Recursion

- Return Type − A function may return a value. The return_type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return_type is the keyword void.
- Function Name − This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- Parameters − A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- Function Body − The function body contains a collection of statements that define what the function does.

```
return_type function_name( parameter list ) {
   body of the function
}
```

Function

```c
#include <stdio.h>

// An example function that takes two parameters 'x' and 'y'
// as input and returns max of two input numbers
int max(int x, int y)
{
    if (x > y)
      return x;
    else
      return y;
}

// main function that doesn't receive any parameter and
// returns integer.
int main(void)
{
    int a = 10, b = 20;

    // Calling above function to find max of 'a' and 'b'
    int m = max(a, b);

    printf("m is %d", m);
    return 0;
}
```

Intro to Recursion

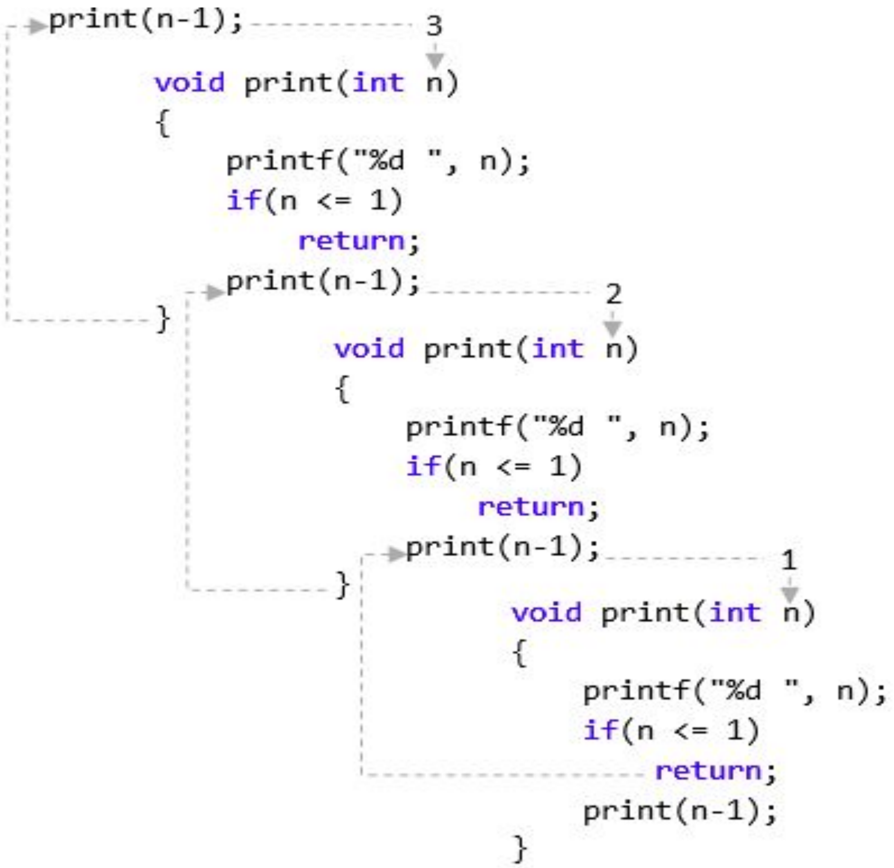
Recursion is expressing an entity in terms of itself. Similarly, a recursive function is the function that calls itself. While using recursion mentioning the exit condition or base condition is the key to avoid infinite loop continuation.

```
void recursion() {
   recursion(); /* function calls itself */
}

int main() {
   recursion();
}
```

```c
#include <stdio.h>

void print(int n);

int main()
{
    print(5);

    return 0;
}

void print(int n)
{
    /* Print the current value of n */
    printf("%d ", n);

    /* Base condition to terminate recursion */
    if(n <= 1)
    {
        /* Return and make no more recursive call */
        return;
    }

    /* Call print() function recursively with n-1 */
    print(n - 1);
}
```

1. The program execution starts from `main()` function. It calls `print()` function with `n=5`.
2. Inside the `print()` function the first statement prints value of `n` (i.e. 5 for first function call).
3. After printing the value of `n`, a condition is checked `if(n <= 1)`, then terminate from the function without executing below tasks.
4. If the condition `(n <= 1)` is false, then a recursive call to `print()` function is made with decreased value of `n` (i.e. 4 if `n=5`).
5. `print()` function is executed again with `n=4` and step 2 to 4 is repeated till `n=1`.

```
int main()
{
    print(5);
    return 0;
}
     void print(int n)
     {
         printf("%d ", n);
         if(n <= 1)
             return 0;
         print(n-1);
     }
          void print(int n)
          {
              printf("%d ", n);
              if(n <= 1)
                  return;
              print(n-1);
          }
```


print(n-1);
3
void print(int n)
{
printf("%d ", n);
if(n <= 1)
return;
print(n-1);
2
}
void print(int n)
{
printf("%d ", n);
if(n <= 1)
return;
print(n-1);
1
}
void print(int n)
{
printf("%d ", n);
if(n <= 1)
return;
print(n-1);
}

