UNIT – 2

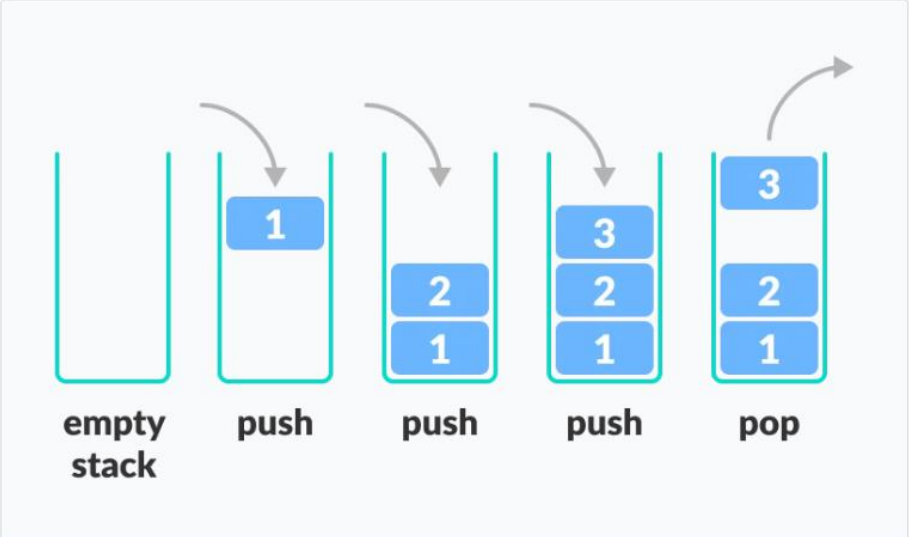
Stacks, Recursion and Queue:
Stacks: Definition, Stack Operations, Array Representation of Stacks, Stacks using Dynamic Arrays, Stack Applications: Polish notation, Infix to postfix conversion, evaluation of postfix expression.
Recursion -Factorial, GCD, Fibonacci Sequence, Tower of Hanoi,
Queues: Definition, Array Representation, Queue Operations, Circular Queues, Circular queues using Dynamic arrays, Deque, Priority Queues and its problems.

STACK

A stack is a linear data structure that follows the principle of **Last In First Out (LIFO).** This means the last element inserted inside the stack is removed first.

LIFO Principle of Stack

In programming terms, putting an item on top of the stack is called **push** and removing an item is called **pop**.



Stack Push and Pop Operations

In the above image, although item **3** was kept last, it was removed first. This is exactly how the **LIFO (Last In First Out) Principle** works.

Basic Operations of Stack

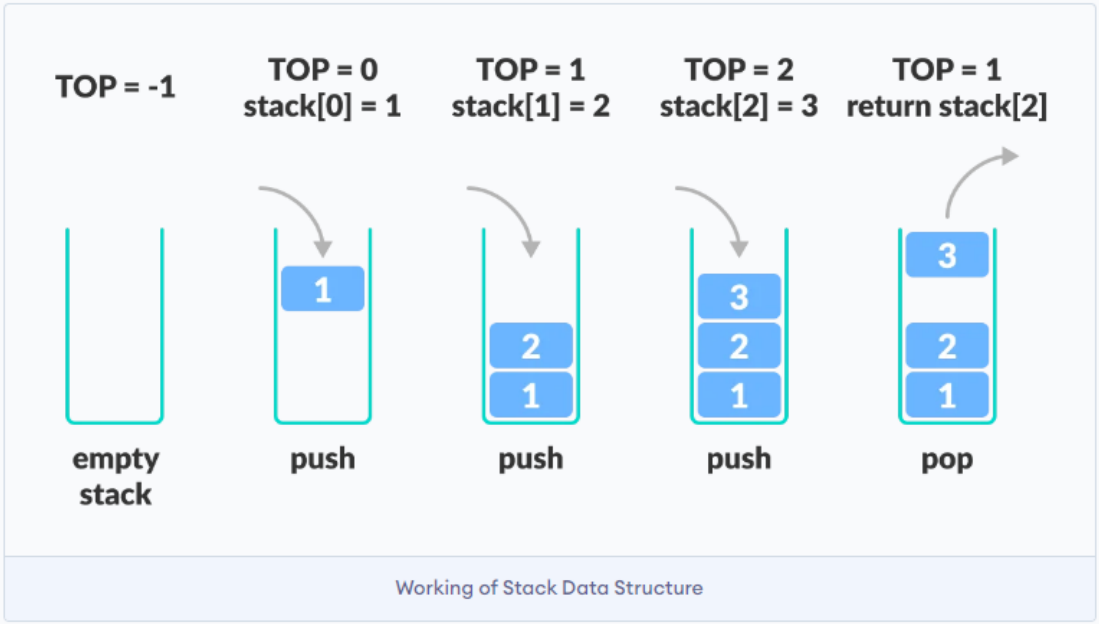
There are some basic operations that allow us to perform different actions on a stack.

