STACKS

A stack is a non-primitive linear data structure. It is an ordered list in which addition of new data item and deletion of already existing data item is done from only one end, known as Top of Stack (TOS). As all the deletion and insertion in a stack is done from top of the stack, the last added element will be the first to be removed from the stack. Due to this reason, the stack is also called Last-In-First-Out (LIFO) type of list. Consider some

Stack Examples (LIFO - Last-In, First-Out)

- **Browser history: When you click the "back" button, you return to the most recently visited page.**
- **Undo/Redo features: In a text editor, clicking "undo" reverts the last action performed.**
- **Stacked plates: A stack of physical plates demonstrates the LIFO principle; the plate on top is the first to be removed.**

SEQUENTIAL IMPLEMENTATION OF STACKS Stack can be implemented in two ways :

(a) Static implementation

(b) Dynamic implementation

Static implementation- Static implementation uses arrays to create stack. Static implementation is a very simple technique, but is not a flexible way of creation, as the size of stack has to be declared during program design, after that the size cannot be varied.

Dynamic implementation- e if we implement a stack using a linked list. In case of a linked list we shall push and pop nodes from one end of a linked list. Linked list representation is commonly known as Dynamic implementation

The node in the list is a structure as shown below :

```
struct node
{
data;
```

node *link; };

where indicates that the data can be of any type like int, float, char etc, and link, is a pointer to the next node in the list.

The pointer to the beginning of the list serves the purpose of the top of the stack.

OPERATIONS ON STACK

The basic operations that can be performed on stack are as follows :

1. **PUSH : The process of adding a new element to the top of the stack is called PUSH operation. Pushing an element in the stack involve adding of element, as the new element will be inserted at the top, so after every push operation, the top is incremented by one. In case the array is full and no new element can be accommodated, it is called STACK-FULL condition. This condition is called STACK OVERFLOW.**

   Algorithm for inserting an item into the stack (PUSH)
   Let STACK[MAXSIZE] is an array for implementing the stack,
   MAXSIZE represents the max. size of array STACK.
   NUM is the element to be pushed in stack &
   TOP is the index number of the element at the top of stack.

   Step 1 : [Check for stack overflow ?]

   If TOP = MAXSIZE – 1,

   then : Write : 'Stack Overflow' and return.

   [End of If Structure]

   Step 2 : Read NUM to be pushed in stack.

   Step 3 : Set TOP = TOP + 1 [Increases TOP by 1]

   Step 4 : Set STACK[TOP] = NUM [Inserts new number NUM in new TOP Position]

   Step 5 : Exit

POP : The process of deleting an element from the top of the stack is called POP operation. After every pop operation, the stack is decremented by one. If there is no element on the stack and the pop is performed then this will result into STACK UNDERFLOW condition.

Let STACK[MAXSIZE] is an array for implementing the stack where MAXSIZE represents the max. size of array STACK. NUM is the element to be popped from stack & TOP is the index number of the element at the top of stack.

Step 1 : [Check for stack underflow ?]

If TOP = -1 : then Write : 'Stack underflow' and return.

[End of If Structure]

Step 2 : Set NUM = STACK[TOP] [Assign Top element to NUM]

Step 3 : Write 'Element popped from stack is : ',NUM.

Step 4 : Set TOP = TOP - 1 [Decreases TOP by 1]

Step 5 : Exit

Multiple stacks

in data structures refer to the concept of managing and utilizing more than one stack within a single program or data structure. This can be implemented in various ways, often with the goal of optimizing memory usage or handling distinct data flows.

Implementation of Multiple Stack

To implement multiple stack in an array, we need to divide the array into multiple sections, with each section representing a separate stack. We also need to keep track of the top pointer for each stack.

Here is a step-by-step implementation of multiple stack:

1. **Define the size of the array and the number of stacks you want to create.**
2. **Divide the array into equal sections for each stack.**
3. **Create an array to store the top pointers for each stack.**
4. **Initialize the top pointers for each stack to -1, indicating an empty stack.**
5. **Implement the push operation for each stack:**

- **Check if the stack is full by comparing the top pointer with the size of the stack.**
- **If the stack is full, display an overflow message.**
- **If the stack is not full, increment the top pointer for that stack and insert the element at the corresponding position in the array.**

Understanding Recursion

Recursion is a programming technique that involves a function calling itself, either directly or indirectly. This can be a powerful tool for solving complex problems, as it allows a program to break down a problem into smaller subproblems that can be solved recursively.