More examples of recursion

- Factorial Finding
- Fibonacci series counting
- Sum of the array elements
- GCD Finding

Factorial

5 ! = 5 * 4 * 3 * 2 * 1

4! = 4 * 3 * 2 * 1

.

.

N ! = N * (N-1) * (N-2) * (N-3) *........*1

```c
#include <stdio.h>

unsigned long long int factorial(unsigned int i) {

   if(i <= 1) {
      return 1;
   }
   return i * factorial(i - 1);
}

int  main() {
   int i = 12;
   printf("Factorial of %d is %d\n", i, factorial(i));
   return 0;
}
```

Fibonacci Series

$F(i) = F(i-1) + F(i-2)$

0, 1, 1 , 2 , 3, 5, 8, 13, 21…...

```c
#include <stdio.h>

int fibonacci(int i) {

   if(i == 0) {
      return 0;
   }

   if(i == 1) {
      return 1;
   }
   return fibonacci(i-1) + fibonacci(i-2);
}

int  main() {

   int i;

   for (i = 0; i < 10; i++) {
      printf("%d\t\n", fibonacci(i));
   }

   return 0;
}
```

Introduction to Queue

Queue is an important data structure that follows “first in first out” fashion. That is the item that goes in first is the item that comes out first too.

Queue Specifications

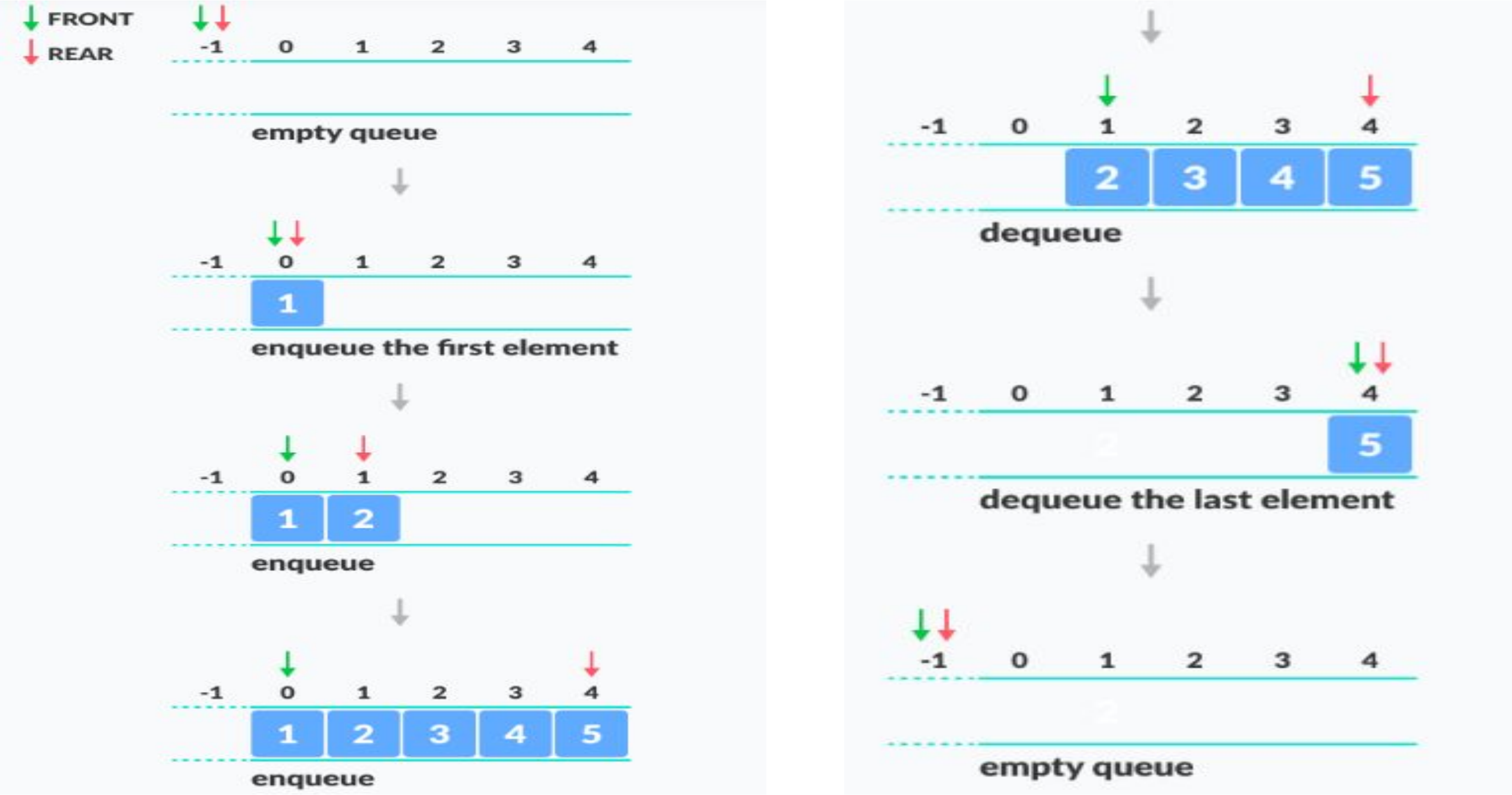
A queue is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- Enqueue: Add an element to the end of the queue
- Dequeue: Remove an element from the front of the queue
- IsEmpty: Check if the queue is empty
- IsFull: Check if the queue is full
- Peek: Get the value of the front of the queue without removing it

