UNIT – II

SYLLABUS:
INTRODUCTION TO ADT:
Stack: Definition, Array Representation of Stack, Operations on Stacks.
Applications of Stack: Expression evaluation, Conversion of Infix to Postfix, Infix to Prefix, Recursion, Tower of Hanoi
Queue: Definition, Representation of Queues, Operations of Queues, Circular Queue.
Applications of Queue: Job Scheduling, A Maze Problem

INTRODUCTION TO ADT:

In this Unit, we will learn about ADT but before understanding what ADT is let us consider different in-built data types that are provided to us. Data types such as int, float, double, long, etc. are in-built data types and we can perform basic operations with them such as addition, subtraction, division, multiplication, etc. Now there might be a situation when we need operations for our user-defined data type which must be defined. These operations can be defined only as and when we require them. So, in order to simplify the process of solving problems, we can create data structures along with their operations, and such data structures that are not in-built are known as Abstract Data Type (ADT).

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of values and a set of operations. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called "abstract" because it gives an implementation-independent view.

The process of providing only the essentials and hiding the details is known as abstraction.

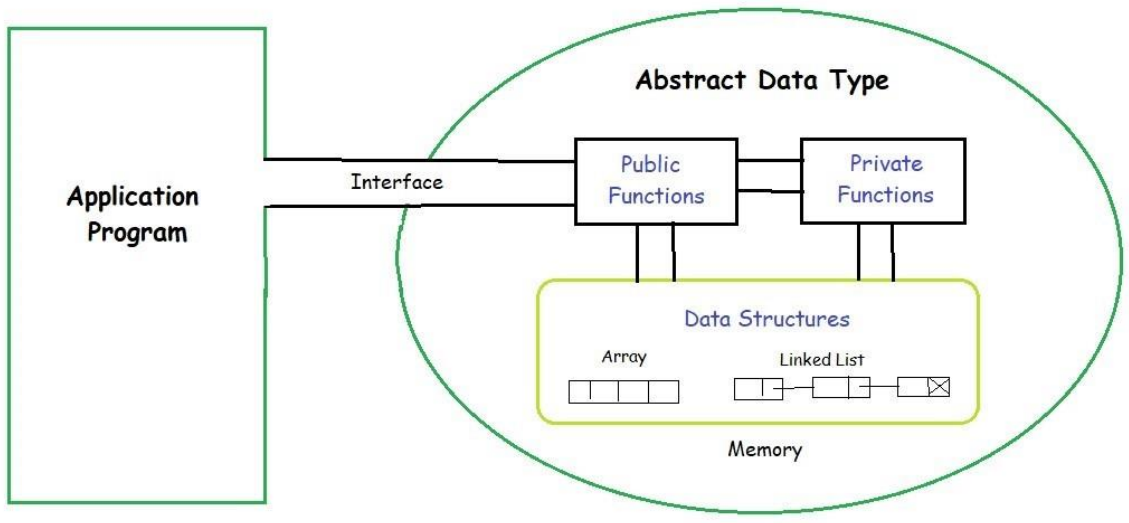


Fig: Process of Abstract Data Type

The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.

So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

1. List ADT

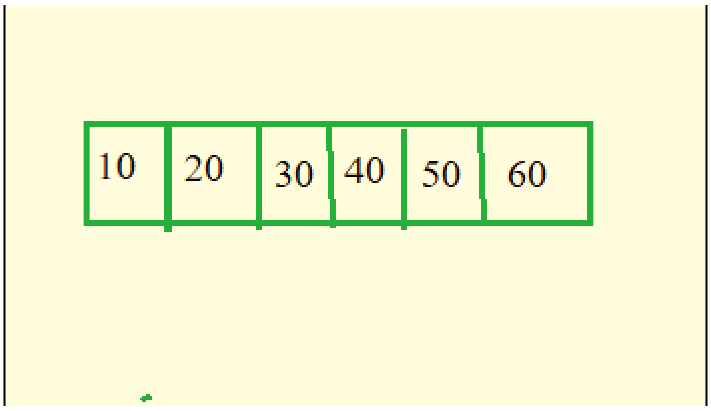


Fig: Vies of list

The data is generally stored in key sequence in a list which has a head structure consisting of count, pointers and address of compare function needed to compare the data in the list.

The data node contains the pointer to a data structure and a self-referential pointer which points to the next node in the list.

The List ADT Functions is given below:

- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace() – Replace an element at any position by another element.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise return false.
- isFull() – Return true if the list is full, otherwise return false.

2. Stack ADT

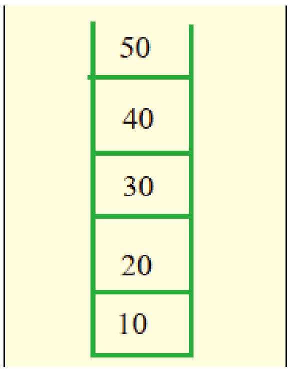


Fig: View of stack

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
The program allocates memory for the data and address is passed to the stack ADT.
The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
The stack head structure also contains a pointer to top and count of number of entries currently in stack.

push() – Insert an element at one end of the stack called top.
pop() – Remove and return the element at the top of the stack, if it is not empty.
peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
size() – Return the number of elements in the stack.
isEmpty() – Return true if the stack is empty, otherwise return false.
isFull() – Return true if the stack is full, otherwise return false.

3. Queue ADT

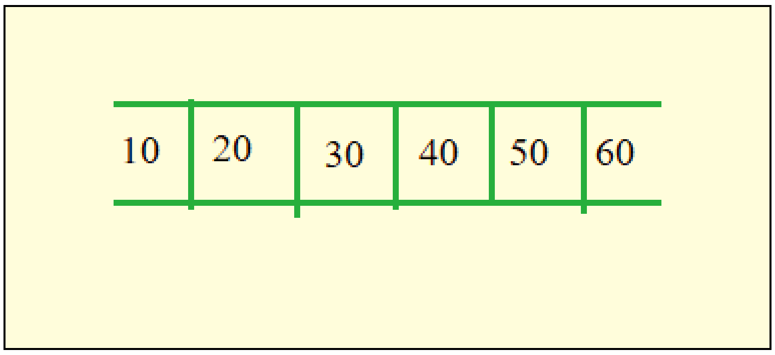


Fig: View of Queue

The queue abstract data type (ADT) follows the basic design of the stack abstract data type. Each node contains a void pointer to the data and the link pointer to the next element in the queue. The program's responsibility is to allocate memory for storing the data.

- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- peek() – Return the element of the queue without removing it, if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull() – Return true if the queue is full, otherwise return false.

Features of ADT:

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- Abstraction: The user does not need to know the implementation of the data structure only essentials are provided.
- Better Conceptualization: ADT gives us a better conceptualization of the real world.
- Robust: The program is robust and has the ability to catch errors.
- Encapsulation: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- Data Abstraction: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- Data Structure Independence: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- Information Hiding: ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.

- Modularity: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

Overall, ADTs provide a powerful tool for organizing and manipulating data in a structured and efficient manner.

Abstract data types (ADTs) have several advantages and disadvantages that should be considered when deciding to use them in software development. Here are some of the main advantages and disadvantages of using ADTs:

Advantages:

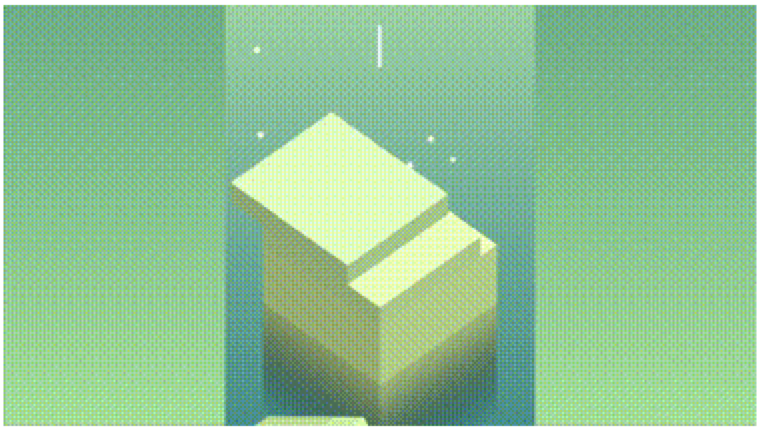
- Encapsulation: ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
- Abstraction: ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.
- Data Structure Independence: ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.
- Information Hiding: ADTs can protect the integrity of data by controlling access and preventing unauthorized modifications.
- Modularity: ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming.

Disadvantages:

- Overhead: Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
- Complexity: ADTs can be complex to implement, especially for large and complex data structures.
- Learning Curve: Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
- Limited Flexibility: Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.
- Cost: Implementing ADTs may require additional resources and investment, which can increase the cost of development.

Overall, the advantages of ADTs often outweigh the disadvantages, and they are widely used in software development to manage and manipulate data in a structured and efficient way. However, it is important to consider the specific needs and requirements of a project when deciding whether to use ADTs. From these definitions, we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.

Stack:



Definition: 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.

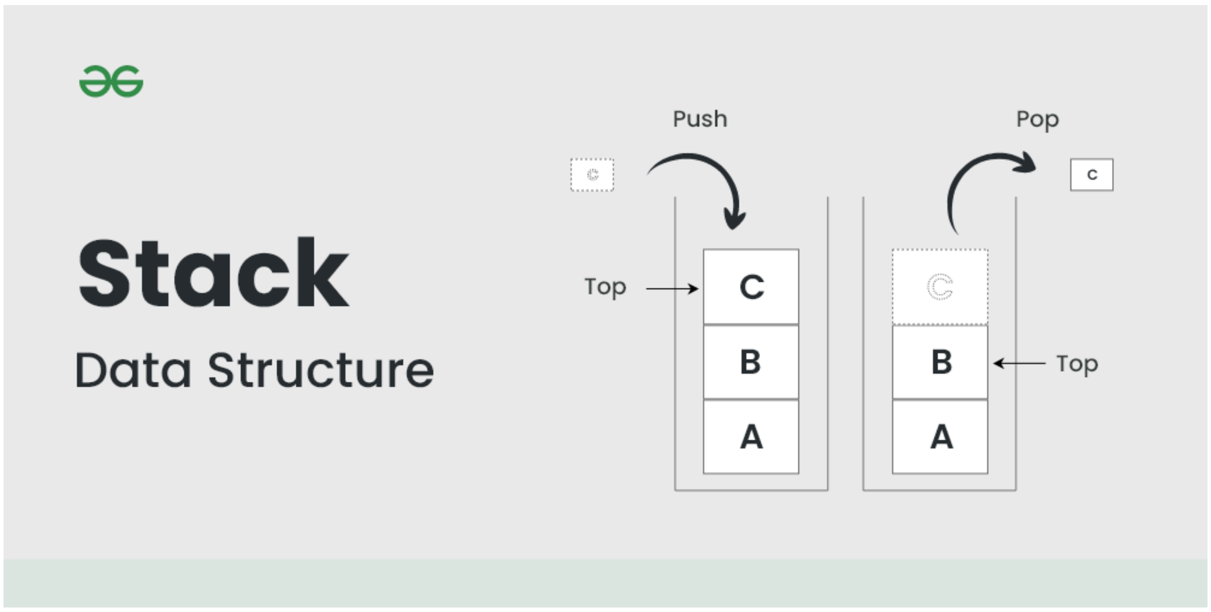


Fig: Stack Data Structure

There are many real-life examples of a stack. Consider an example of plates stacked over one another in the canteen. The plate which is at the top is the first one to be removed, i.e. the plate which has been placed at the bottommost position remains in the stack for the longest period of time. So, it can be simply seen to follow LIFO(Last In First Out)/FILO(First In Last Out) order.