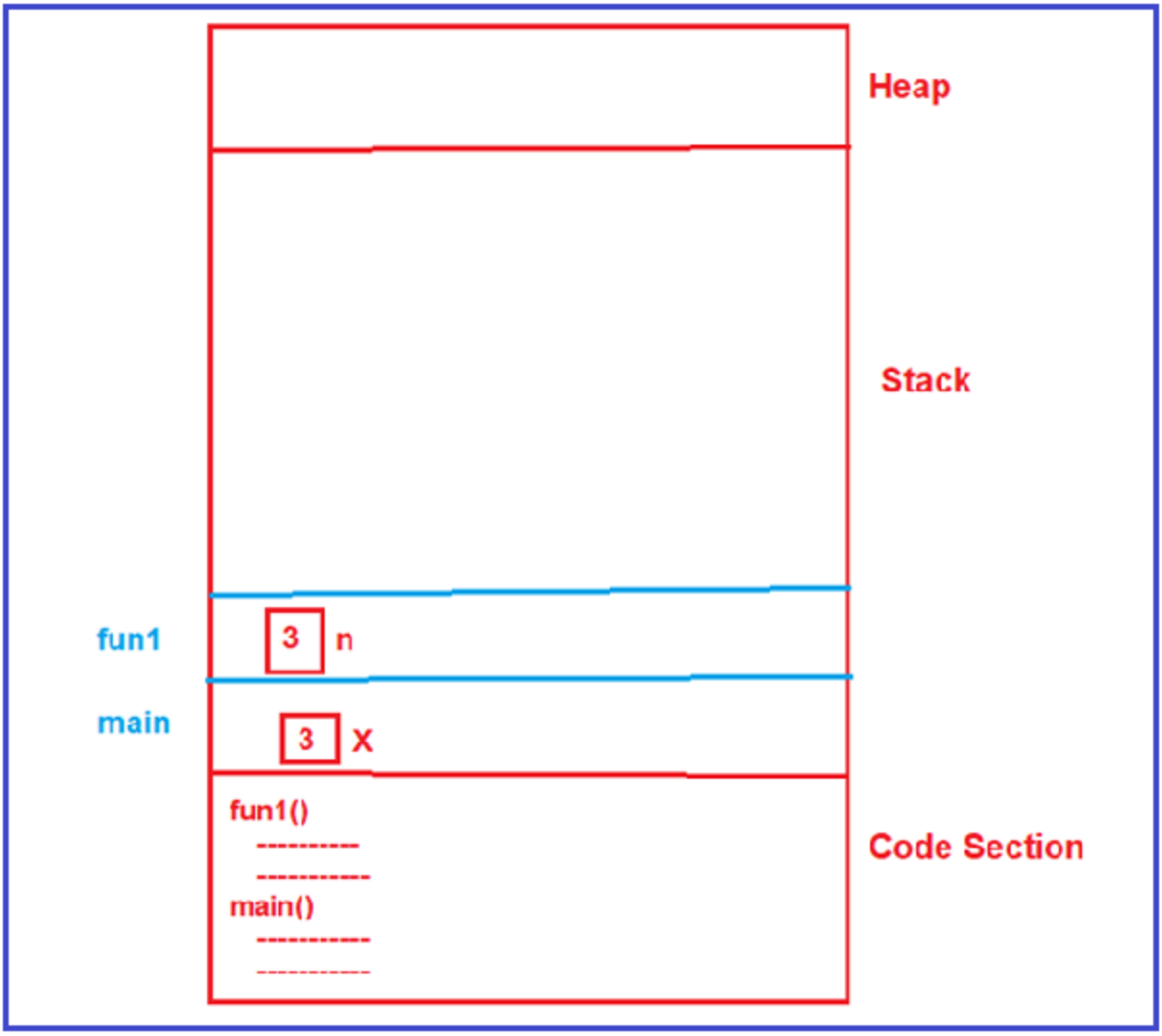
How Recursion Uses Stack

In this article, I am going to discuss **How Recursion uses Stack** in detail. Please read our previous article, where we discussed **how recursion works**. We already discussed that the memory is used by dividing into three sections i.e. code section, stack section, and heap section. We will take the following example and will show you how the stack is created and utilized as a recursive function

