Stack is full can't insert");

}

int pop()

{

int data;

if (!isempty())

{

data = stack[top];

top = top - 1;

return data;

}

else

printf("Stack is empty");

}

int Top()

{

printf("Top element is: %d\n",stack[top]);

}

int main()

{

push(1);

push(2);

push(3);

push(4);

push(5);

push(6);

push(7);

push(8);

Top();

pop();

printf("Elements in Stack after PUSH and POP operation: \n") ;

while(!isempty())

{

int data = pop();

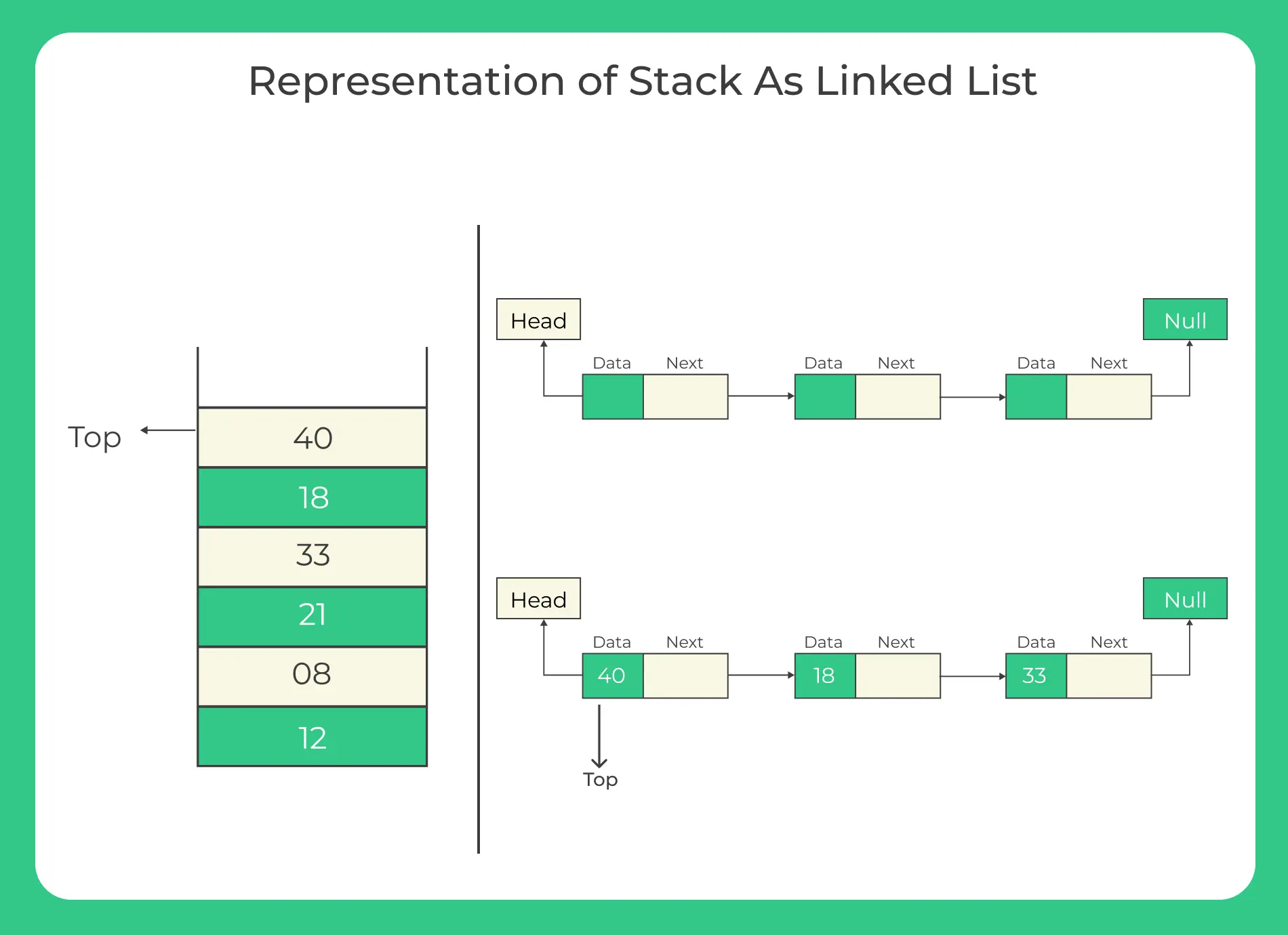
if (data!=0)

printf("%d\n", data);