- **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.

Working of Stack Data Structure

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



Working of Stack Data Structure

Program:

```
#include <stdio.h>
#include <stdlib.h>

#define MAX 100

int stack[MAX];
int top = -1;

// Function to push an element onto the stack
void push(int element)
```

```
{
   if (top==MAX-1)
   {
      printf("Stack overflow! Cannot push %d\n", element);
   }
        else
        {
           top++;
      stack[top] = element;
      printf("Pushed %d onto the stack\n", element);
        }
}

// Function to pop an element from the stack
void pop()
{
   if (top==-1)
   {
      printf("Stack underflow! Cannot pop\n");
   }
        else
        {
   top--;

        }
}
// Function to peek at the top element of the stack
void peek()
{
   if (top==-1)
   {
      printf("Stack is empty! Cannot peek\n");
   }
   else
   {
      printf("Peek element is %d",stack[top]);

   }
}

void isFull()
```

```
{
   if(MAX-1 ==top)
   {
      printf("Stack is Full");
   }
   else
   {
      printf("Stack is not Full");
   }
}

void isEmpty()
{
   if(top==-1)
   {
      printf("Stack is Empty");
   }
   else
   {
      printf("Stack is not Empty");
   }
}

// Function to display the stack elements
void display()
{
   if (top==-1)
   {
      printf("Stack is empty!\n");

   }
   printf("Stack elements: \n");
   for (int i = top; i>=0; i--)
   {
      printf("%d\n", stack[i]);
   }
   printf("\n");
}

// Function to search for an element in the stack
void search(int element)
```

```
{
    int found = 0;
    printf("Element %d found at positions: ", element);
    for (int i = 0; i <= top; i++)
    {
        if (stack[i] == element)
        {
            printf("%d ", i);
            found = 1;
        }
    }
    if (!found)
    {
        printf("not found in the stack");
    }
    printf("\n");
}
//main method
int main()
{
    int choice, element;

    while (1)
    {
        printf("\n1. Push\n2. Pop\n3. Peek\n4. Display\n5. Search\n6. isEmpty\n7. isFull\n8.
Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
            case 1:
                printf("Enter element to push: ");
                scanf("%d", &element);
                push(element);
                break;
            case 2:
                pop();
                break;
            case 3:
                peek();
```

```
            break;
        case 4:
            display();
            break;
        case 5:
            printf("Enter element to search: ");
            scanf("%d", &element);
            search(element);
            break;
        case 6:
            isEmpty();
            break;
        case 7:
                    isFull();
                    break;
        case 8:
                    exit(0);
        default:
        printf("Invalid choice! Please try again.\n");
      }
   }

   return 0;
}
```

Applications of Stack Data Structure

Although stack is a simple data structure to implement, it is very powerful. The most common uses of a stack are:

- **To reverse a word** - Put all the letters in a stack and pop them out. Because of the LIFO order of stack, you will get the letters in reverse order.
- **In compilers** - Compilers use the stack to calculate the value of expressions like `2 + 4 / 5 * (7 - 9)` by converting the expression to prefix or postfix form.
- **In browsers** - The back button in a browser saves all the URLs you have visited previously in a stack. Each time you visit a new page, it is added on top of the stack. When you press the back button, the current URL is removed from the stack, and the previous URL is accessed.

Polish Notation in Data Structure

Introduction to Polish Notation in Data Structure

This type of notation was introduced by the **Polish mathematician Lukasiewicz**.

Polish Notation in data structure tells us about different ways to write an arithmetic expression. An arithmetic expression contains 2 things, i.e., **operands** and **operators**.

Operands are either numbers or variables that can be replaced by numbers to evaluate the expressions. Operators are symbols symbolizing the operation to be performed between operands present in the expression. Like the expression (1+2)∗(3+4)(1+2)∗(3+4) standard becomes ∗+12+34∗+12+34 in Polish Notation.