To implement the stack, it is required to maintain the pointer to the top of the stack, which is the last element to be inserted because we can access the elements only on the top of the stack.

LIFO (Last In First Out):

This strategy states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first.

Array Representation of Stack & Basic Operations on Stack

In order to make manipulations in a stack, there are certain operations provided to us.
push() to insert an element into the stack
pop() to remove an element from the stack
top() Returns the top element of the stack.
isEmpty() returns true if stack is empty else false.
size() returns the size of stack.

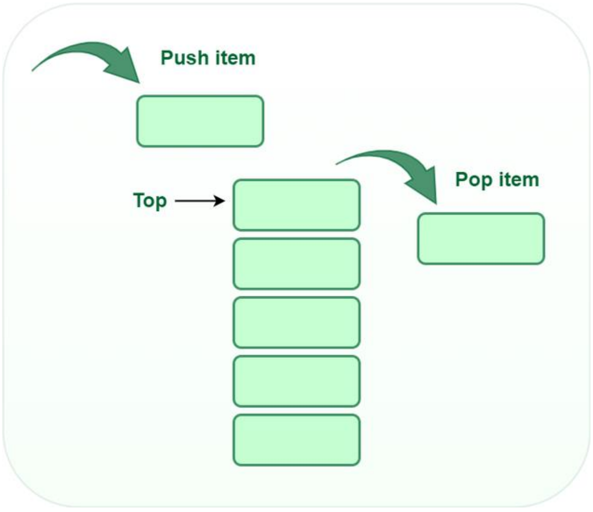


Fig: Stack

Push:
Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.
Algorithm for push:

```
begin
    if stack is full
        return
    endif
    else
    increment top
        stack[top] assign value
end else
```

Pop:
Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.
Algorithm for pop:

```
begin
    if stack is empty
        return
    endif
    else
        store value of stack[top]
        decrement top
        return value
    end else
end procedure
```

Top:
Returns the top element of the stack.
Algorithm for Top:
```
begin
        return stack[top]
end procedure
```

isEmpty:
Returns true if the stack is empty, else false.
Algorithm for isEmpty:
```
begin
         if top < 1
                 return true
         else
                 return false
end procedure
```

Code: Implementing Stack using Arrays:

```
// C program for array implementation of stack
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>

// A structure to represent a stack
struct Stack {
        int top;
        unsigned capacity;
        int* array;
};

// function to create a stack of given capacity. It initializes size of
// stack as 0
struct Stack* createStack(unsigned capacity)
{
        struct Stack* stack = (struct Stack*)malloc(sizeof(struct Stack));
        stack->capacity = capacity;
        stack->top = -1;
        stack->array = (int*)malloc(stack->capacity * sizeof(int));
        return stack;
}

// Stack is full when top is equal to the last index
int isFull(struct Stack* stack)
{
        return stack->top == stack->capacity - 1;
}

// Stack is empty when top is equal to -1
int isEmpty(struct Stack* stack)
{
	return stack->top == -1;
}

// Function to add an item to stack. It increases top by 1
void push(struct Stack* stack, int item)
{
	if (isFull(stack))
		return;
	stack->array[++stack->top] = item;
	printf("%d pushed to stack\n", item);
}

// Function to remove an item from stack. It decreases top by 1
int pop(struct Stack* stack)
{
	if (isEmpty(stack))
		return INT_MIN;
	return stack->array[stack->top--];
}

// Function to return the top from stack without removing it
int peek(struct Stack* stack)
{
	if (isEmpty(stack))
		return INT_MIN;
	return stack->array[stack->top];
}

// Driver program to test above functions
int main()
{
	struct Stack* stack = createStack(100);

	push(stack, 10);
	push(stack, 20);
	push(stack, 30);

	printf("%d popped from stack\n", pop(stack));

	return 0;
}
```

Output

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
30 Popped from stack
Top element is : 20
Elements present in stack : 20 10
```

Advantages of array implementation:

- Easy to implement.
- Memory is saved as pointers are not involved.

Disadvantages of array implementation:

- It is not dynamic i.e., it doesn't grow and shrink depending on needs at runtime. [But in case of dynamic sized arrays like vector in C++, list in Python, ArrayList in Java, stacks can grow and shrink with array implementation as well].
- The total size of the stack must be defined beforehand.

Applications of Stack:

Stack is a simple linear data structure used for storing data. Stack follows the LIFO(Last In First Out) strategy that states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first. It can be implemented through an array or linked lists. Some of its main operations are: **push(), pop(), top(), isEmpty(), size(), etc.** In order to make manipulations in a stack, there are certain operations provided to us. When we want to insert an element into the stack the operation is known as the push operation whereas when we want to remove an element from the stack the operation is known as the pop operation. If we try to pop from an empty stack then it is known as underflow and if we try to push an element in a stack that is already full, then it is known as overflow.

Application of Stack Data Structure:

- **Function calls and recursion:** When a function is called, the current state of the program is pushed onto the stack. When the function returns, the state is popped from the stack to resume the previous function's execution.
- **Undo/Redo operations:** The undo-redo feature in various applications uses stacks to keep track of the previous actions. Each time an action is performed, it is pushed onto the stack. To undo the action, the top element of the stack is popped, and the reverse operation is performed.
- **Expression evaluation:** Stack data structure is used to evaluate expressions in infix, postfix, and prefix notations. Operators and operands are pushed onto the stack, and operations are performed based on the stack's top elements.
- **Browser history:** Web browsers use stacks to keep track of the web pages you visit. Each time you visit a new page, the URL is pushed onto the stack, and when you hit the back button, the previous URL is popped from the stack.
- **Balanced Parentheses:** Stack data structure is used to check if parentheses are balanced or not. An opening parenthesis is pushed onto the stack, and a closing parenthesis is popped from the stack. If the stack is empty at the end of the expression, the parentheses are balanced.
- **Backtracking Algorithms:** The backtracking algorithm uses stacks to keep track of the states of the problem-solving process. The current state is pushed onto the stack, and when the algorithm backtracks, the previous state is popped from the stack.

Application of Stack in real life:

- CD/DVD stand.
- Stack of books in a book shop.
- Call center systems.
- Undo and Redo mechanism in text editors.
- The history of a web browser is stored in the form of a stack.
- Call logs, E-mails, and Google photos in any gallery are also stored in form of a stack.

- YouTube downloads and Notifications are also shown in LIFO format(the latest appears first).
- Allocation of memory by an operating system while executing a process.

Advantages of Stack:

- **Easy implementation:** Stack data structure is easy to implement using arrays or linked lists, and its operations are simple to understand and implement.
- **Efficient memory utilization**: Stack uses a contiguous block of memory, making it more efficient in memory utilization as compared to other data structures.
- **Fast access time:** Stack data structure provides fast access time for adding and removing elements as the elements are added and removed from the top of the stack.
- **Helps in function calls:** Stack data structure is used to store function calls and their states, which helps in the efficient implementation of recursive function calls.
- **Supports backtracking:** Stack data structure supports backtracking algorithms, which are used in problem-solving to explore all possible solutions by storing the previous states.
- **Used in Compiler Design:** Stack data structure is used in compiler design for parsing and syntax analysis of programming languages.
- **Enables undo/redo operations**: Stack data structure is used to enable undo and redo operations in various applications like text editors, graphic design tools, and software development environments.

Disadvantages of Stack:

- **Limited capacity:** Stack data structure has a limited capacity as it can only hold a fixed number of elements. If the stack becomes full, adding new elements may result in stack overflow, leading to the loss of data.
- **No random access:** Stack data structure does not allow for random access to its elements, and it only allows for adding and removing elements from the top of the stack. To access an element in the middle of the stack, all the elements above it must be removed.
- **Memory management:** Stack data structure uses a contiguous block of memory, which can result in memory fragmentation if elements are added and removed frequently.
- **Not suitable for certain applications:** Stack data structure is not suitable for applications that require accessing elements in the middle of the stack, like searching or sorting algorithms.
- **Stack overflow and underflow**: Stack data structure can result in stack overflow if too many elements are pushed onto the stack, and it can result in stack underflow if too many elements are popped from the stack.
- **Recursive function calls limitations:** While stack data structure supports recursive function calls, too many recursive function calls can lead to stack overflow, resulting in the termination of the program.

Expression Evaluation

Evaluate an expression represented by a String. The expression can contain parentheses, you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, *, and /. Arithmetic Expressions can be written in one of three forms:

- ***Infix Notation:*** Operators are written between the operands they operate on, e.g. 3 + 4.
- ***Prefix Notation:*** Operators are written before the operands, e.g + 3 4
- ***Postfix Notation:*** Operators are written after operands.

Infix Expressions are harder for Computers to evaluate because of the additional work needed to decide precedence. Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Given that they are harder to evaluate, they are generally converted to one of the two remaining forms. A very well known algorithm for converting an infix notation to a postfix notation is Shunting Yard Algorithm by Edgar Dijkstra.

This algorithm takes as input an Infix Expression and produces a queue that has this expression converted to postfix notation. The same algorithm can be modified so that it outputs the result of the evaluation of expression instead of a queue. The trick is using two stacks instead of one, one for operands, and one for operators.