}

}

Stack implementation using Linked List



**CODE**

# include <stdio.h>

# include <stdlib.h>

struct node

{

int data;

struct node \*next;

};

node \*top = NULL;

int push(int data)

{

struct node \*newnode;

newnode = (struct node\*)malloc(sizeof(struct node));

newnode -> data = data;

newnode -> next = top;

top = newnode;

printf("(%d) --> Pushed\n", newnode -> data);

}

int pop()

{

if (top == NULL)

{

printf("Empty Stack!!");

return -1;

}

else

{

struct node \*temp = top;

int data = top -> data;

top = top -> next;

free(temp);

printf("(%d) --> Popped\n", data);

}

}

int Top()

{

printf("\nTop Element --> (%d)\n", top -> data);

}

int main()

{

push(1);

push(2);

push(3);

push(4);

push(5);

push(6);

pop();

Top();

printf("\nSTACK: \n");

while(top -> data != NULL)

{

printf("| %d |\n", top -> data);

top = top -> next;

}

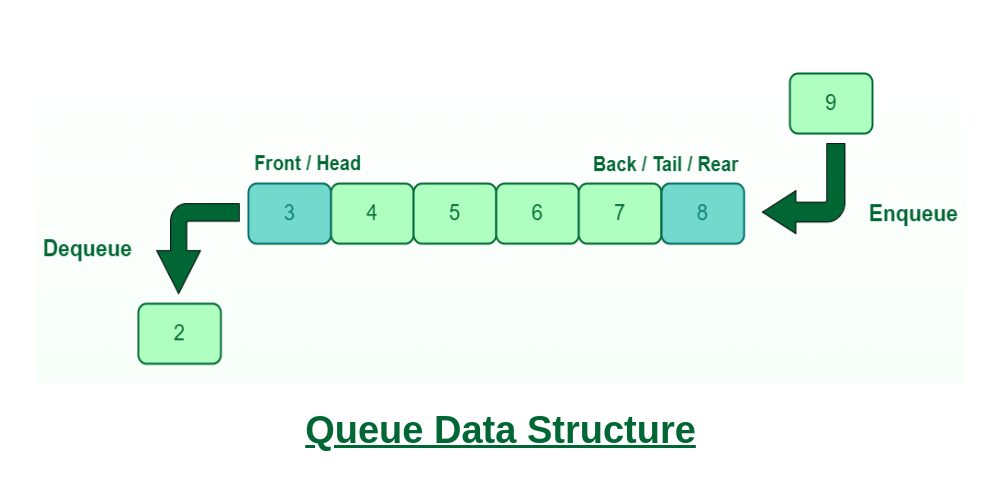
}

Applications of Stack

1. Evaluation of Arithmetic Expressions
2. Backtracking
3. Delimeter Checking
4. Reverse a data
5. Processing of Function Calls

**QUEUE**

A **Queue** is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.



Operation in Queue -

1. **Enqueue() –** Adds (or stores) an element to the end of the queue..
2. **Dequeue() –** Removal of elements from the queue.
3. **Peek() or front() –** Acquires the data element available at the front node of the queue without deleting it.
4. **rear() –** This operation returns the element at the rear end without removing it.
5. **isFull() –** Validates if the queue is full.
6. **isNull() –** Checks if the queue is empty.