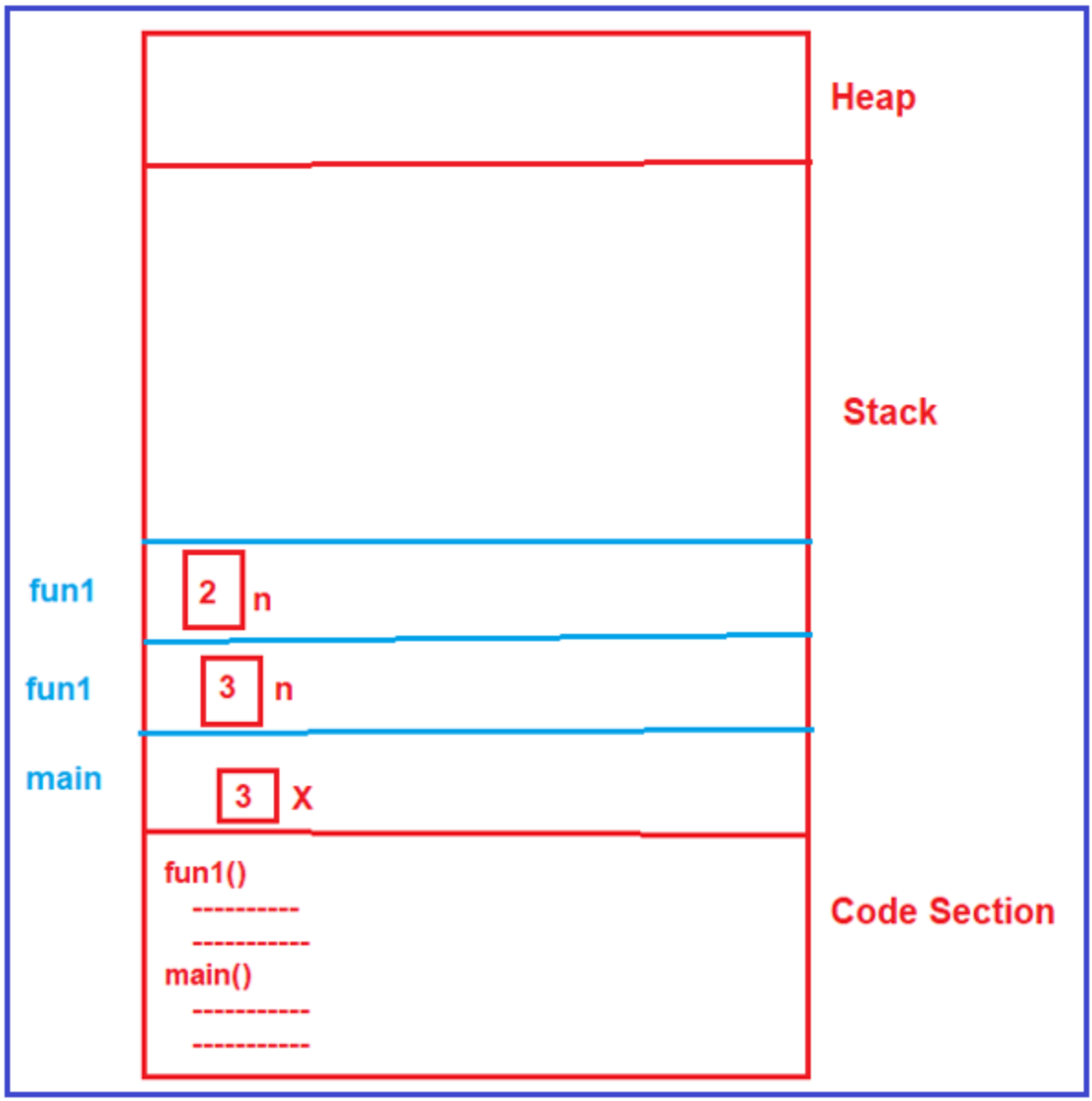
```
void fun1(int n)
{
    if (n>0)
    {
       printf("%d",n);
       fun1(n-1)
    }
}

void main()
{
    int x=3;
    fun1(x);
}
```