```
1. While there are still tokens to be read in,
    1.1 Get the next token.
    1.2 If the token is:
        1.2.1 A number: push it onto the value stack.
        1.2.2 A variable: get its value, and push onto the value stack.
        1.2.3 A left parenthesis: push it onto the operator stack.
        1.2.4 A right parenthesis:
            1 While the thing on top of the operator stack is not a left parenthesis,
                1 Pop the operator from the operator stack.
                2 Pop the value stack twice, getting two operands.
                3 Apply the operator to the operands, in the correct order.
                4 Push the result onto the value stack.
            2 Pop the left parenthesis from the operator stack, and discard it.
        1.2.5 An operator (call it thisOp):
            1 While the operator stack is not empty, and the top thing on the
              operator stack has the same or greater precedence as thisOp,
                1 Pop the operator from the operator stack.
                2 Pop the value stack twice, getting two operands.
                3 Apply the operator to the operands, in the correct order.
                4 Push the result onto the value stack.
            2 Push thisOp onto the operator stack.
2. While the operator stack is not empty,
    1 Pop the operator from the operator stack.
    2 Pop the value stack twice, getting two operands.
    3 Apply the operator to the operands, in the correct order.
    4 Push the result onto the value stack.
3. At this point the operator stack should be empty, and the value stack should have only one value in
it, which is the final result.
```

Infix expression: The expression of the form “a operator b” (a + b) i.e., when an operator is in-between every pair of operands.
Postfix expression: The expression of the form “a b operator” (ab+) i.e., When every pair of operands is followed by an operator.

Examples:

Input: *A + B * C + D*
Output: *ABC*+D+*

Input: *((A + B) – C * (D / E)) + F*
Output: *AB+CDE/*-F+*

Why postfix representation of the expression?

The compiler scans the expression either from left to right or from right to left. Consider the expression: **a + b * c + d**

- The compiler first scans the expression to evaluate the expression b * c, then again scans the expression to add a to it.

- The result is then added to d after another scan.

The repeated scanning makes it very inefficient. Infix expressions are easily readable and solvable by humans whereas the computer cannot differentiate the operators and parenthesis easily so, it is better to convert the expression to postfix(or prefix) form before evaluation.

The corresponding expression in postfix form is **abc*+d+**. The postfix expressions can be evaluated easily using a stack.

How to convert an Infix expression to a Postfix expression?

*To convert infix expression to postfix expression, use the **stack data structure**. Scan the infix expression from left to right. Whenever we get an operand, add it to the postfix expression and if we get an operator or parenthesis add it to the stack by maintaining their precedence.*

Below are the steps to implement the above idea:

1. Scan the infix expression **from left to right**.
2. If the scanned character is an operand, put it in the postfix expression.
3. Otherwise, do the following
    - If the precedence and associativity of the scanned operator are greater than the precedence and associativity of the operator in the stack [or the stack is empty or the stack contains a '**(**'], then push it in the stack. ['^' operator is right associative and other operators like '+','–','*' and '/' are left-associative].
        - Check especially for a condition when the operator at the top of the stack and the scanned operator both are '^'. In this condition, the precedence of the scanned operator is higher due to its right associativity. So it will be pushed into the operator stack.
        - In all the other cases when the top of the operator stack is the same as the scanned operator, then pop the operator from the stack because of left associativity due to which the scanned operator has less precedence.
    - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator.
        - After doing that Push the scanned operator to the stack. (If you encounter parenthesis while popping then stop there and push the scanned operator in the stack.)
4. If the scanned character is a '**(**', push it to the stack.
5. If the scanned character is a '**)**', pop the stack and output it until a '**(**' is encountered, and discard both the parenthesis.
6. Repeat steps **2-5** until the infix expression is scanned.
7. Once the scanning is over, Pop the stack and add the operators in the postfix expression until it is not empty.
8. Finally, print the postfix expression.

Illustration:

Follow the below illustration for a better understanding

*Consider the infix expression **exp = "a+b*c+d"***
*and the infix expression is scanned using the iterator **i**, which is initialized as **i = 0**.*

***1st Step:** Here i = 0 and exp[i] = 'a' i.e., an operand. So add this in the postfix expression. Therefore, **postfix = "a"**.*

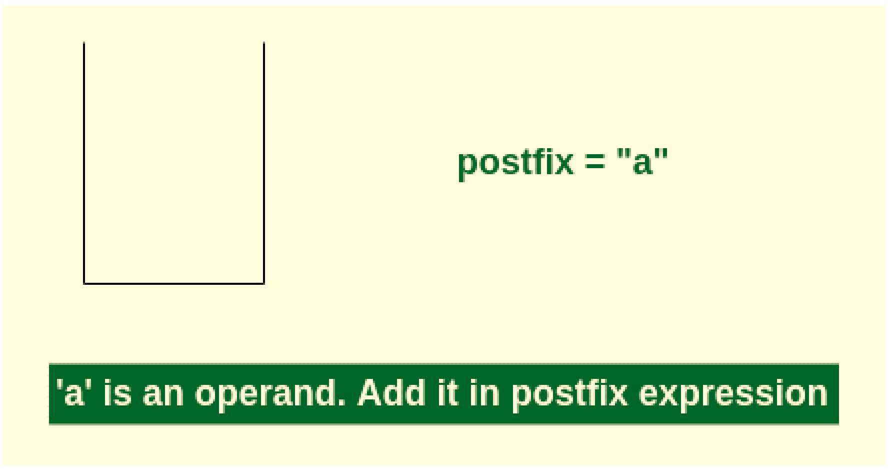


Fig: Add 'a' in the postfix

***2nd Step:** Here i = 1 and exp[i] = '+' i.e., an operator. Push this into the stack. **postfix = "a"** and **stack** = {+}.*

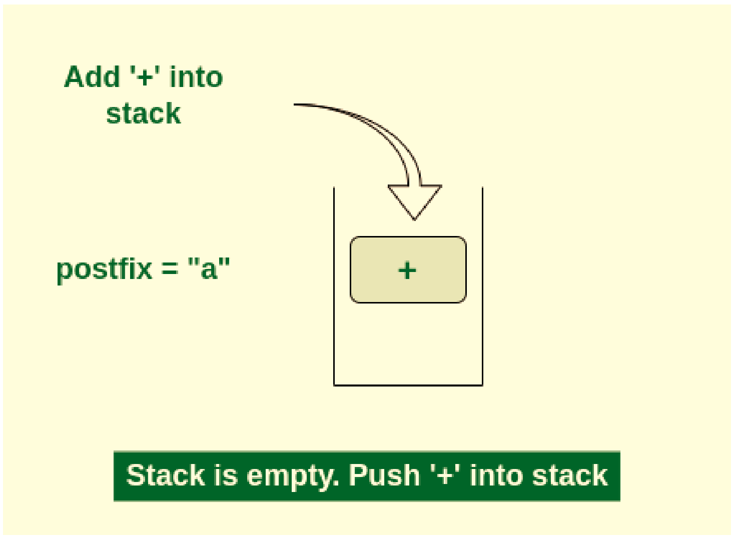


Fig: Push '+' in the stack

***3rd Step:** Now i = 2 and exp[i] = 'b' i.e., an operand. So add this in the postfix expression. **postfix** = **"ab"** and **stack** = {+}.*

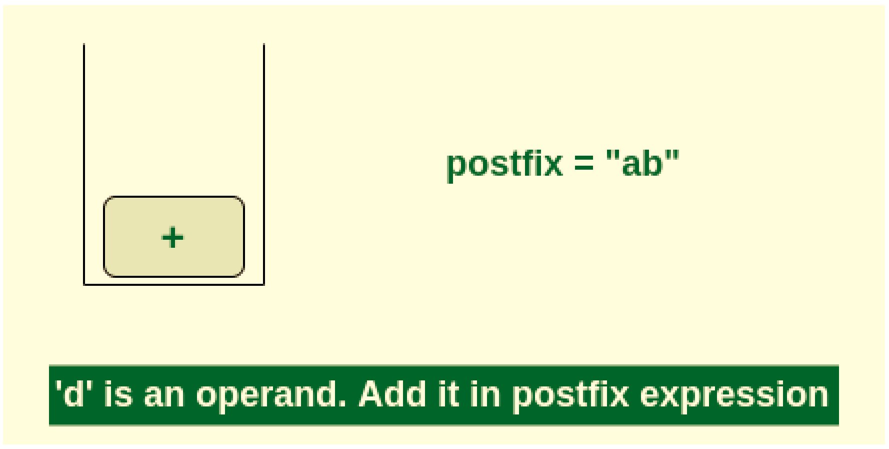


Fig: Add 'b' in the postfix

4th Step: *Now i = 3 and exp[i] = '*' i.e., an operator. Push this into the stack.* ***postfix = "ab"*** *and* ***stack*** *= {+, *}.*

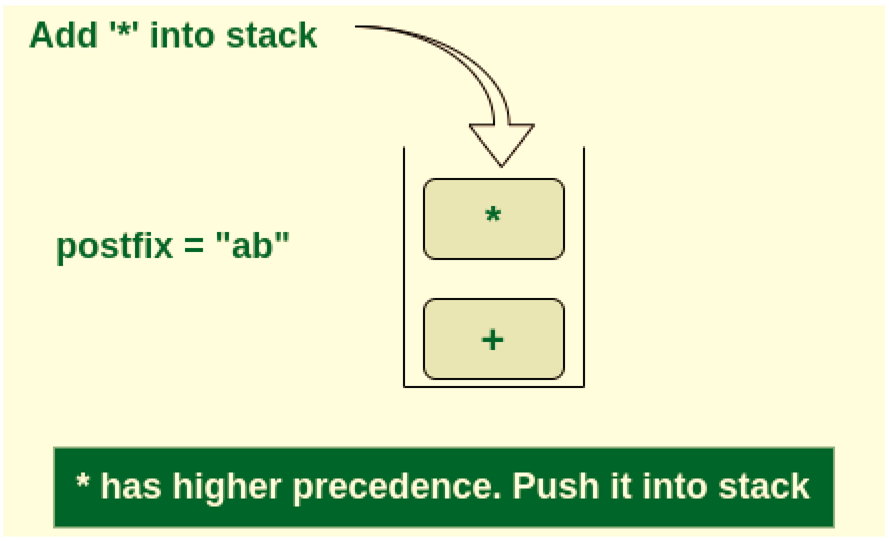


Fig: Push '' in the stack*

5th Step: *Now i = 4 and exp[i] = 'c' i.e., an operand. Add this in the postfix expression.* ***postfix = "abc"*** *and* ***stack*** *= {+, *}.*

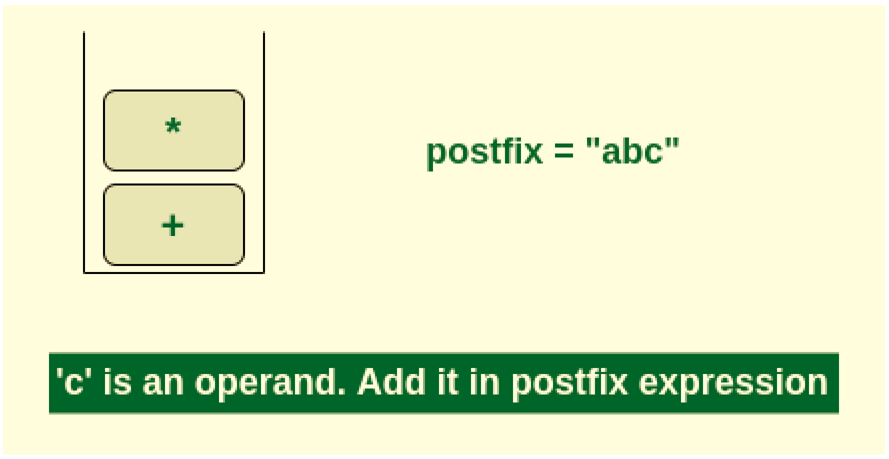


Fig: Add 'c' in the postfix

6th Step: *Now i = 5 and exp[i] = '+' i.e., an operator. The topmost element of the stack has higher precedence. So pop until the stack becomes empty or the top element has less precedence. '*' is popped and added in postfix. So* ***postfix*** *=* ***"abc*"*** *and* ***stack*** *= {+}.*

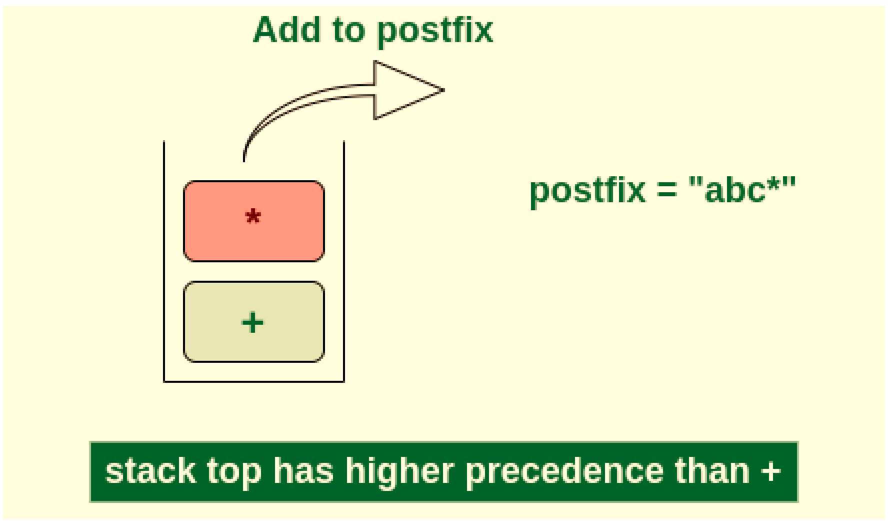


Fig: Pop '' and add in postfix*

Now top element is '+' that also doesn't have less precedence. Pop it. ***postfix*** *=* ***"abc*+"****.*

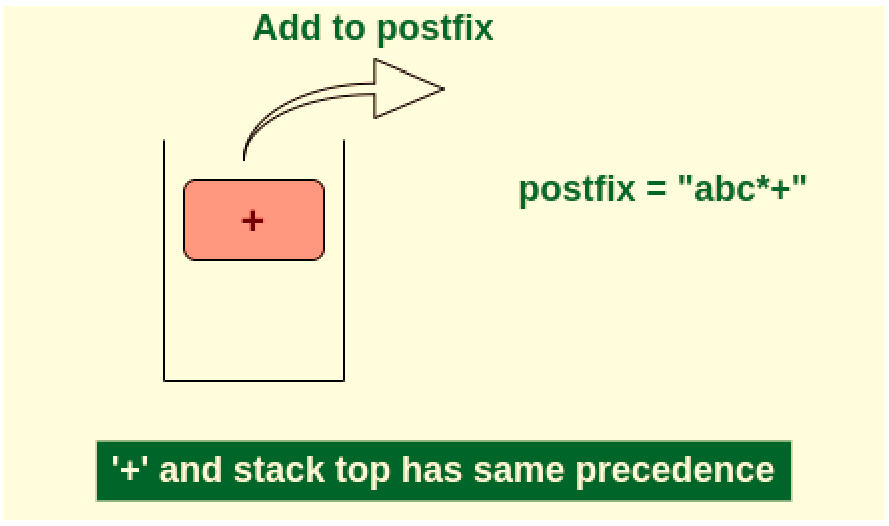


Fig: Pop '+' and add it in postfix

Now stack is empty. So push ***'+'*** *in the stack.* ***stack*** *= {+}.*

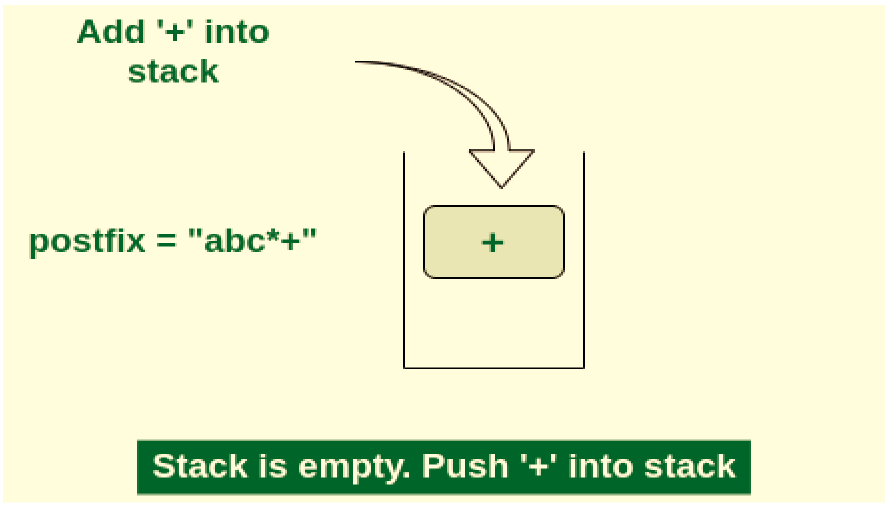


Fig: Push '+' in the stack

***7th Step:** Now i = 6 and exp[i] = 'd' i.e., an operand. Add this in the postfix expression. **postfix = "abc*+d"**.*

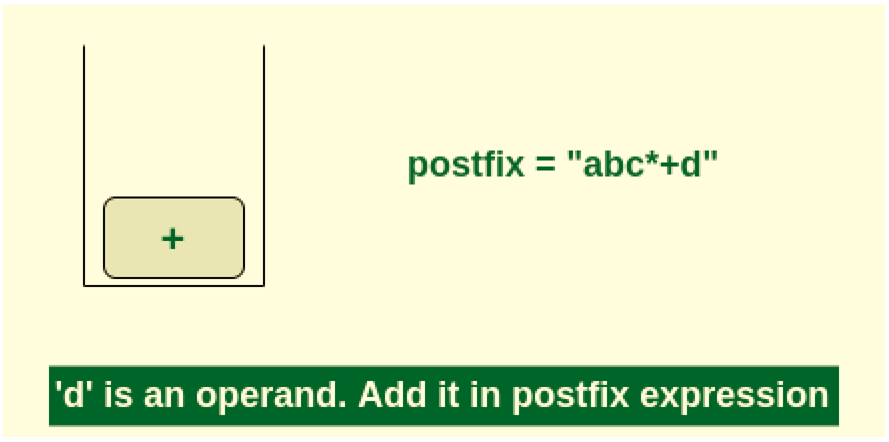


Fig: Add 'd' in the postfix

***Final Step:** Now no element is left. So empty the stack and add it in the postfix expression. **postfix = "abc*+d+"**.*

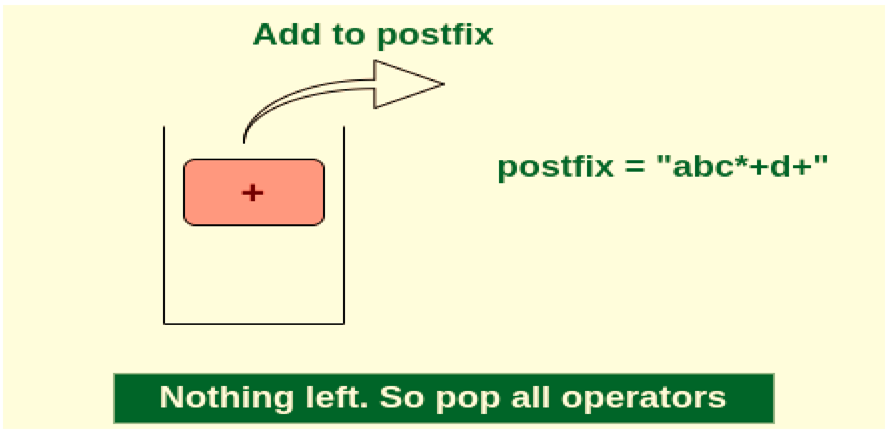


Fig: Pop '+' and add it in postfix

Below is the implementation of the above algorithm:

```
// C code to convert infix to postfix expression

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

#define MAX_EXPR_SIZE 100

// Function to return precedence of operators
int precedence(char operator)
{
	switch (operator) {
	case '+':
	case '-':
		return 1;
	case '*':
	case '/':
		return 2;
	case '^':
		return 3;
	default:
		return -1;
	}
}

// Function to check if the scanned character
// is an operator
int isOperator(char ch)
{
	return (ch == '+' || ch == '-' || ch == '*' || ch == '/'
			|| ch == '^');
}

// Main functio to convert infix expression
// to postfix expression
char* infixToPostfix(char* infix)
{
	int i, j;
	int len = strlen(infix);
	char* postfix = (char*)malloc(sizeof(char) * (len + 2));
	char stack[MAX_EXPR_SIZE];
	int top = -1;

	for (i = 0, j = 0; i < len; i++) {
		if (infix[i] == ' ' || infix[i] == '\t')
			continue;

		// If the scanned character is operand
		// add it to the postfix expression
		if (isalnum(infix[i])) {
				postfix[j++] = infix[i];
			}

			// if the scanned character is '('
			// push it in the stack
			else if (infix[i] == '(') {
				stack[++top] = infix[i];
			}

			// if the scanned character is ')'
			// pop the stack and add it to the
			// output string until empty or '(' found
			else if (infix[i] == ')') {
				while (top > -1 && stack[top] != '(')
					postfix[j++] = stack[top--];
				if (top > -1 && stack[top] != '(')
					return "Invalid Expression";
				else
					top--;
			}

			// If the scanned character is an operator
			// push it in the stack
			else if (isOperator(infix[i])) {
				while (top > -1
					&& precedence(stack[top])
							>= precedence(infix[i]))
					postfix[j++] = stack[top--];
				stack[++top] = infix[i];
			}
		}

		// Pop all remaining elements from the stack
		while (top > -1) {
			if (stack[top] == '(') {
				return "Invalid Expression";
			}
			postfix[j++] = stack[top--];
		}
		postfix[j] = '\0';
		return postfix;
}

// Driver code
int main()
{
		char infix[MAX_EXPR_SIZE] = "a+b*(c^d-e)^(f+g*h)-i";

		// Function call
		char* postfix = infixToPostfix(infix);
		printf("%s\n", postfix);
```