**Algorithm for Enqueue()**

**Step 1:** Check if the queue is full.

**Step 2:**If the queue is full, return overflow error and exit.

**Step 3:** If the queue is not full, increment the rear pointer to point to the next empty space.

**Step 4:** Add the data element to the queue location, where the rear is pointing.

**Step 5:** return success.

**Algorithm for Dequeue()**

**Step 1:** Check if the queue is empty.

**Step 2:** If the queue is empty, return the underflow error and exit.

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

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

**Step 5:** The Return success.

**Algorithm for Peek()**

**Step 1:** Check if the queue is empty or not.

**Step 2:** If the queue is empty, return the underflow error and exit.

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

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

**Step 5:** The Return success.

**Algorithm for Rear()**

**Step 1:** Check if the queue is empty or not.

**Step 2:** If the queue is empty, return the underflow error and exit.

**Step 3:** If the queue is not empty, access the data where the rear is pointing.

**Step 4:** Decrement the rear pointer to point to the previous available data element.

**Step 5:** The Return success.

Time Complexity for every operations in Queue is O(n).

**CODE**

1. **Queue - Array Implementation**

# include <stdio.h>

# include <stdlib.h>

int MAX = 8;

int Queue[8];

int front = -1, rear = -1;

int isfull()

{

if (rear==MAX)

return 1;

else

return 0;

}

int isempty()

{

if (rear==-1)

return 1;

else

return 0;

}

int enqueue(int a)

{

if(!isfull())

{

front = 0;

rear = rear + 1;

Queue[rear] = a;

}

else

printf("Queue Full");

}

int dequeue()

{

if (!isempty())

{

front = front + 1;

}

else

printf("Queue Empty");

}

int peek()

{

if (!isempty())

{

printf("FRONT element is: %d\n", Queue[front]);

}

else

printf("Empty Queue");

}

int main()

{

int data;

enqueue(9);

enqueue(11);

enqueue(14);

enqueue(3);

enqueue(5);

enqueue(7);

dequeue();

peek();

printf("Queue elements after ENQUEUE & DEQUEUE OPERTIONS: \n");

while(front!=rear+1)

{

data = Queue[front];

front ++;

if (data!=0)

printf("%d ", data);

}

}

1. **Queue - Linked List Implementation**

# include <stdio.h>

# include <stdlib.h>

struct node

{

int data;

struct node \*next;

};

struct node \*front = NULL;

struct node \*rear = NULL;

int isempty()

{

if (front == NULL)

return 1;

else

return 0;

}

void enqueue(int data)

{

struct node \*newnode;

newnode = (struct node\*)malloc(sizeof(struct node));

newnode -> data = data;

if (isempty())

{

front = newnode;

front -> next = rear;

rear = newnode;

}

else

{

rear -> next = newnode;

rear = newnode;

}

printf("(%d) --> Enqueued\n", newnode -> data);

}

void dequeue()

{

if (isempty())

{

printf("Queue is empty.");

}

else

{

printf("\n(%d) --> Dequeued", front -> data);

front = front -> next;

}

}

int main()

{

enqueue(1);

enqueue(2);

enqueue(3);

printf("\nQueue --> ");

while(front != rear)

{

printf("(%d) ", front -> data);

front = front -> next;

}

printf("(%d)", front -> data);

}

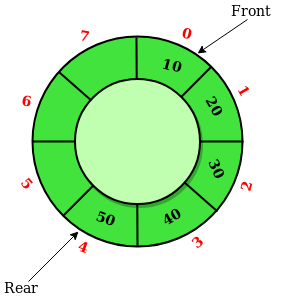
Applications of Queue: -

1. Semaphores
2. FCFS Scheduling
3. CPU Scheduling
4. Memory Management

**CIRCULAR QUEUE**

A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.

The operations are performed based on FIFO (First In First Out) principle. It is also called **‘Ring Buffer’**. 



**CODE**

1. **Circular Queue - Array Implementation**

#include <stdio.h>

#define SIZE 5

int items[SIZE];

int front = -1, rear = -1;

int isFull()

{

if ((front == rear + 1) || (front == 0 && rear == SIZE - 1))

return 1;

else

return 0;

}

int isEmpty()

{

if (front == -1)

return 1;

else

return 0;

}

void enQueue(int element)

{

if (isFull())

printf("\n Queue is full!! \n");

else

{

if (front == -1) front = 0;

rear = (rear + 1) % SIZE;

items[rear] = element;

printf("\n(%d) --> Enqueued", element);

}

}

int deQueue()

{

int element;

if (isEmpty())