How Queue Works

Queue operations work as follows:

1. Two pointers called `FRONT` and `REAR` are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of `FRONT` and `REAR` to -1.
3. On enqueuing an element, we increase the value of `REAR` index and place the new element in the position pointed to by `REAR`.
4. On dequeuing an element, we return the value pointed to by `FRONT` and increase the `FRONT` index.
5. Before enqueuing, we check if the queue is already full.
6. Before dequeuing, we check if the queue is already empty.
7. When enqueuing the first element, we set the value of `FRONT` to 0.
8. When dequeuing the last element, we reset the values of `FRONT` and `REAR` to -1.


FRONT
REAR
-1 0 1 2 3 4
empty queue
1
enqueue the first element
1 2
enqueue
1 2 3 4 5
enqueue
2 3 4 5
dequeue
5
dequeue the last element
empty queue

```
int main() {
  //deQueue is not possible on empty queue
  deQueue();

  //enQueue 5 elements
  enQueue(1);
  enQueue(2);
  enQueue(3);
  enQueue(4);
  enQueue(5);

  //6th element can't be added to queue because queue is full
  enQueue(6);

  display();

  //deQueue removes element entered first i.e. 1
  deQueue();

  //Now we have just 4 elements
  display();

  return 0;
}
```

```c
void enQueue(int value) {
  if (rear == SIZE - 1)
    printf("\nQueue is Full!!");
  else {
    if (front == -1)
      front = 0;
    rear++;
    items[rear] = value;
    printf("\nInserted -> %d", value);
  }
}

void deQueue() {
  if (front == -1)
    printf("\nQueue is Empty!!");
  else {
    printf("\nDeleted : %d", items[front]);
    front++;
    if (front > rear)
      front = rear = -1;
  }
}
```

```c
// Function to print the queue
void display() {
  if (rear == -1)
    printf("\nQueue is Empty!!!");
  else {
    int i;
    printf("\nQueue elements are:\n");
    for (i = front; i <= rear; i++)
      printf("%d  ", items[i]);
  }
  printf("\n");
}
```

Intro to Stack

Stack is a data structure that follows “first in last out” fashion.