```
	free(postfix);
	return 0;
}
```

Output
abcd^e-fgh*+^*+i-
Time Complexity: O(N), where N is the size of the infix expression
Auxiliary Space: O(N), where N is the size of the infix expression

Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods (A, B, and C) and N disks. Initially, all the disks are stacked in decreasing value of diameter i.e., the smallest disk is placed on the top and they are on rod A. The objective of the puzzle is to move the entire stack to another rod (here considered C), obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

Rules of Tower of Hanoi Puzzle
The Tower of Hanoi problem is solved using the set of rules given below:

- Only one disc can be moved at a time.
- Only the top disc of one stack can be transferred to the top of another stack or an empty rod.
- Larger discs cannot be stacked over smaller ones.
- The complexity of this problem can be mapped by evaluating the number of possible moves. The least movements needed to solve the Tower of Hanoi problem with n discs are 2n-1.

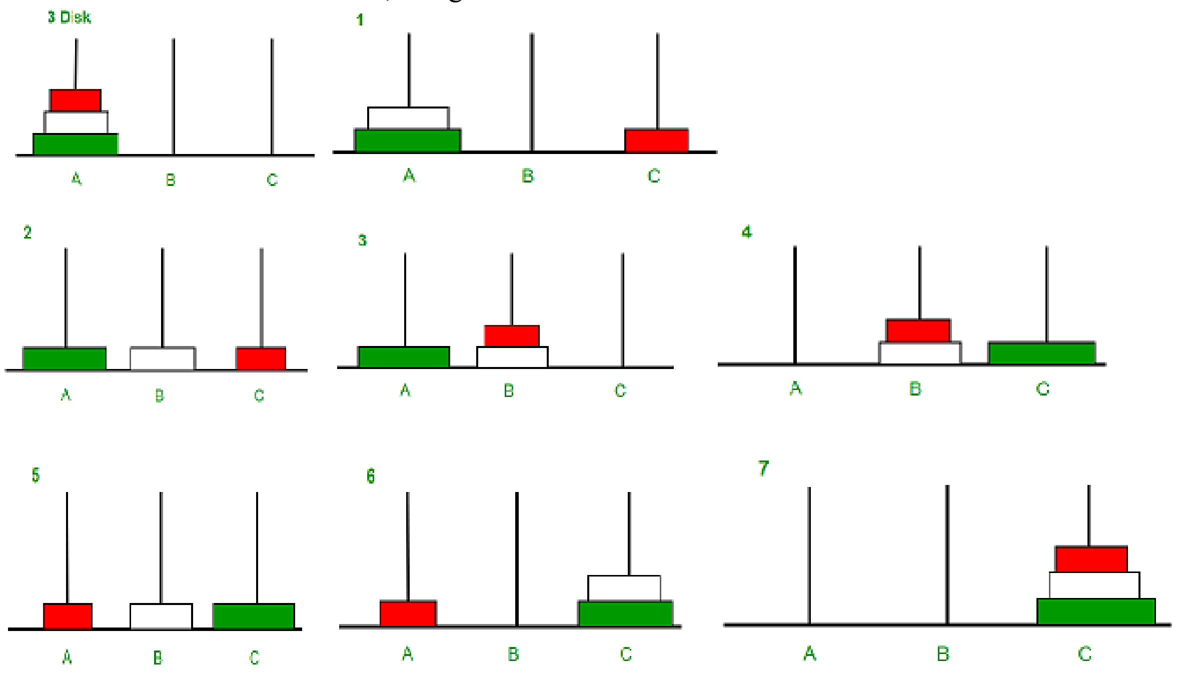
Examples:

Input: 2
Output: Disk 1 moved from A to B
Disk 2 moved from A to C
Disk 1 moved from B to C

Input: 3
Output: Disk 1 moved from A to C
Disk 2 moved from A to B
Disk 1 moved from C to B
Disk 3 moved from A to C
Disk 1 moved from B to A
Disk 2 moved from B to C
Disk 1 moved from A to C
Tower of Hanoi using Recursion:

The idea is to use the helper node to reach the destination using recursion. Below is the pattern for this problem:

Shift ‘N-1’ disks from ‘A’ to ‘B’, using C.
Shift last disk from ‘A’ to ‘C’.
Shift ‘N-1’ disks from ‘B’ to ‘C’, using A.

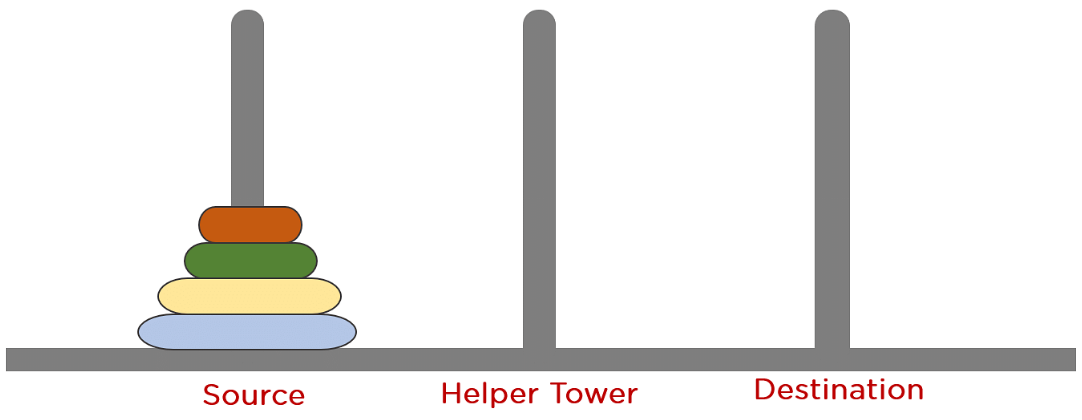


Follow the steps below to solve the problem:

- Create a function towerOfHanoi where pass the N (current number of disk), from_rod, to_rod, aux_rod.
- Make a function call for N – 1 th disk.
- Then print the current the disk along with from_rod and to_rod
- Again make a function call for N – 1 th disk.

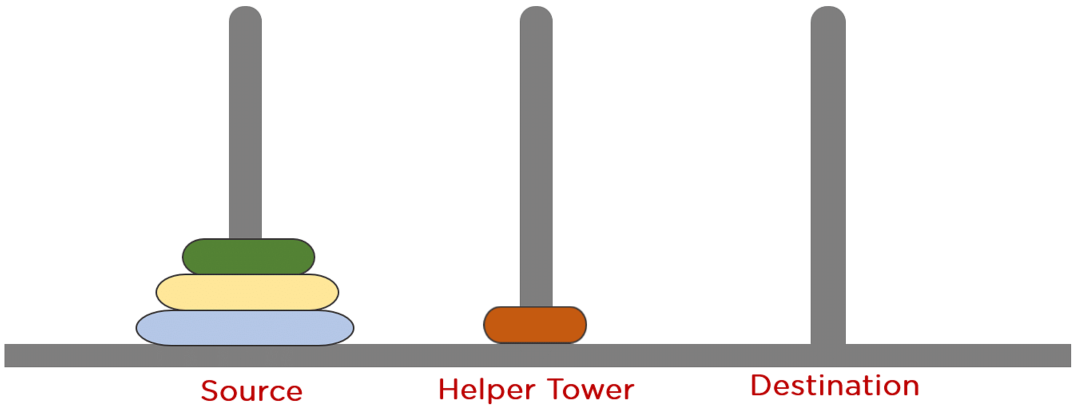
Logical Approach to Implement Solution For TOH Problem

In this section, you will implement a logical solution to the four-ring TOH problem. The initial problem setup will be as represented in the image below.