The program execution starts from the main functions. Inside the main function, int x=3; is the first statement that is X is created. Inside the stack, the activation record for the main function is created and it will have its own variable that is X having value 3.

The very next statement is fun1() i.e. call to the fun1 function. So. once the function fun1() is called, it is having just one variable that is n. The activation record for that fun1() function is created and it will have that variable n and the value x is passed to that n variable, so it will store the value 3. For better understanding, please have a look at the below image. This is the first call of the fun1 function.

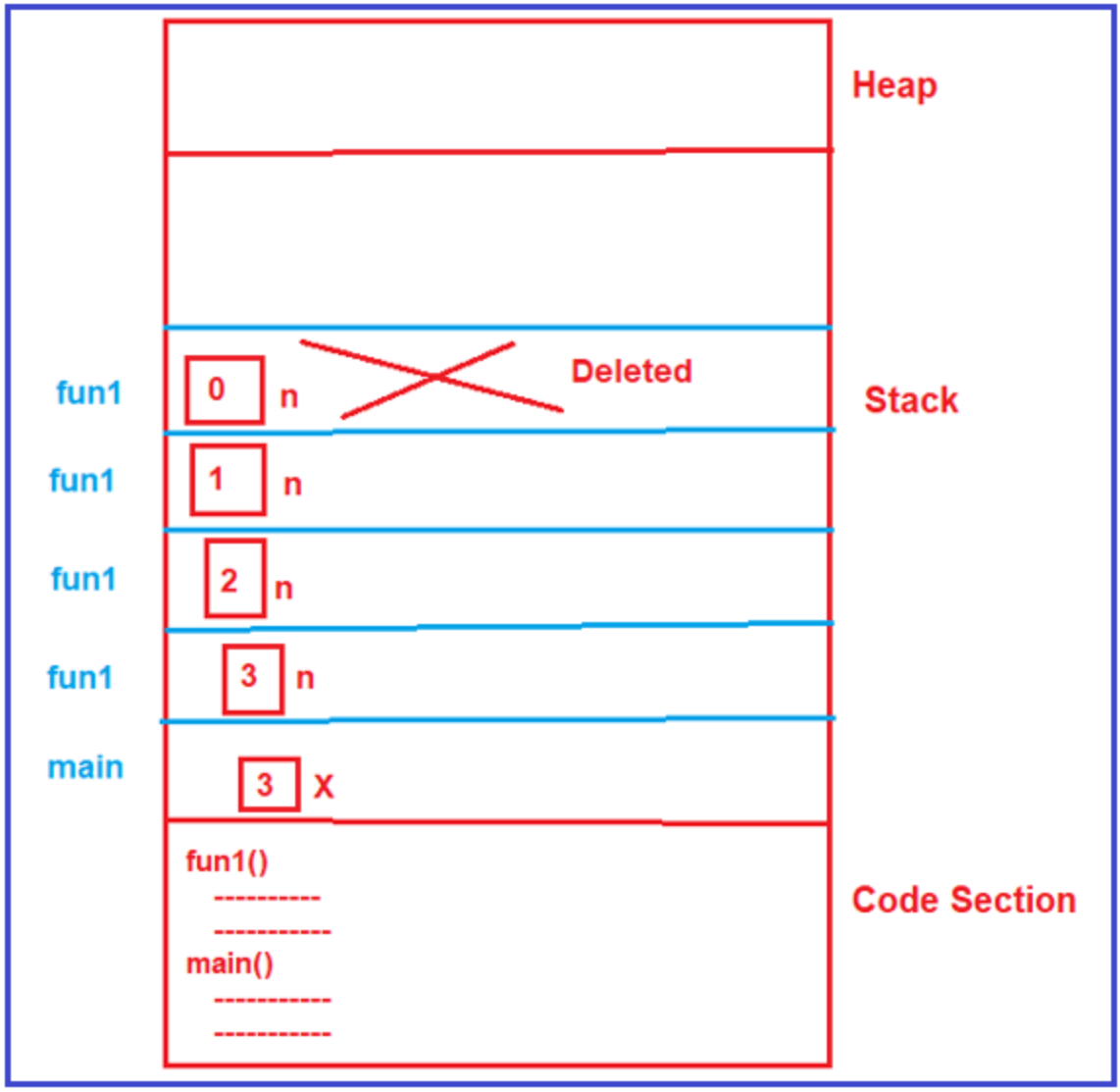


Let's continue. Inside the fun1 function, first, it will check whether n is greater than 0. Yes, n (3) is greater than 0 and the condition satisfies. So, it will print the value 3 and will call the fun1() function