Task No.01

1. Explain the definition of Asymptotic notations (Big O, Big Omega Big Theta)

- **Big-O**

  We define an algorithm's worst-case time complexity by using the Big-O notation, which determines the set of functions grows slower than or at the same rate as the expression. Furthermore, it explains the maximum amount of time an algorithm requires to consider all input values.

- **Big Omega**

  It defines the best case of an algorithm's time complexity, the Omega notation defines whether the set of functions will grow faster or at the same rate as the expression. Furthermore, it explains the minimum amount of time an algorithm requires to consider all input values.

- **Big Theta**

  It defines the average case of an algorithm's time complexity, the Theta notation defines when the set of functions lies in both O(expression) and Omega(expression), then Theta notation is used. This is how we define a time complexity average case for an algorithm.

2. Illustrate the main applications of stack?

- **Function Calls:** When a program calls a function, the stack stores the return address. This ensures the program knows where to resume execution after the function finishes.
- **Recursion:** During recursion, a function calls itself. Stacks come in handy here by storing the local variables and return addresses of each recursive call. This allows the program to track the current state of the recursion. Towers of Hanoi is a good example on that.
- **Expression Evaluation:** Stacks are efficient for evaluating expressions in postfix notation (Reverse Polish Notation). In postfix, operands come before operators, so a stack can process them in the correct order.

3. Apply the array and stack using C++ or java programming language

```
// ================ header file <Stack.h> ================
#pragma once
#define Stack_Entry int
#define Max_Stack 100
enum Error_code {
    underflow = -1,
    success = 0,
    overflow = 1
};
class Stack
{
private:
    Stack_Entry entry[Max_Stack];
    int top;
public:
    Stack();
    bool empty();
    Error_code pop(Stack_Entry &e);
    Error_code push(Stack_Entry e);
    bool full();
    void traverse( void (*pf)(Stack_Entry) );
    Error_code clear();
    int size();
    Error_code ctop(Stack_Entry &e);
};


```

```cpp
//  ================ Implementation file <Stack.cpp> ================
#include "Stack.h"
Stack::Stack() {
    top = 0;
}

Error_code Stack::pop(Stack_Entry &e) {
    if (empty()) {
        return underflow;
    }
    e = entry[--top];
    return success;
}

bool Stack::empty() {
    return (!top);
}

Error_code Stack::push(Stack_Entry e) {
    if (top==Max_Stack) {
        return overflow;
    }
    entry[top++] = e;
    return success;
}

bool Stack::full() {
    return (top== Max_Stack);
}

Error_code Stack::clear() {
    top = 0;
    return success;
}

int Stack::size() {
    int out = top-1;
    return out;
}

void Stack::traverse(void (*pf)(Stack_Entry)) {
    for (int i = top - 1; i >= 0; i--) {
        (*pf)(entry[i]);
    }
}

Error_code Stack::ctop(Stack_Entry& e) {
    e = entry[top - 1];
    return success;
}
```

Task No.02

1. Explain a concrete data structure for a First In First out (FIFO) and illustrate the main applications of it?

- **Data Structure:** Queue
  A Queue is a linear data structure that follows the principle of FIFO. Items are inserted at the back (rear) and removed from the front. Imagine a line at a store; the first person in line (front) gets served first, and new customers join at the back.

- **Main Applications of FIFO:**
  - **Task Scheduling:** Operating systems use FIFO queues to manage processes. Processes are added to a queue and executed in the order they are received.
  - **Breadth-First Search (BFS) in Graphs:** BFS algorithms in graphs explore neighboring nodes level by level. A FIFO queue ensures that nodes are explored in the order they are discovered.
  - **Printer Queues:** Print jobs are typically added to a FIFO queue and processed one by one.
  - **Data Stream Processing**: FIFO queues can buffer incoming data for real-time processing.

2. Define the operation of linked list and compare between the different types of linked lists?

- **Linked List Operations**
  - **Traversal:** Visit each node in the list, typically starting from the head and following the pointers.
  - **Insertion:** Add a new node at a specific position (beginning, end, or in-between).
  - **Deletion:** Remove an existing node based on its data or position.
  - **Search:** Find a node containing a specific data value.

- **Types of Linked Lists**
  - **Singly Linked List:** The most basic type. Each node has a data field and a pointer to the next node.
    - Advantages: Simple to implement, efficient for insertion/deletion at the beginning.
    - Disadvantages: Inefficient for insertion/deletion in the middle or end, as requires traversing to the target node.

- **Doubly Linked List:** Each node has a data field, a pointer to the next node, and a pointer to the previous node.
    - **Advantages:** Enables efficient insertion/deletion at any position, as you can navigate both directions.
    - **Disadvantages:** More complex structure due to the extra pointer, consumes slightly more memory.
- **Circular Linked List:** Similar to a singly linked list, but the last node points back to the first node, creating a loop.
    - **Advantages:** Useful for representing circular structures or implementing algorithms that require continuous traversal (e.g., Josephus problem).
    - **Disadvantages:** Can't be used for scenarios where a defined end is necessary.

3. Apply the queues using C++ or Java accurately?

```cpp
// ================ header file <queue.h> ================
#pragma once
#define Queue_Entry int
#define Max_Queue 10
enum Error_Code {
    underflow = -1,
    success = 0,
    overflow = 1
};
class Queue
{
private:
    int front, rear, size;
    Queue_Entry entry[Max_Queue];
public:
    Queue();
    Error_Code append(Queue_Entry e);
    Error_Code serve(Queue_Entry &e);
    int get_size();
    bool empty();
    bool full();
    void clear();

};


```

```cpp
// ================ implementation file <queue.cpp> ================
#include "Queue.h"

Queue::Queue() {
    front = size = 0;
    rear = -1;
}
int Queue::get_size() {
    return size;
}
bool Queue::empty() {
    return !size;
}
bool Queue::full() {
    return (size == Max_Queue);
}
Error_Code Queue::append(Queue_Entry e) {
    if (full()) return overflow;
    rear = (rear + 1) % Max_Queue;
    entry[rear] = e;
    size++;
    return success;
}
Error_Code Queue::serve(Queue_Entry &e){
    if (empty()) return underflow;
    e = entry[front];
    front = (front + 1) % Max_Queue;
    size--;
    return success;
}
int Queue::get_size() {
    return size;
}
void Queue::clear() {
    front = size = 0;
    rear = -1;
}
```