In order to solve this problem, you are going to utilize an auxiliary or Helper tower. The first move you will make will be to transfer the orange ring to the helper tower.

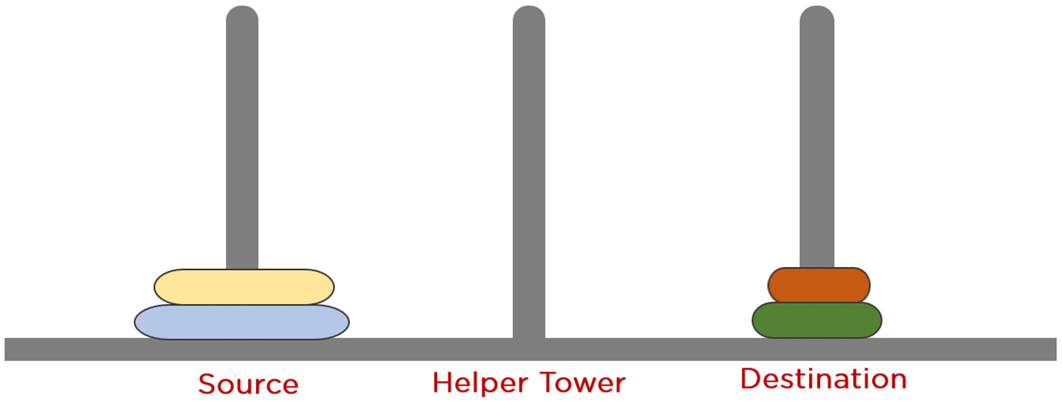
Move 1



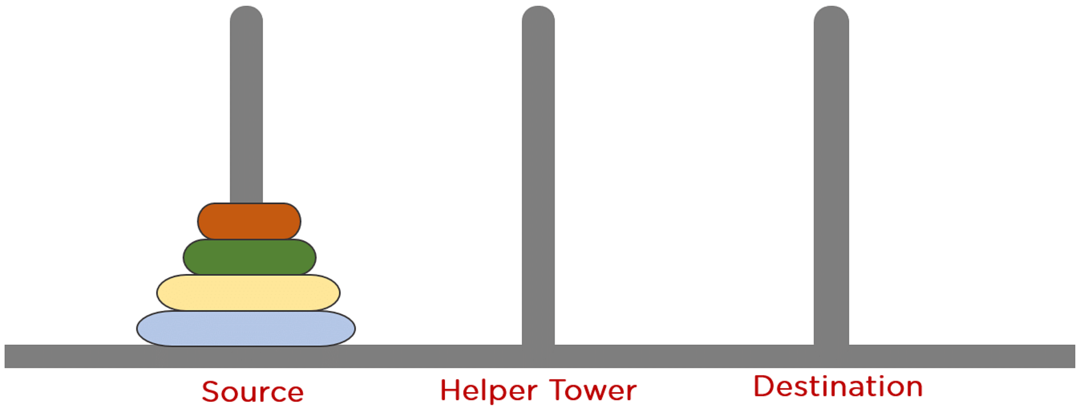
Next, you can move the green disc to the destination tower.

After that, you can move the orange disc to the destination tower. You will place this orange disc above the green disc.

Move 3

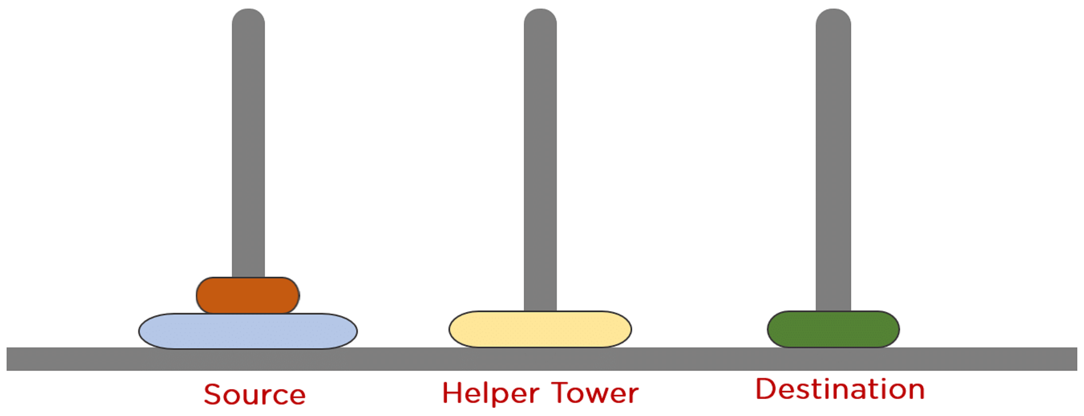


Now, the helper tower is empty. So, you will take out the yellow ring from the source tower to the helper tower.



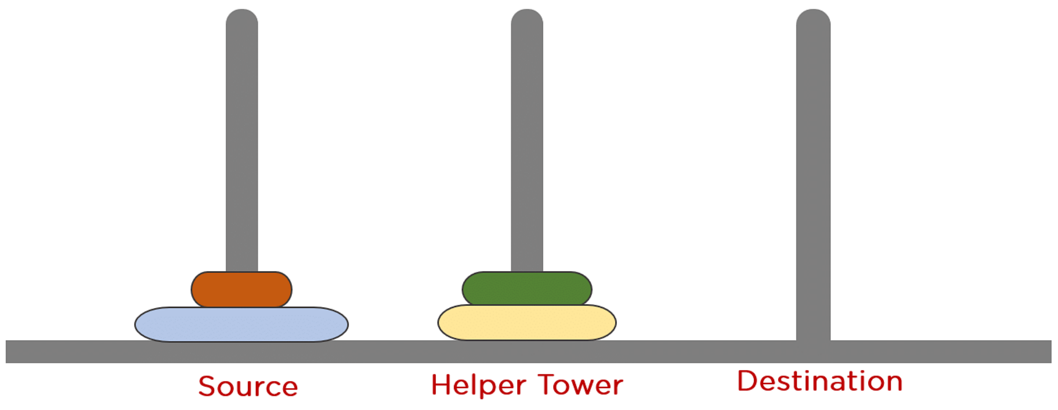
Next, you will move the orange ring to the source tower.

Move 5



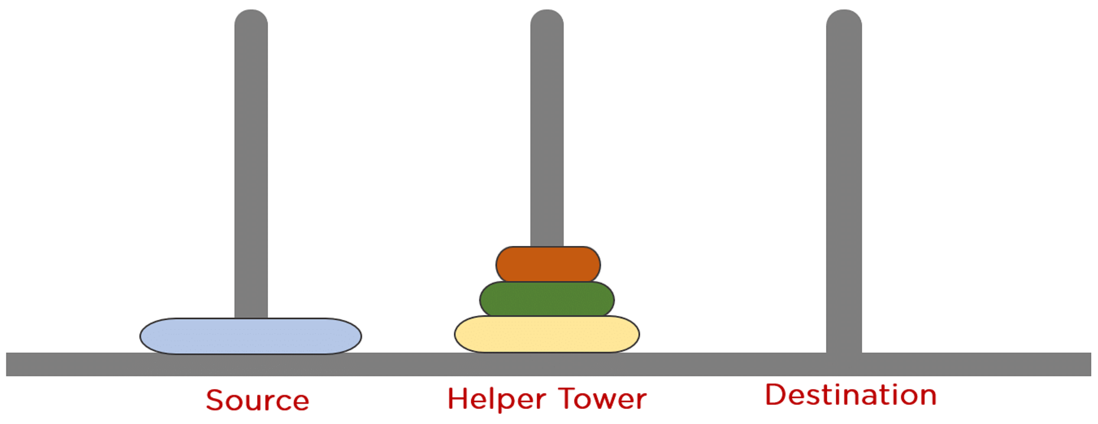
Further, you will move the green ring to the helper tower.

Move 6



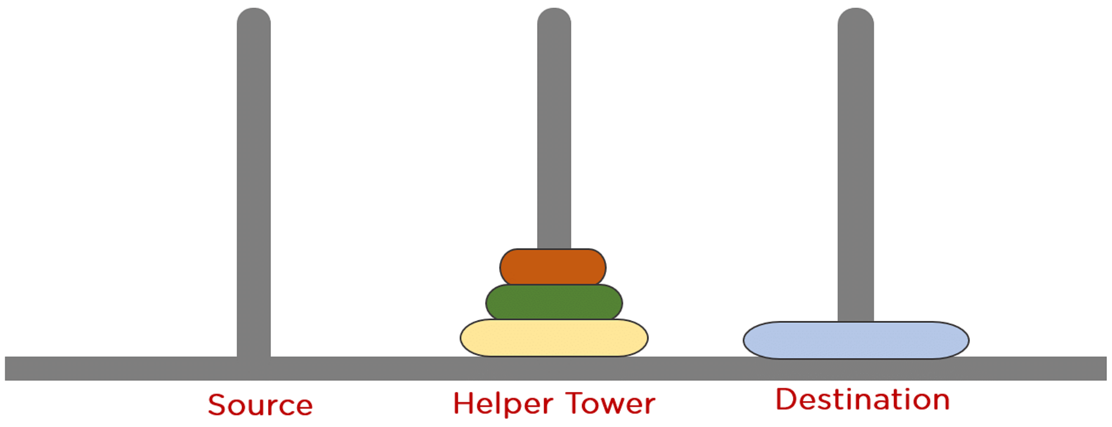
Next, you will move the orange ring to the helper tower. This move will make room for the transfer of the largest disc to the destination tower.

Move 7



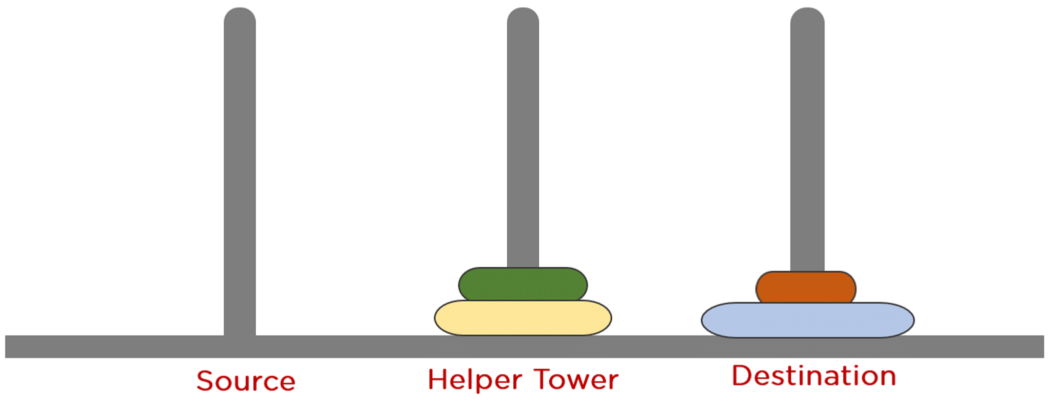
Here, you can bring the largest disc to the destination tower. However, you still need to bring three more discs to our destination tower in order to solve this TOH problem.

Move 8



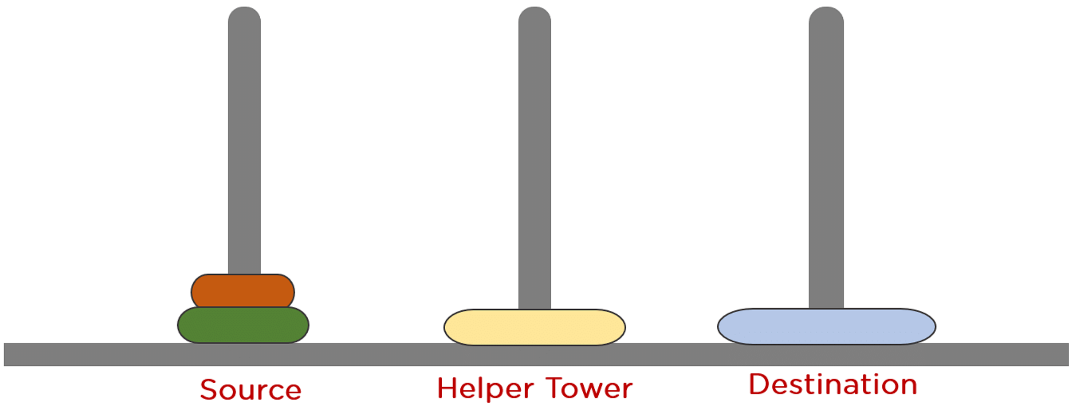
Here, you will move the orange ring to the destination tower.

Move 9



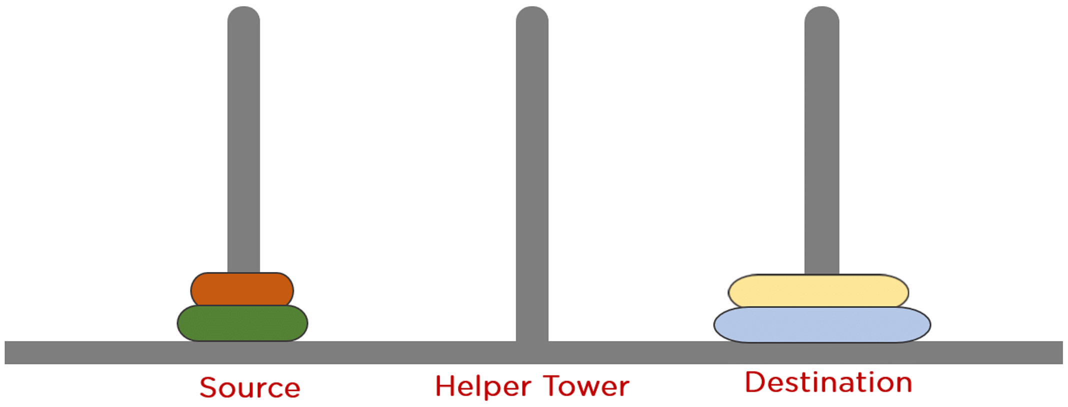
After that, you will move the green ring to the source tower. And you will also transfer the orange ring to the source tower. These two moves will allow you to transfer the second largest disc to the destination tower.

Move 10 & 11



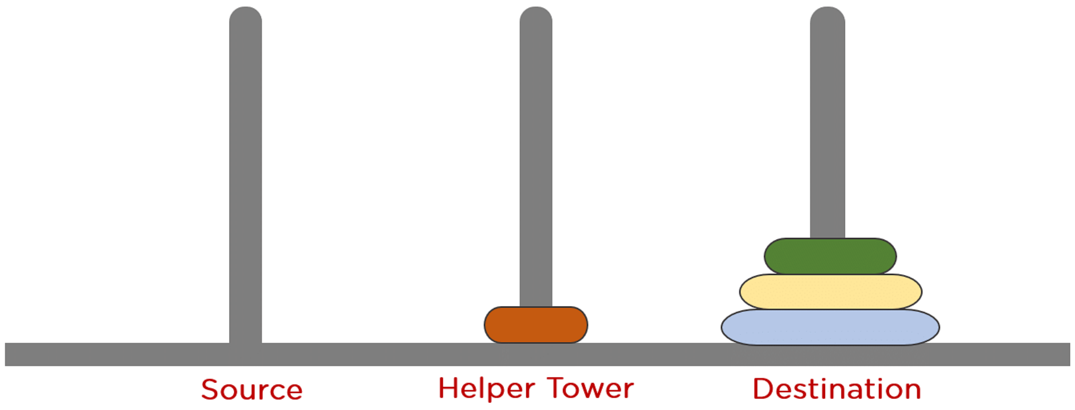
Here, you will place the second ring at the destination tower. Now, you are only left with the arrangement of two more rings.

Move 12



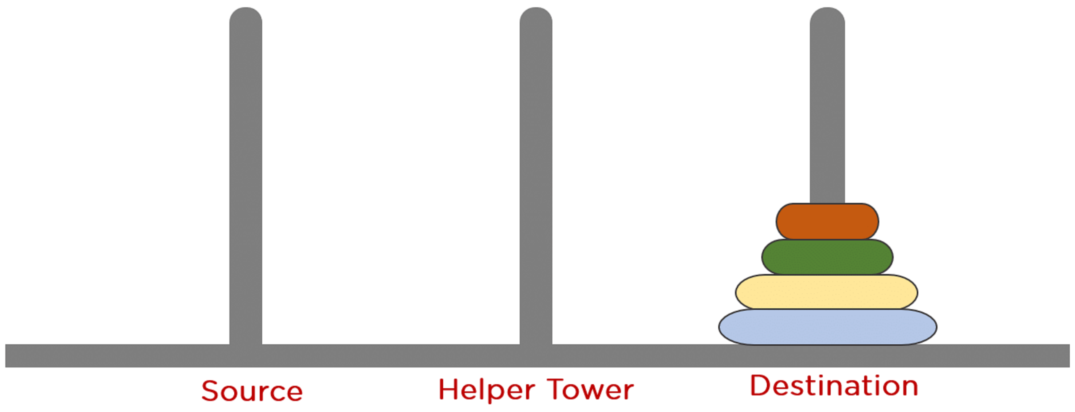
You will move the orange ring to the helper tower and the green ring to the destination tower.

Move 13 and 14



Finally, you will move the orange disc to the destination tower.

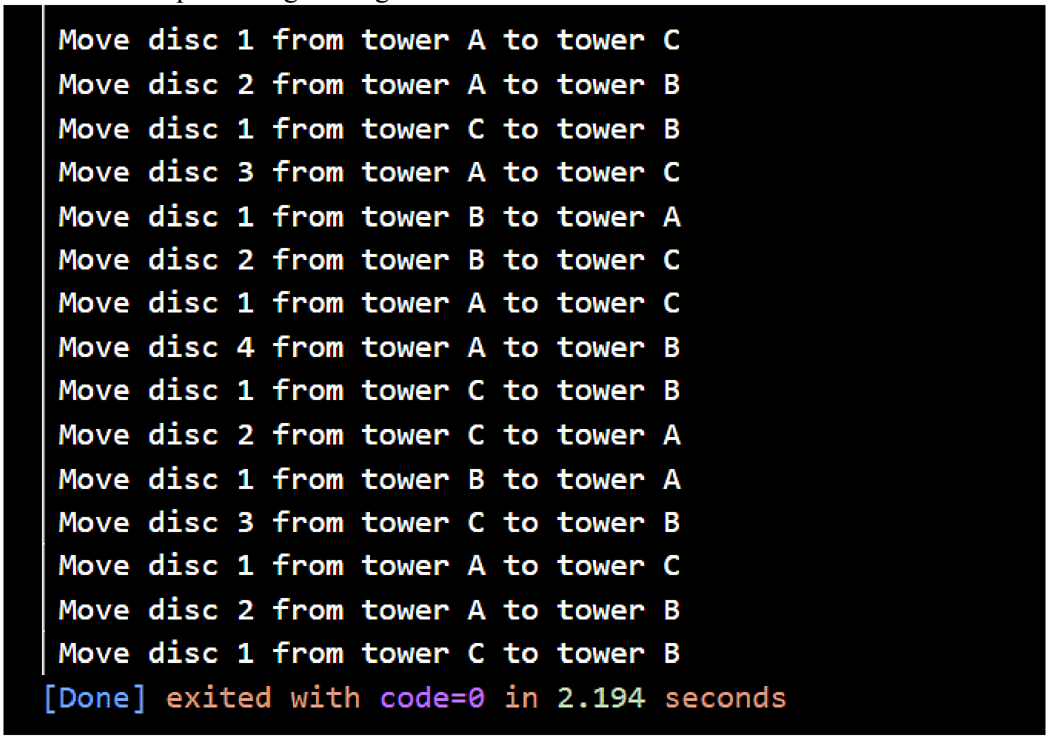
Move 15



Coding Implementation of TOH Solution
The TOH puzzle can be solved using a recursive programming paradigm. The C program for building a solution to the four-ring TOH problem is given below.