with the reduced value of n i.e. n-1 i.e. 2. Once the fun1 function is called, again another activation record for that function is created inside the stack. Within this activation record, again the variable n is created with the value 2 as shown in the below image. This is the second call of the fun1 function.


Heap
Stack
fun1
2
n
fun1
3
n
main
3
X
fun1()

---
main()

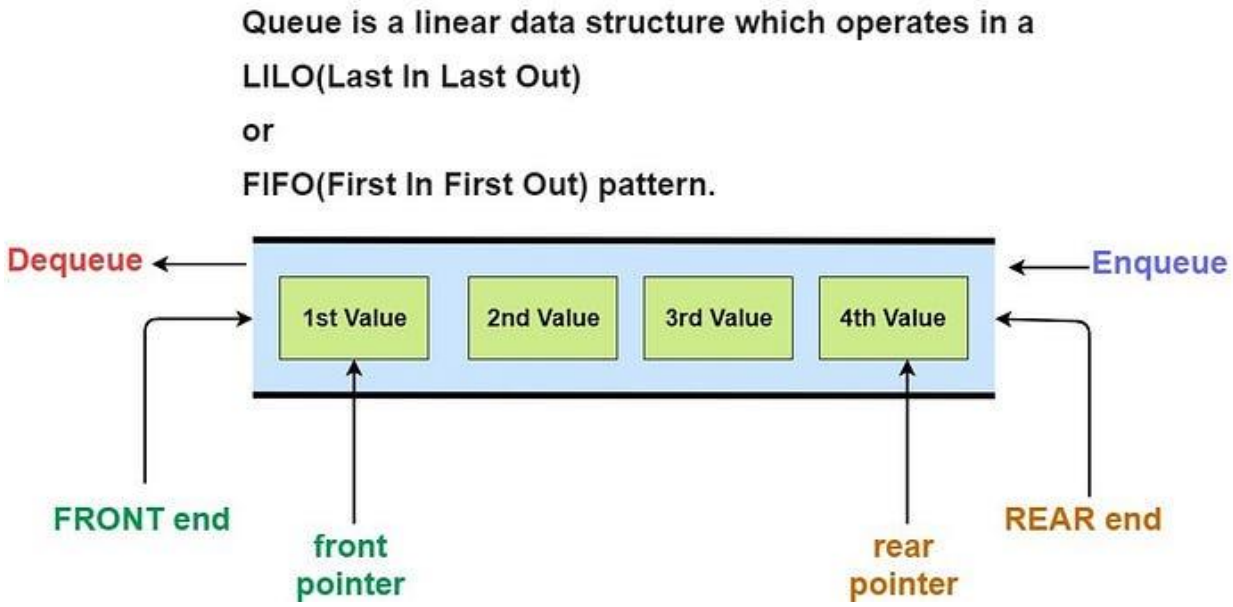
Code Section



When a function call is completed, its stack frame is removed from the stack, and control returns to the address specified in the return address of the now topmost stack frame. This process continues until all function calls are completed and the stack is empty.

What is Queue in Data Structure?

A queue in data structures is a linear collection of elements that operates under the First In, First Out (FIFO) principle. This means that the first element added to the queue will be the first one removed.