Polish notation is also called **prefix notation**. It means that operations are written before the operands. The operators are placed left for every pair of operands. Let's say for the expression a+b, the prefix notation would be +ab.

Types of Notations

Three types of polish notations exist in the data structure. Let's have look at them one-by-one.

- Infix Notation
- Prefix Notation
- Postfix Notation

Infix Notation :

This polish notation in data structure states that the operator is written in between the operands. It is the most common type of notation we generally use to represent expressions. It's the fully parenthesized notation.

Infix Expression
A + B * C + D
(A + B) * (C + D)
A * B + C * D
A + B + C + D

Prefix Notation :

This polish notation in data structure states that the operator should be present as a prefix or before the operands. This notation is also known as **"Polish Notation"**. For example, if we have an expression like x+y, then here x and y are operands, and '+' is the operator. The prefix notation or polish notation of this expression will be "+xy".

Infix Expression	Prefix Expression
A + B * C + D	+ + A * B C D
(A + B) * (C + D)	* + A B + C D
A * B + C * D	+ * A B * C D
A + B + C + D	+ + + A B C D

Postfix Notation :

This notation states that the operator should be present as a suffix, postfix, or after the operands. It is also known as Suffix notation or Reverse Polish Notation. For example, if we have an expression like x+y, then here x and y are operands, and '+' is the operator. The prefix notation or polish notation of this expression will be "xy+".

In general, a computer can easily understand postfix expressions. This notation is universally accepted and is preferred for designing and programming the arithmetic and logical units of a **CPU** (Central Processing Unit). All the expressions that are entered into a computer are converted into **Postfix** or **Reverse Polish Notation**, stored in a stack, and then computed. Therefore, postfix expression plays a vital role in the tech industry.

Infix Expression	Postfix Expression
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +
A + B + C + D	A B + C + D +

Conversion of an Infix Expression to Postfix Expression

Generally, humans find infix polish notation much easier to understand than postfix or reverse polish notation. To convert each expression from infix to postfix, we assign priority to each of the operators present in the expression. Each operator has its priority for an expression. For example, if we take some operators, i.e., +, -, *, /, then these will be arranged in priority.

Operator	Precedence
Parentheses ()	Highest
Exponents ^	High
Multiplication *	Medium
Division /	Medium
Addition +	Low
Subtraction –	Low

ALGORITHM

Scan the symbols of the expression from left to right and for each symbol, do the following:

a..If symbol is an operand

- **Print that symbol onto the screen.**

b. If symbol is a left parenthesis

- **Push it on the stack.**

c. If symbol is a right parenthesis

- **Pop all the operators from the stack upto the first left parenthesis and print them on the screen.**
- **Discard the left and right parentheses.**

d. If symbol is an operator

- **If the precedence of the operators in the stack are greater than or equal to the current operator, then**

else

- **Pop the operators out of the stack and print them onto the screen, and push the current operator onto the stack.**

e. Push the current operator onto the stack.

Expression: A+CB*C)

Character Scanned	Stack	Postfix Expression
A	C	A
+	C	A
C	C+	A
B	C+C	AB
*	C+C	AB
C	C+C*	ABC
)	C+C*	ABC*+

Output : ABC*+

Program:

```
#include <stdio.h>
#define MAX 100
char stack[MAX];
int top = -1;


int isEmpty() {
   return top == -1;
}
int isFull() {
   return top == MAX - 1;
}
void push(char item) {
   if (isFull()) {
      return;
```

```
    }
    stack[++top] = item;
}

char pop() {
    if (isEmpty()) {
        return '\0';
    }
    return stack[top--];
}
char peek() {
    if (isEmpty()) {
        return '\0';
    }
    return stack[top];
}
int isOperand(char ch) {
    return (ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z') || (ch >= '0' && ch <= '9');
}
int precedence(char ch) {
    switch (ch) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '^':
            return 3;
    }
```

```
    return -1;
}
int length(char* exp) {
    int len = 0;
    while (exp[len] != '\0') {
        len++;
    }
    return len;
}
void infixToPostfix(char* exp) {
    int i, k;
    int len = length(exp);

    for (i = 0, k = -1; i < len; i++) {
        if (isOperand(exp[i])) {
            exp[++k] = exp[i];
        } else if (exp[i] == '(') {
            push(exp[i]);
        } else if (exp[i] == ')') {
            while (!isEmpty() && peek() != '(') {
                exp[++k] = pop();
            }
            pop();
        } else {
            while (!isEmpty() && precedence(exp[i]) <= precedence(peek())) {
                exp[++k] = pop();
            }
            push(exp[i]);
        }
    }
```

```
    while (!isEmpty()) {
        exp[++k] = pop();
    }
    exp[++k] = '\0';
    printf("Postfix expression: %s\n", exp);
}
int main() {
    char exp[] = "A*B+C*D";
    infixToPostfix(exp);
    return 0;
}
```