{

printf("\n Queue is empty !! \n");

return (-1);

}

else

{

element = items[front];

if (front == rear)

{

front = -1;

rear = -1;

}

else

{

front = (front + 1) % SIZE;

}

printf("\n(%d) --> Dequeued\n", element);

return (element);

}

}

void display()

{

int i;

if (isEmpty())

printf(" \nEmpty Queue\n");

else

{

printf("\nCircular Queue -> ");

for (i = front; i != rear; i = (i + 1) % SIZE) {

printf("(%d)", items[i]);

}

printf("(%d)", items[i]);

}

}

int main()

{

enQueue(1);

enQueue(2);

enQueue(3);

enQueue(4);

enQueue(5);

deQueue();

display();

return 0;

}

1. **Circular Queue - Linked List implementation**

# include <stdio.h>

# include <stdlib.h>

struct node

{

int data;

struct node \*next;

};

struct node \*front = NULL;

struct node \*rear = NULL;

int enqueue(int data)

{

struct node \*newnode;

newnode = (struct node\*)malloc(sizeof(struct node));

newnode -> data = data;

newnode -> next = NULL;

if ((front == NULL) && (rear == NULL))

{

front = newnode;

rear = newnode;

rear -> next = front;

}

else

{

rear -> next = newnode;

rear = newnode;

newnode -> next = front;

}

printf("(%d) --> Enqueued\n", newnode -> data);

}

int dequeue()

{

if (rear == NULL)

{

printf("Queue Empty.");

return -1;

}

else

{

printf("(%d) --> Dequeued\n", front -> data);

front = front -> next;

}

}

int peek()

{

printf ("\nFront Element --> %d\n", front -> data);

}

int main()

{

enqueue(1);

enqueue(2);

enqueue(3);

dequeue();

peek();

printf("\nCircular Queue --> ");

while (front != rear)

{

printf("%d ", front -> data);

front = front -> next;

}

printf("%d", front -> data);