```
#include<stdio.h>
#include<conio.h>
void TOH(int n, char source, char destination, char helper_t){
   if(n==0){
      return 0;
   }
   TOH(n-1,source, helper_t,destination);
   printf("\n Move disc %d from tower %c to tower %c",n, source, destination);
   TOH(n-1, helper_t, destination,source);
}
int main()
{
   TOH(4, 'A','B','C');
   return 0;
}
```

Output:
The console representing 15 ring movements to reach the final solution.

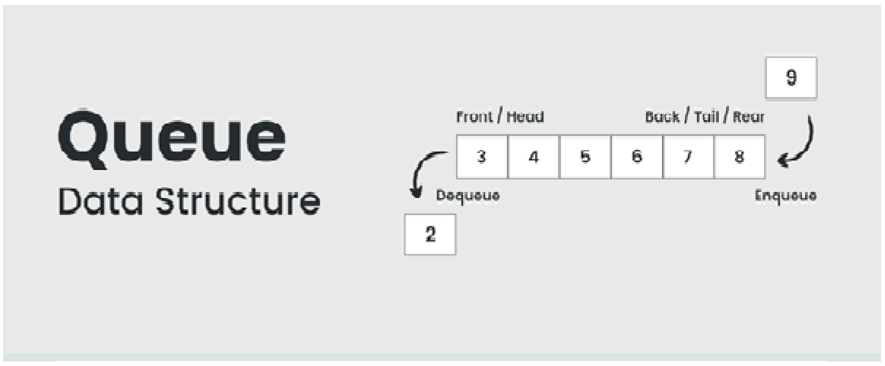


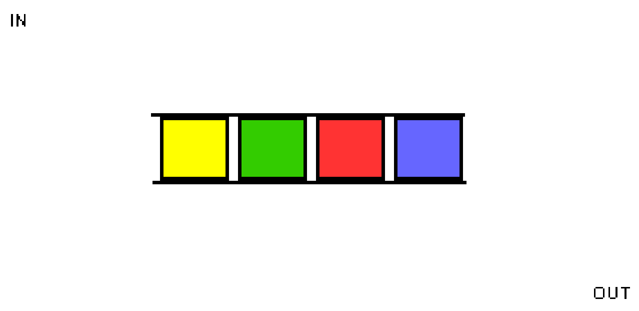
```
 Move disc 1 from tower A to tower C
 Move disc 2 from tower A to tower B
 Move disc 1 from tower C to tower B
 Move disc 3 from tower A to tower C
 Move disc 1 from tower B to tower A
 Move disc 2 from tower B to tower C
 Move disc 1 from tower A to tower C
 Move disc 4 from tower A to tower B
 Move disc 1 from tower C to tower B
 Move disc 2 from tower C to tower A
 Move disc 1 from tower B to tower A
 Move disc 3 from tower C to tower B
 Move disc 1 from tower A to tower C
 Move disc 2 from tower A to tower B
 Move disc 1 from tower C to tower B
[Done] exited with code=0 in 2.194 seconds
```

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.

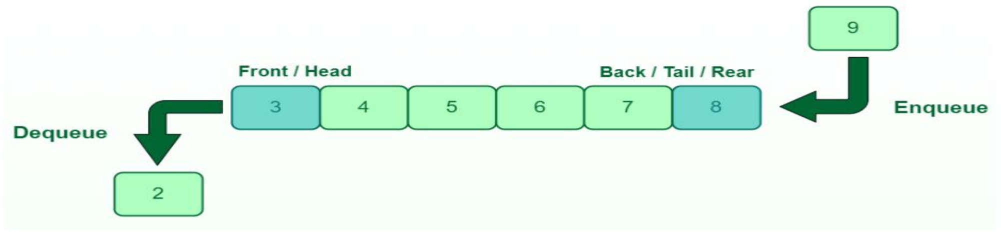
We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.





FIFO Principle of Queue:

- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).
- Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **front** of the queue(sometimes, **head** of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the **rear** (or the **tail**) of the queue. See the below figure.



Queue Data Structure

Characteristics of Queue:

- Queue can handle multiple data.
- We can access both ends.
- They are fast and flexible.

Queue Representation:

Like stacks, Queues can also be represented in an array: In this representation, the Queue is implemented using the array. Variables used in this case are

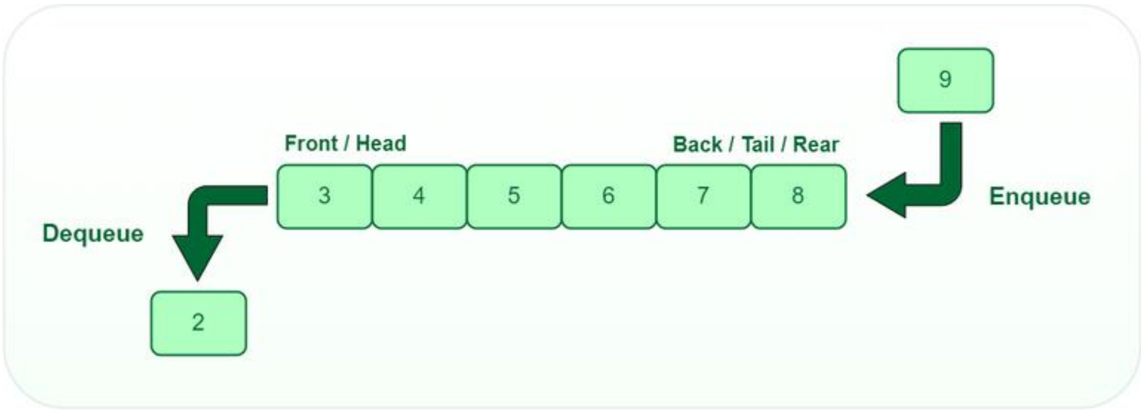
- **Queue:** the name of the array storing queue elements.

- **Front**: the index where the first element is stored in the array representing the queue.
- **Rear:** the index where the last element is stored in an array representing the queue.

Basic Operations on Queue:

Some of the basic operations for Queue in Data Structure are:

- **enqueue() –** Insertion of elements to the queue.
- **dequeue() –** Removal of elements from the queue.
- **peek() or front()-** Acquires the data element available at the front node of the queue without deleting it.
- **rear() –** This operation returns the element at the rear end without removing it.
- **isFull() –** Validates if the queue is full.
- **isEmpty() –** Checks if the queue is empty.
- **size():** This operation returns the size of the queue i.e. the total number of elements it contains.

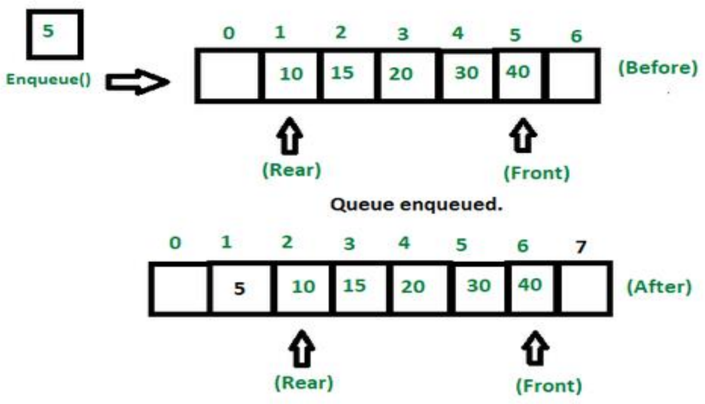


Operation 1: enqueue()

Inserts an element at the end of the queue i.e. at the rear end.

The following steps should be taken to enqueue (insert) data into a queue:

- Check if the queue is full.
- If the queue is full, return overflow error and exit.
- If the queue is not full, increment the rear pointer to point to the next empty space.
- Add the data element to the queue location, where the rear is pointing.
- return success.

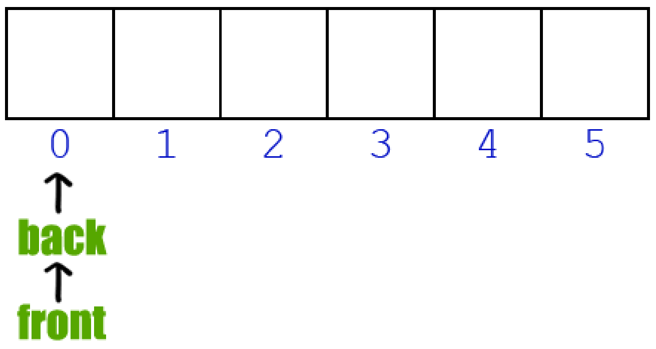


Operation 2: dequeue()

This operation removes and returns an element that is at the front end of the queue.

The following steps are taken to perform the dequeue operation:

- Check if the queue is empty.
- If the queue is empty, return the underflow error and exit.
- If the queue is not empty, access the data where the front is pointing.
- Increment the front pointer to point to the next available data element.
- The Return success.

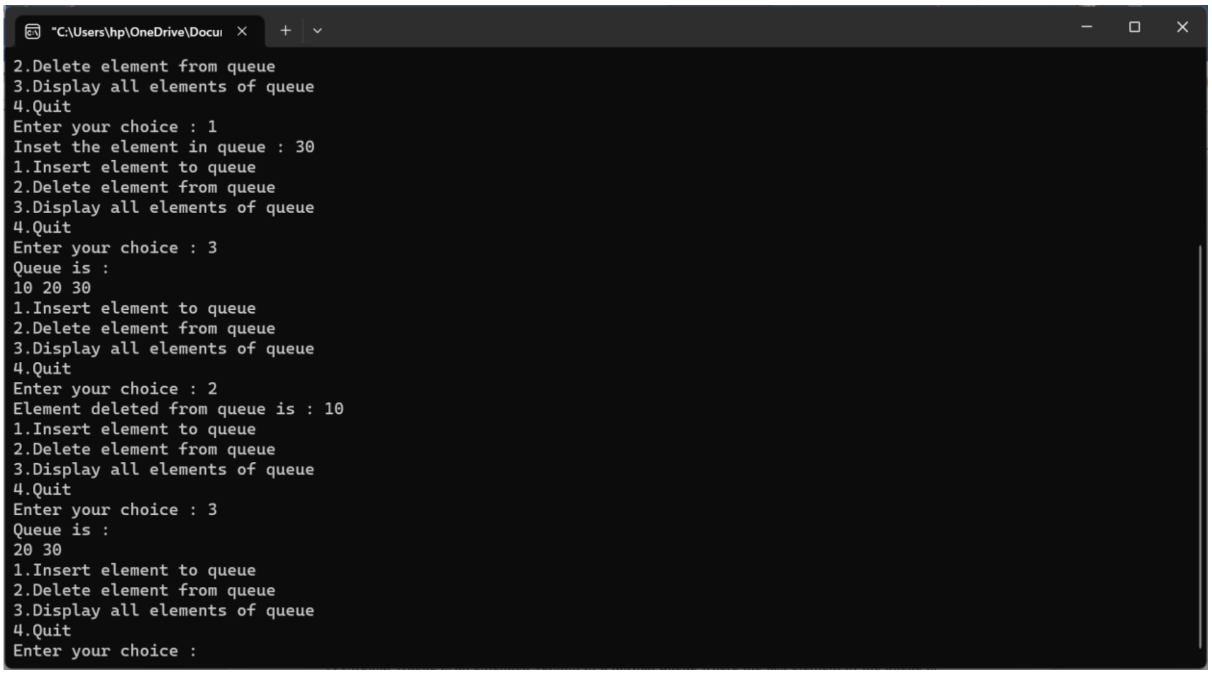


Array Implementation of Queue