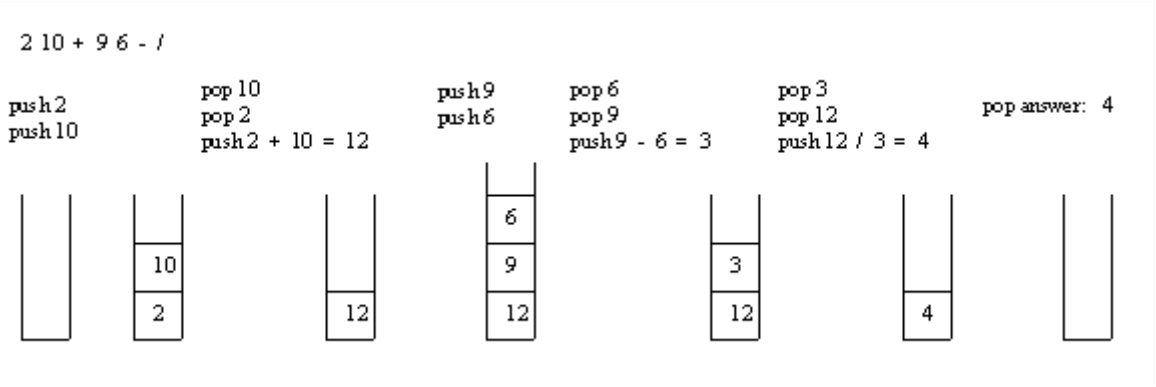
Output: Postfix expression: AB*CD*+

Evaluation of Postfix Expression

Postfix evaluation algorithm is a simple algorithm that allows us to evaluate postfix expressions. The algorithm uses a stack to keep track of operands and performs arithmetic operations when an operator is encountered. The algorithm can be summarized in the following steps:

Algorithm for Postfix Evaluation

1. Create an empty stack called operandStack.
2. Scan the input from left to right.
    - If the input is an operand, convert it from a string to an integer and push the value onto the operand stack.
    - If the input is an operator, *, /, +, or -, it will need two operands. Pop the operandStack twice. The first pop is the second operand and the second pop is the first operand. Perform the arithmetic operation. Push the result back on the operand stack.
3. When the input expression has been completely processed, the result is on the stack. Pop the operand stack and return the value.



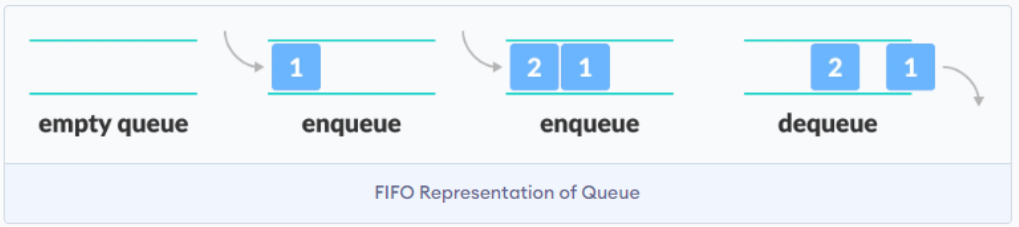
Program:

```
#include <stdio.h>
#define MAX_STACK_SIZE 100
int stack[MAX_STACK_SIZE];
int top = -1;

void push(int value) {
   stack[++top] = value;
}

int pop() {
   return stack[top--];
}

int evaluatePostfix(char* expression) {
   int i;
   for (i = 0; expression[i] != '\0'; i++) {
      if (expression[i] >= '0' && expression[i] <= '9') {
         // Manually converting character to integer
         push(expression[i] - '0');
      } else {
         int right = pop();
         int left = pop();
```

```
            switch (expression[i]) {
                case '+':
                    push(left + right);
                    break;
                case '-':
                    push(left - right);
                    break;
                case '*':
                    push(left * right);
                    break;
                case '/':
                    push(left / right);
                    break;
            }
        }
    }

    return pop();
}

int main() {
    char expression[] = "34+2*7/";
    int result = evaluatePostfix(expression);
    printf("The result of the postfix expression is: %d\n", result);

    return 0;
}
```

Queue Data Structure

A queue is a useful data structure in programming. Queue follows the **First In First Out (FIFO)** Principal - the item that goes in first is the item that comes out first.