}

Applications of Circular Queue: -

1. Process Scheduling
2. Memory Management
3. Computer Controlled Traffic System

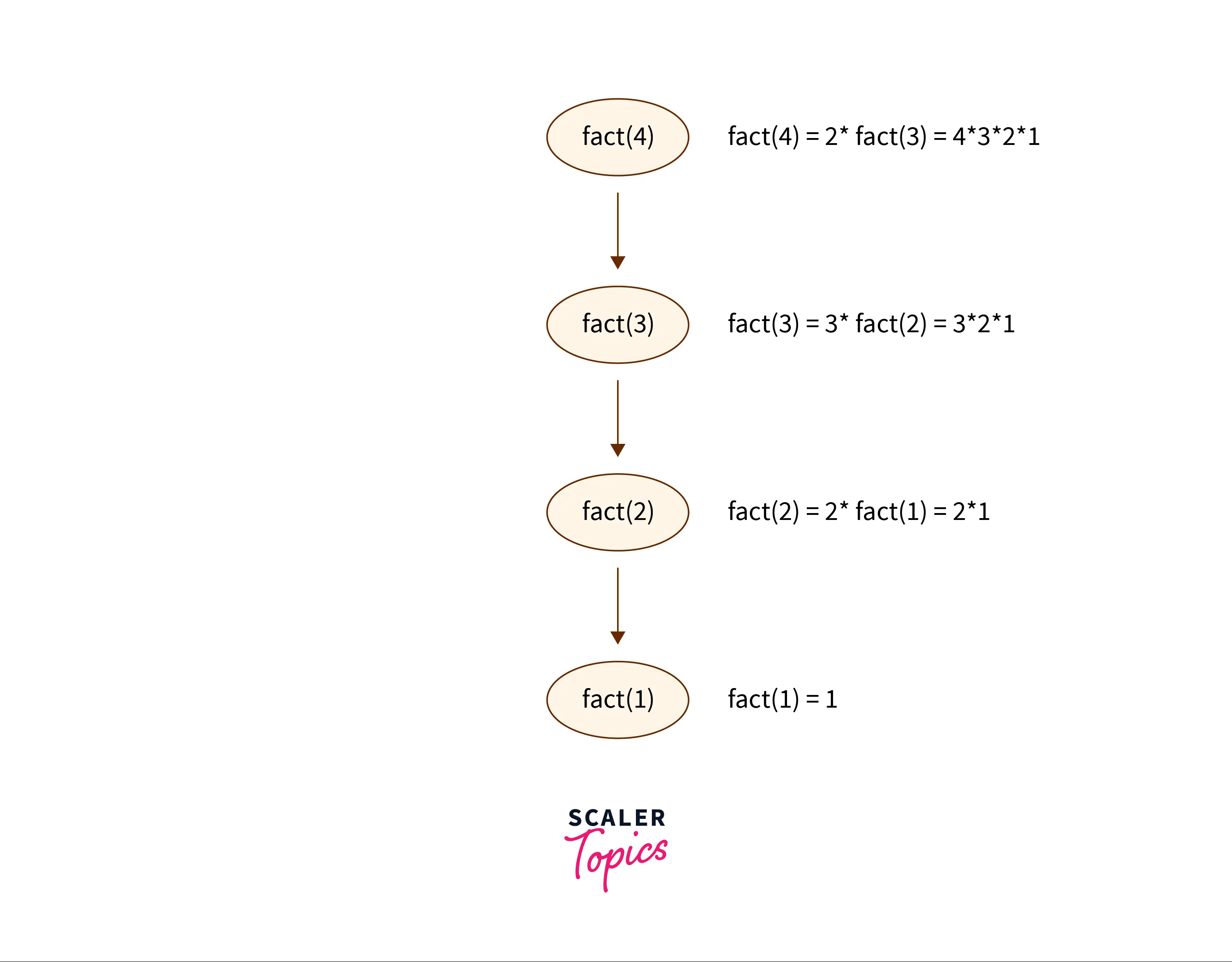
**Principles of Recursion**

1. A recursive algorithm must call itself recursively.
2. A recursive algorithm must have a base case.
3. A recursive algorithm must change its state and move towards the best case.