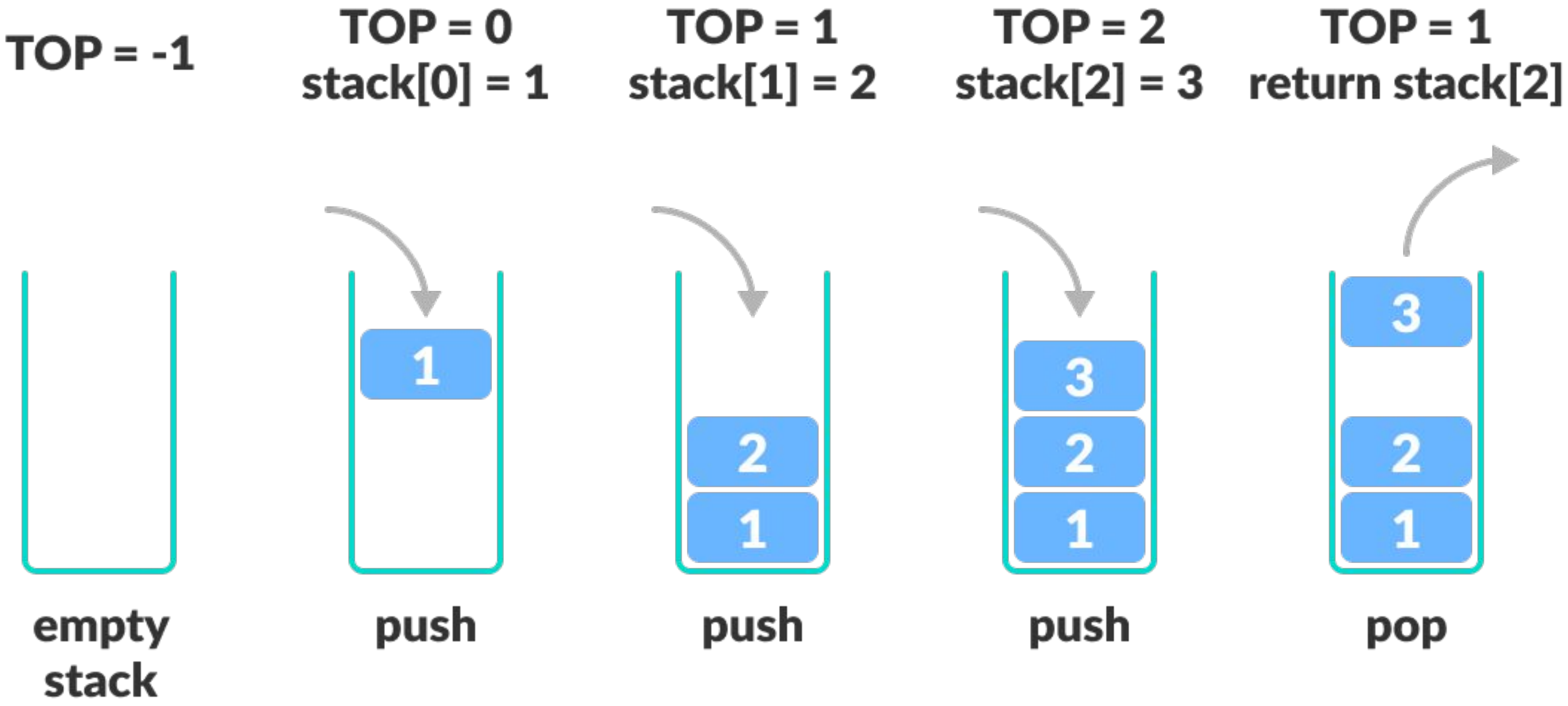
A stack is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- `Push`: Add an element to the top of a stack
- `Pop`: Remove an element from the top of a stack
- `IsEmpty`: Check if the stack is empty
- `IsFull`: Check if the stack is full
- `Peek`: Get the value of the top element without removing it

How a Stack Works

The operations work as follows:

1. A pointer called `TOP` is used to keep track of the top element in the stack.
2. When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing `TOP == -1`.
3. On pushing an element, we increase the value of `TOP` and place the new element in the position pointed to by `TOP`.
4. On popping an element, we return the element pointed to by `TOP` and reduce its value.
5. Before pushing, we check if the stack is already full
6. Before popping, we check if the stack is already empty


TOP = -1
TOP = 0
stack[0] = 1
TOP = 1
stack[1] = 2
TOP = 2
stack[2] = 3
TOP = 1
return stack[2]
1
2
1
3
2
1
3
2
1
empty
stack
push
push
push
pop

```c
int main() {
   // push items on to the stack
   push(3);
   push(5);
   push(9);
   push(1);
   push(12);
   push(15);

   printf("Element at top of the stack: %d\n" ,peek());
   printf("Elements: \n");

   // print stack data
   while(!isempty()) {
      int data = pop();
      printf("%d\n",data);
   }

   printf("Stack full: %s\n" , isfull()?"true":"false");
   printf("Stack empty: %s\n" , isempty()?"true":"false");

   return 0;
}
```

```c
#include <stdio.h>

int MAXSIZE = 8;
int stack[8];
int top = -1;

int isempty() {

   if(top == -1)
      return 1;
   else
      return 0;
}

int isfull() {

   if(top == MAXSIZE)
      return 1;
   else
      return 0;
}

int peek() {
   return stack[top];
}
```

```c
int pop() {
   int data;

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}

int push(int data) {

   if(!isfull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```