FIFO Representation of Queue

In the above image, since 1 was kept in the queue before 2, it is the first to be removed from the queue as well. It follows the **FIFO** rule.

In programming terms, putting items in the queue is called **enqueue**, and removing items from the queue is called **dequeue**.

We can implement the queue in any programming language like C, C++, Java, Python or C#, but the specification is pretty much the same.

Basic Operations of Queue

A queue is an object (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

Working of Queue

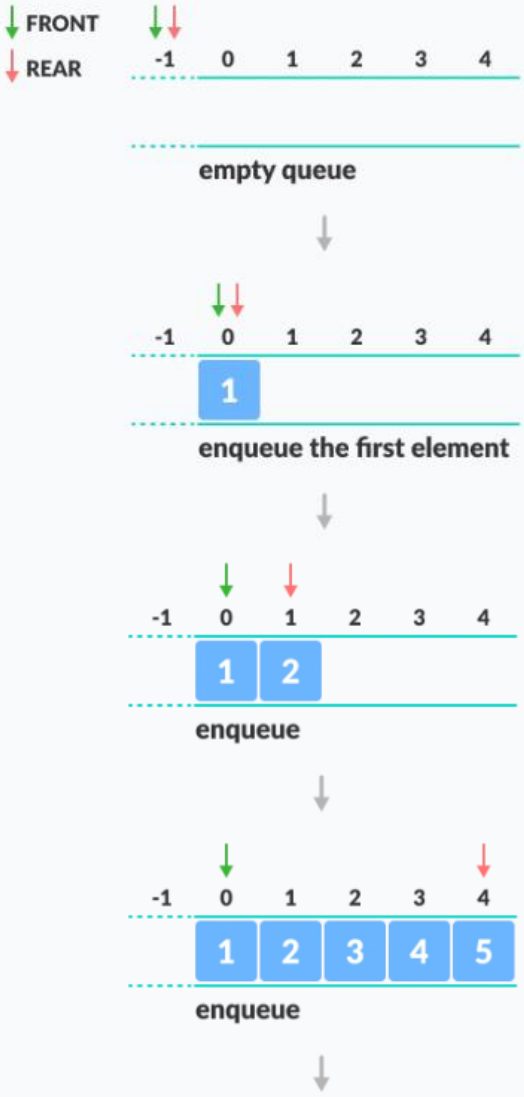
- two pointers *FRONT* and *REAR*
- *FRONT* track the first element of the queue
- *REAR* track the last element of the queue
- initially, set value of *FRONT* and *REAR* to -1

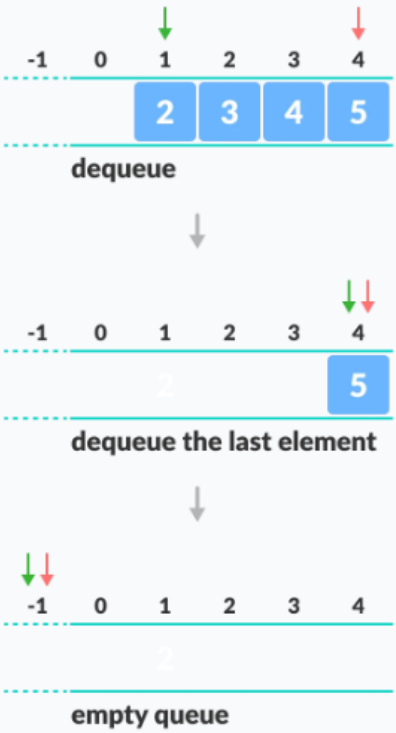
Enqueue Operation

- check if the queue is full
- for the first element, set the value of *FRONT* to 0
- increase the *REAR* index by 1
- add the new element in the position pointed to by *REAR*

Dequeue Operation

- check if the queue is empty
- return the value pointed by *FRONT*
- increase the *FRONT* index by 1
- for the last element, reset the values of *FRONT* and *REAR* to -1