If the queue is represented as a list of names:

- **Enqueue (Join the Queue): John, Mary, and Alice join the queue in this order. The queue now looks like [John, Mary, Alice], where John is at the front.**
- **Dequeue (Leave the Queue): John buys his ticket and leaves the queue. Now, the queue is [Mary, Alice], with Mary at the front ready to be the next to buy a ticket.**

Operations of Queue in Data Structure

The following are the primary operations performed on a queue data structure:

Enqueue

The enqueue operation involves adding an element to the rear of the queue. When you "enqueue," you are putting an item at the end of the line, waiting to be processed.

This operation may involve updating the rear pointer of the queue to point to the new element, which becomes the new rear of the queue.

Dequeue

The dequeue operation removes an element from the front of the queue. This is the primary action reflecting the FIFO nature of the queue; the oldest element added (the one at the front) gets removed and processed first.

After the dequeue, the front pointer of the queue needs to be updated to the next element.

Peek or Front

This operation allows you to look at the front element of the queue without removing it. It's useful when you need to check which element is next to be processed but do not want to dequeue it yet. This operation simply retrieves the value of the front element.

IsEmpty

The isEmpty operation checks if the queue is empty. This is crucial to avoid errors such as underflow when attempting to dequeue from an empty queue. If the front of the queue is null (or the front pointer equals the rear pointer in an empty circular queue), the queue is empty.

types of queue in data structure in depth-

Simple Queue

In a simple queue, we follow the First In, First Out (FIFO) rule. Insertion takes place at one end, i.e., the rear end or the tail of the queue, and deletion takes place at the other end, i.e., the front end or the head of the queue.

Circular Queue-

A Circular Queue, also known as a Ring Buffer, is a type of linear data structure that adheres to the FIFO (First In, First Out) principle.

Priority Queue -

A priority queue also exhibits similar characteristics to a simple queue where each element is assigned a particular priority value, where elements in the queue are assigned based on priority.

- There are two types of priority queues:

Ascending Order: In this priority queue, the elements are arranged in ascending Order of their priority, i.e., the element with the smallest priority comes at the start, and the element with the greatest priority comes at the end.

Descending Order: In this priority queue, the elements are arranged in descending Order of their priority, i.e., the element with the greatest priority is at the start, and the element with the smallest priority is present at the end of the queue.

.Double-ended Queue- Double Ended Queue

A Double-ended Queue, or Deque, is a different type of queue where enqueue (insertion) and dequeue (deletion) operations are performed at both ends, i.e., the rear-end (tail) and the front-end (head).

Applications of Queues-

Queues are used in a variety of real-world and computational scenarios:

1. CPU Scheduling: Managing processes in an operating system.
2. Disk Scheduling: Organizing read/write requests in a storage system.
3. Network Buffers: Handling packets in network routers.
4. Print Spooling: Managing print jobs in a printer queue.
5. Breadth-First Search (BFS): In graph algorithms.

Queue simulation

Queue simulation in data structures involves using the queue data structure to model real-world scenarios where items or entities wait in a line to be processed or served in a First-In-First-Out (FIFO) manner. This simulation helps in understanding and analyzing the behavior of such systems.

Key Concepts in Queue Simulation:

- **Events:**
  The simulation progresses based on events, such as:
- **Arrivals:** New items entering the queue.

- **Departures:** Items being served and leaving the queue.
  Simulation Clock:
  A variable that tracks the passage of time within the simulation, often incremented in discrete steps.
  .
  Metrics:
  The goal of queue simulation is often to gather performance metrics, such as:
- Average waiting time in the queue.
- Average service time.
- Queue length over time.
- Utilization of the server(s).

Examples of Queue Simulations:

- **Printing Tasks:**

  Simulating a printer queue where print jobs arrive and are processed in order. The simulation might analyze average waiting time for print jobs.
- **Customer Service:**

  Modeling a customer service line where customers arrive and wait for a service agent. This can help determine optimal staffing levels.
- **Operating Systems:**

  Simulating job scheduling in an operating system, where processes wait for CPU time.
- **Traffic Flow:**
  Simulating traffic at an intersection or on a road, where vehicles form queues.