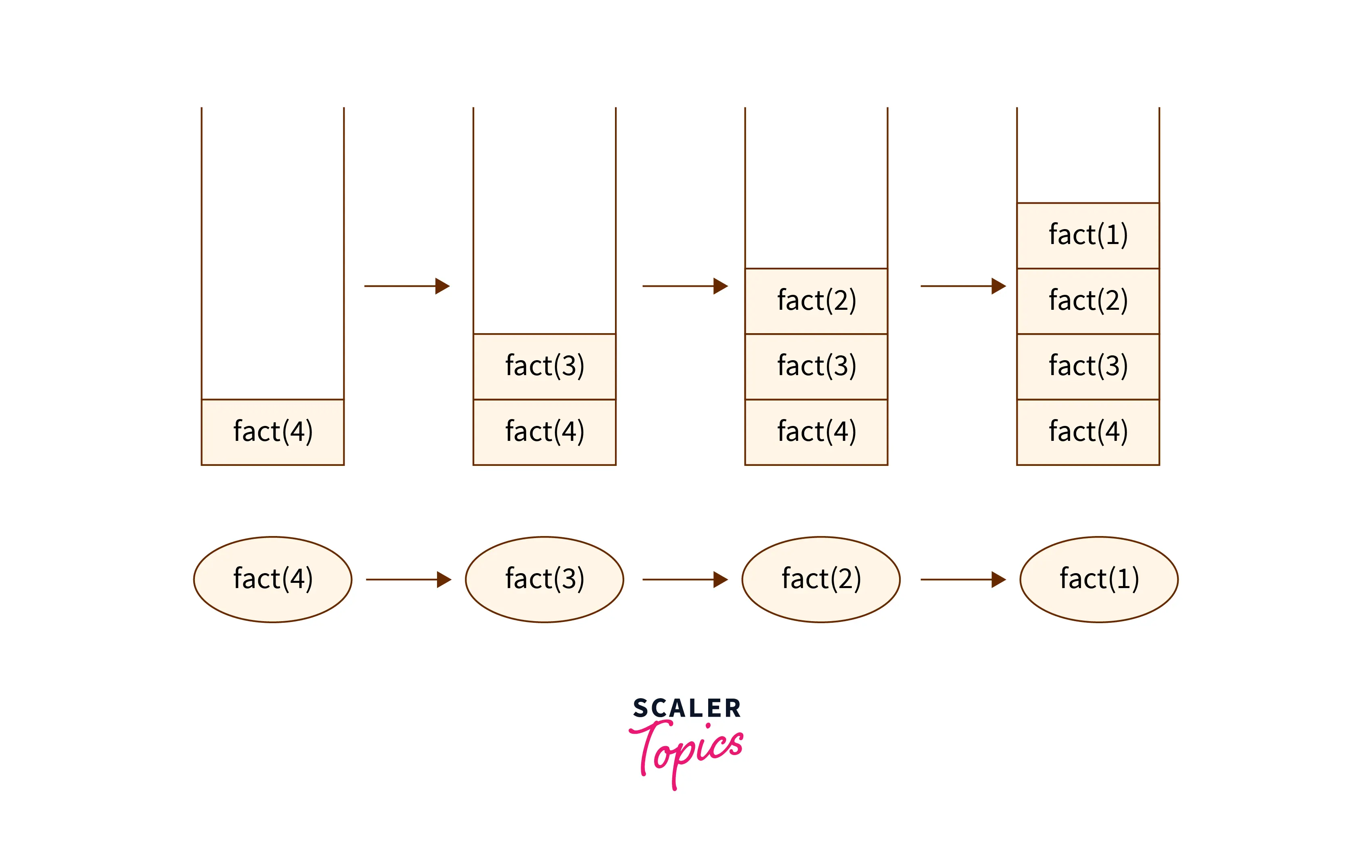
**Use of Stack in Recursion**

Lets take an example: - We need to find the factorial of 4.