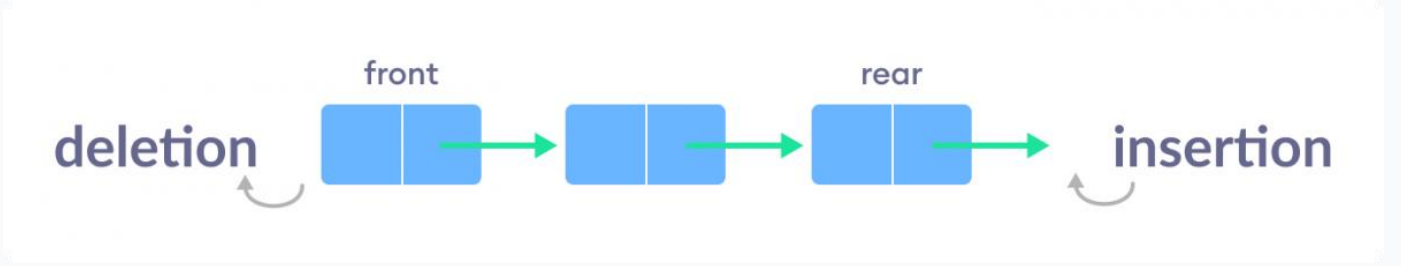


Types of Queues

- Simple Queue
- Circular Queue
- Priority Queue
- Double Ended Queue

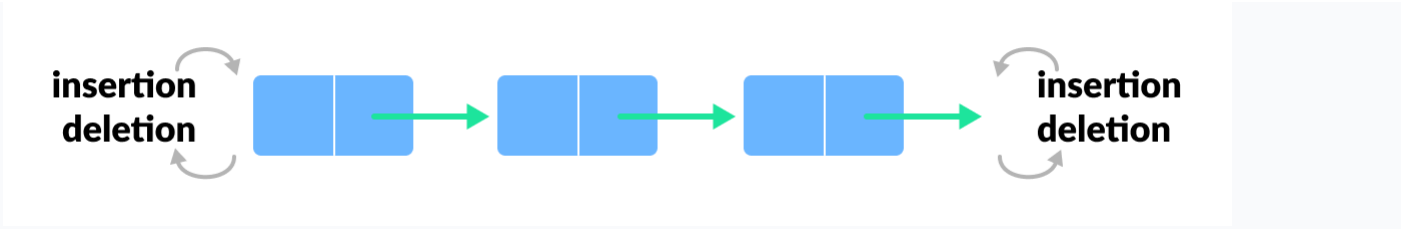
Simple Queue

In a simple queue, insertion takes place at the rear and removal occurs at the front. It strictly follows the FIFO (First in First out) rule.



Deque (Double Ended Queue)

In a double ended queue, insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow the FIFO (First In First Out) rule.



Queue Implementation

Program:

```
#include <stdio.h>
#include <stdlib.h>

int *queue;
int front = 0;
int rear = -1;
int size = 0;
int capacity = 2; // Initial capacity

void initializeQueue() {
    queue = (int*)malloc(capacity * sizeof(int));
    if (queue == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }}
```

```
void resizeQueue() {
    capacity *= 2;
    queue = (int*)realloc(queue, capacity * sizeof(int));
    if (queue == NULL) {
        printf("Memory reallocation failed\n");
        exit(1);
    }
}

void enqueue(int value) {
    if (size == capacity) {
        resizeQueue();
    }
    rear = (rear + 1) % capacity;
    queue[rear] = value;
    size++;
}

int dequeue() {
    if (size == 0) {
        printf("Queue underflow\n");
        exit(1);
    }
    int value = queue[front];
    front = (front + 1) % capacity;
    size--;
    return value;
}

int isEmpty() {
    return size == 0;
}
```

```
int getFront() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        exit(1);
    }
    return queue[front];
}

int getSize() {
    return size;
}

void displayQueue() {
    if (isEmpty()) {
        printf("Queue is empty\n");
    } else {
        printf("Queue elements are: ");
        for (int i = 0; i < size; i++) {
            printf("%d ", queue[(front + i) % capacity]);
        }
        printf("\n");
    }
}

int main() {
    int choice, value;
    initializeQueue();
    while (1) {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Get Front\n");
        printf("4. Get Size\n");
```

```
        printf("5. Display Queue\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                printf("Dequeued value: %d\n", dequeue());
                break;
            case 3:
                printf("Front element is: %d\n", getFront());
                break;
            case 4:
                printf("Queue size is: %d\n", getSize());
                break;
            case 5:
                displayQueue();
                break;
            case 6:
                free(queue);
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice, please try again.\n");
        }
    }
    return 0;
}
```

Output:

Queue Operations:
1. Enqueue
2. Dequeue
3. Get Front
4. Get Size
5. Display Queue
6. Exit
Enter your choice: 1
Enter value to enqueue: 10

Enter your choice: 1
Enter value to enqueue: 20

Enter your choice: 1
Enter value to enqueue: 30

Enter your choice: 5
Queue elements are: 10 20 30

Enter your choice: 2
Dequeued value: 10

Enter your choice: 3
Front element is: 20

Enter your choice: 4
Queue size is: 2

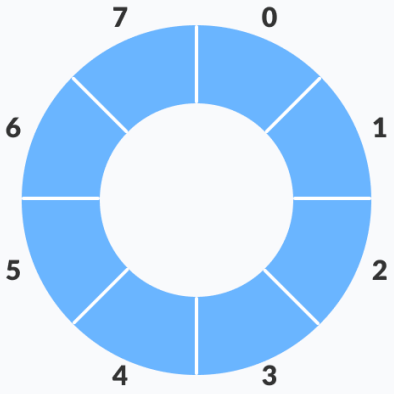
Enter your choice: 1
Enter value to enqueue: 40

Enter your choice: 5
Queue elements are: 20 30 40

Enter your choice: 6
Exiting...

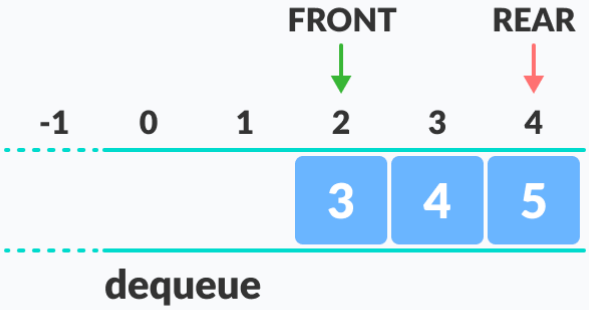
Circular Queue Data Structure

A circular queue is the extended version of a regular queue where the last element is connected to the first element. Thus forming a circle-like structure.



Circular queue representation

The circular queue solves the major limitation of the normal queue. In a normal queue, after a bit of insertion and deletion, there will be non-usable empty space.



Here, indexes **0** and **1** can only be used after resetting the queue (deletion of all elements). This reduces the actual size of the queue.

Circular Queue Operations

The circular queue work as follows:

- two pointers *FRONT* and *REAR*
- *FRONT* track the first element of the queue
- *REAR* track the last elements of the queue

- initially, set value of *FRONT* and *REAR* to -1

1. Enqueue Operation

- check if the queue is full
- for the first element, set value of *FRONT* to 0
- circularly increase the *REAR* index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- add the new element in the position pointed to by *REAR*

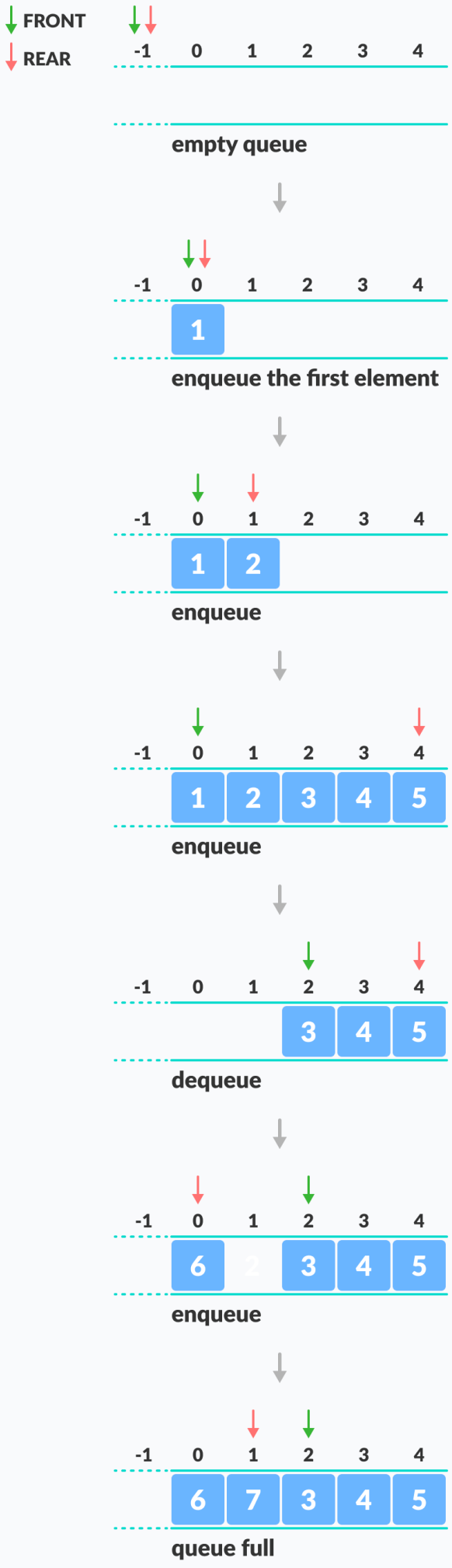
2. Dequeue Operation

- check if the queue is empty
- return the value pointed by *FRONT*
- circularly increase the *FRONT* index by 1
- for the last element, reset the values of *FRONT* and *REAR* to -1

However, the check for full queue has a new additional case:

- Case 1: *FRONT* = 0 && REAR == SIZE - 1
- Case 2: FRONT = REAR + 1

The second case happens when *REAR* starts from 0 due to circular increment and when its value is just 1 less than *FRONT*, the queue is full.


FRONT
REAR
-1 0 1 2 3 4
empty queue
-1 0 1 2 3 4
1
enqueue the first element
-1 0 1 2 3 4
1 2
enqueue
-1 0 1 2 3 4
1 2 3 4 5
enqueue
-1 0 1 2 3 4
3 4 5
dequeue
-1 0 1 2 3 4
6 3 4 5
enqueue
-1 0 1 2 3 4
6 7 3 4 5
queue full

Circular queues using Dynamic Array

Program:

```
#include <stdio.h>
#include <stdlib.h>

int *queue;
int front = -1;
int rear = -1;
int size = 0;
int capacity = 2; // Initial capacity

void initializeQueue()
{
    queue = (int*)malloc(capacity * sizeof(int));
    if (queue == NULL)
    {
        printf("Memory allocation failed\n");
        exit(1);
    }
}

void resizeQueue()
{
    int newCapacity = capacity * 2;
    int *newQueue = (int*)malloc(newCapacity * sizeof(int));
    if (newQueue == NULL) {
        printf("Memory reallocation failed\n");
        exit(1);
    }
    for (int i = 0; i < size; i++)
```

```
    {
        newQueue[i] = queue[(front + i) % capacity];
    }
    free(queue);
    queue = newQueue;
    front = 0;
    rear = size - 1;
    capacity = newCapacity;
}

void enqueue(int value) {
    if (size == capacity) {
        resizeQueue();
    }
    rear = (rear + 1) % capacity;
    queue[rear] = value;
    if (front == -1) // If queue was empty, set front to 0
        front = 0;
    size++;
}

int dequeue() {
    if (size == 0) {
        printf("Queue underflow\n");
        exit(1);
    }
    int value = queue[front];
    front = (front + 1) % capacity;
    size--;
    if (size == 0) { // If queue becomes empty, reset front and rear
```

```
        front = -1;
        rear = -1;
    }
    return value;
}

int isEmpty() {
    return size == 0;
}

int getFront() {
    if (isEmpty()) {
        printf("Queue is empty\n");
        exit(1);
    }
    return queue[front];
}

int getSize() {
    return size;
}

void displayQueue() {
    if (isEmpty()) {
        printf("Queue is empty\n");
    } else {
        printf("Queue elements are: ");
        for (int i = 0; i < size; i++) {
            printf("%d ", queue[(front + i) % capacity]);
        }
```

```
        printf("\n");
    }
}

int main() {
    int choice, value;

    initializeQueue();

    while (1) {
        printf("\nQueue Operations:\n");
        printf("1. Enqueue\n");
        printf("2. Dequeue\n");
        printf("3. Get Front\n");
        printf("4. Get Size\n");
        printf("5. Display Queue\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to enqueue: ");
                scanf("%d", &value);
                enqueue(value);
                break;
            case 2:
                printf("Dequeued value: %d\n", dequeue());
                break;
            case 3:
```

```
            printf("Front element is: %d\n", getFront());
            break;
        case 4:
            printf("Queue size is: %d\n", getSize());
            break;
        case 5:
            displayQueue();
            break;
        case 6:
            free(queue);
            printf("Exiting...\n");
            exit(0);
        default:
            printf("Invalid choice, please try again.\n");
      }
   }
   return 0;
}
```