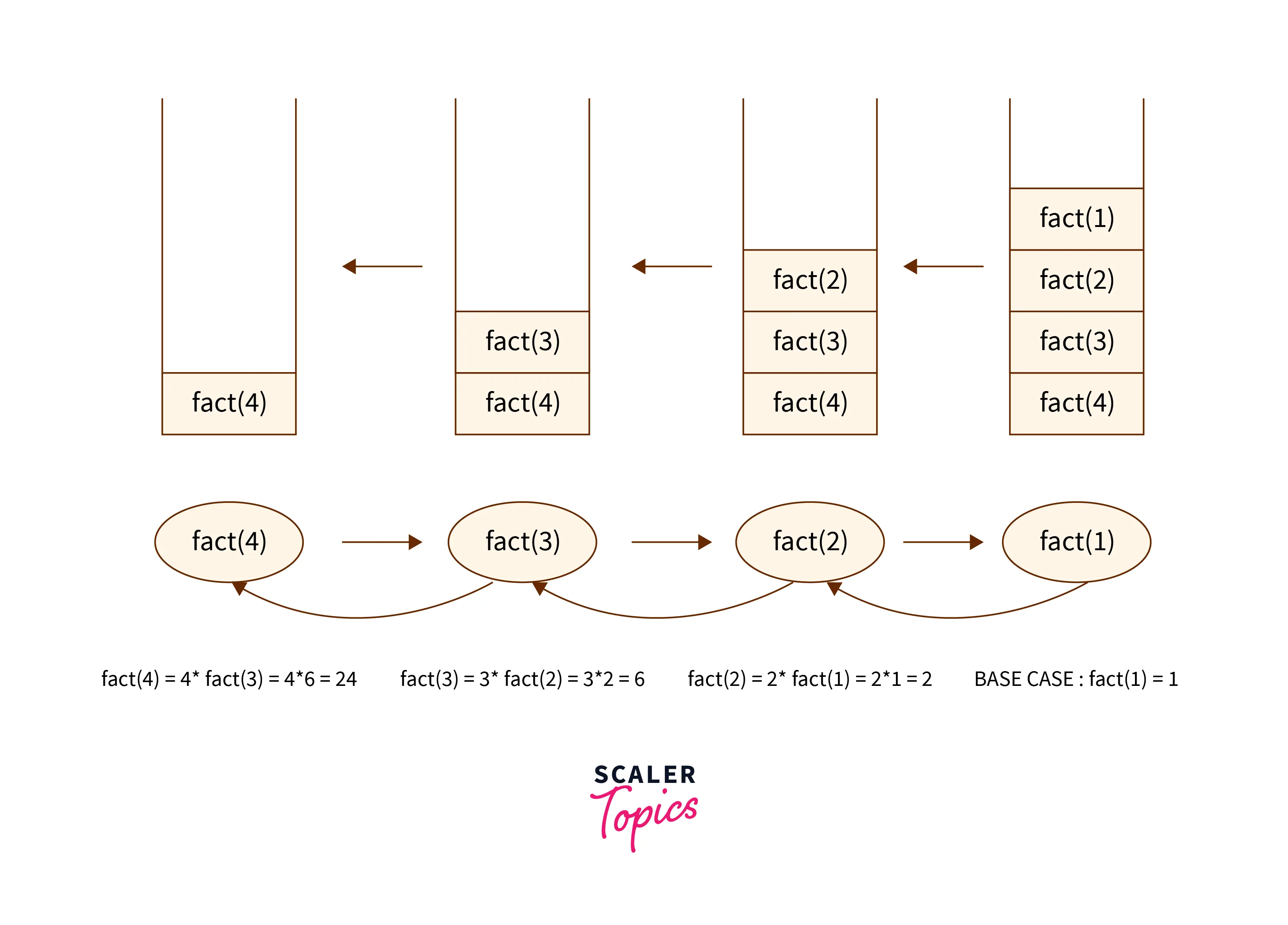




Recursion backtracks to previous input once it finds the base case and the temporary function calls which are pending are stored in the ****stack data structure**** in the memory as follows.



with each function call, the stack keeps filling until the base case arrives which is ****fact(1) = 1**** in this case. After that, each function call is evaluated in the last in first out order.



## Difference between Iteration and Recursion

| **Property** | **Recursion** | **Iteration** |
| --- | --- | --- |
| **Definition** | Function calls itself. | A set of instructions repeatedly executed. |
| **Application** | For functions. | For loops. |
| **Termination** | Through base case, where there will be no function call. | When the termination condition for the iterator ceases to be satisfied. |
| **Usage** | Used when code size needs to be small, and time complexity is not an issue. | Used when time complexity needs to be balanced against an expanded code size. |
| **Code Size** | Smaller code size | Larger Code Size. |
| **Time Complexity** | Very high(generally exponential) time complexity. | Relatively lower time complexity(generally polynomial-logarithmic). |
| **Space Complexity** | The space complexity is higher than iterations. | Space complexity is lower. |
| **Stack** | Here the stack is used to store local variables when the function is called. | Stack is not used. |
| **Speed** | It is comparatively slower | It is usually faster |
| **Memory** | Recursion uses more memory as compared to iteration. | Iteration uses less memory as compared to recursion. |

**Tower of Hanoi (Application of recursion)**

Rules: -

* Only one disk can be moved among the towers at any given time.
* Only the "top" disk can be removed.
* No large disk can sit over a small disk.

**Algorithm**

Step 1: START

Step 2: Procedure Hanoi(disk, source, dest, aux)

Step 3: IF disk == 1, THEN

Step 4: move disk from source to dest

Step 5: ELSE

Step 6: Hanoi(disk - 1, source, aux, dest) // Step 1

Step 7: move disk from source to dest // Step 2

Step 8: Hanoi(disk - 1, aux, dest, source) // Step 3

Step 9: END IF

Step 10: END Procedure

Step 11: STOP

**Code**

#include <stdio.h>

void towerOfHanoi(int n, char from, char to, char aux) {

if (n == 1) {

printf("Move disk from %c to %c\n", from, to);

return;

}

towerOfHanoi(n - 1, from, aux, to);

printf("Move disk from %c to %c\n", from, to);

towerOfHanoi(n - 1, aux, to, from);

}

int main() {

int n;

printf("Enter the number of disks: ");

scanf("%d", &n);

towerOfHanoi(n, 'A', 'C', 'B');

return 0;

}