```
#include <stdio.h>
#include <cstdlib>
#define MAX 50

void insert();
void del();
void display();
int queue_array[MAX];
int rear = - 1;
int front = - 1;
main()
{
    int choice;
    while (1)
    {
        printf("1.Insert element to queue \n");
        printf("2.Delete element from queue \n");
        printf("3.Display all elements of queue \n");
        printf("4.Quit \n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1:
            insert();
            break;
            case 2:
            del();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            default:
```

```
            printf("Wrong choice \n");
        } /* End of switch */
    } /* End of while */
} /* End of main() */

void insert()
{
    int add_item;
    if (rear == MAX - 1)
    printf("Queue Overflow \n");
    else
    {
        if (front == - 1)
        /*If queue is initially empty */
        front = 0;
        printf("Inset the element in queue : ");
        scanf("%d", &add_item);
        rear = rear + 1;
        queue_array[rear] = add_item;
    }
} /* End of insert() */

void del()
{
    if (front == - 1 || front > rear)
    {
        printf("Queue Underflow \n");
        return ;
    }
    else
    {
        printf("Element deleted from queue is : %d\n", queue_array[front]);
        front = front + 1;
    }
} /* End of delete() */

void display()
{
    int i;
    if (front == - 1)
        printf("Queue is empty \n");
    else
    {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", queue_array[i]);
        printf("\n");
    }
} /* End of display() */
```

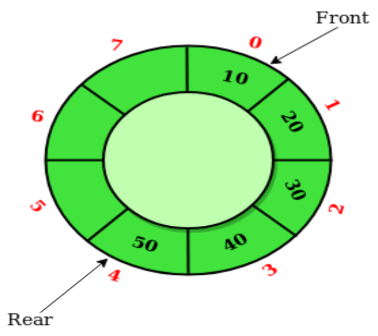
OUTPUT



```
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 1
Inset the element in queue : 30
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
10 20 30
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 2
Element deleted from queue is : 10
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice : 3
Queue is :
20 30
1.Insert element to queue
2.Delete element from queue
3.Display all elements of queue
4.Quit
Enter your choice :
```

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'**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we cannot insert the next element even if there is a space in front of queue.

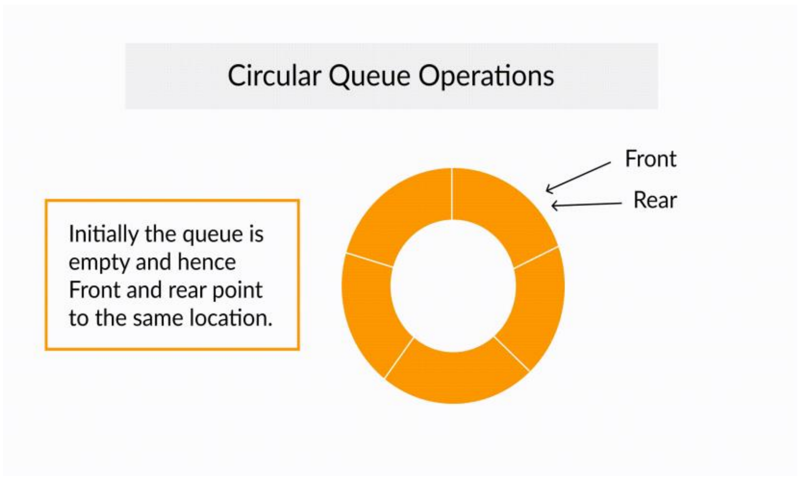
Operations on Circular Queue:

- **Front:** Get the front item from the queue.
- **Rear:** Get the last item from the queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at the rear position.
  - Check whether the queue is full – [i.e., the rear end is in just before the front end in a circular manner].
  - If it is full then display Queue is full.
  - If the queue is not full then, insert an element at the end of the queue.
- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.
  - Check whether the queue is Empty.
  - If it is empty then display Queue is empty.

- If the queue is not empty, then get the last element and remove it from the queue.

Implement Circular Queue using Array:

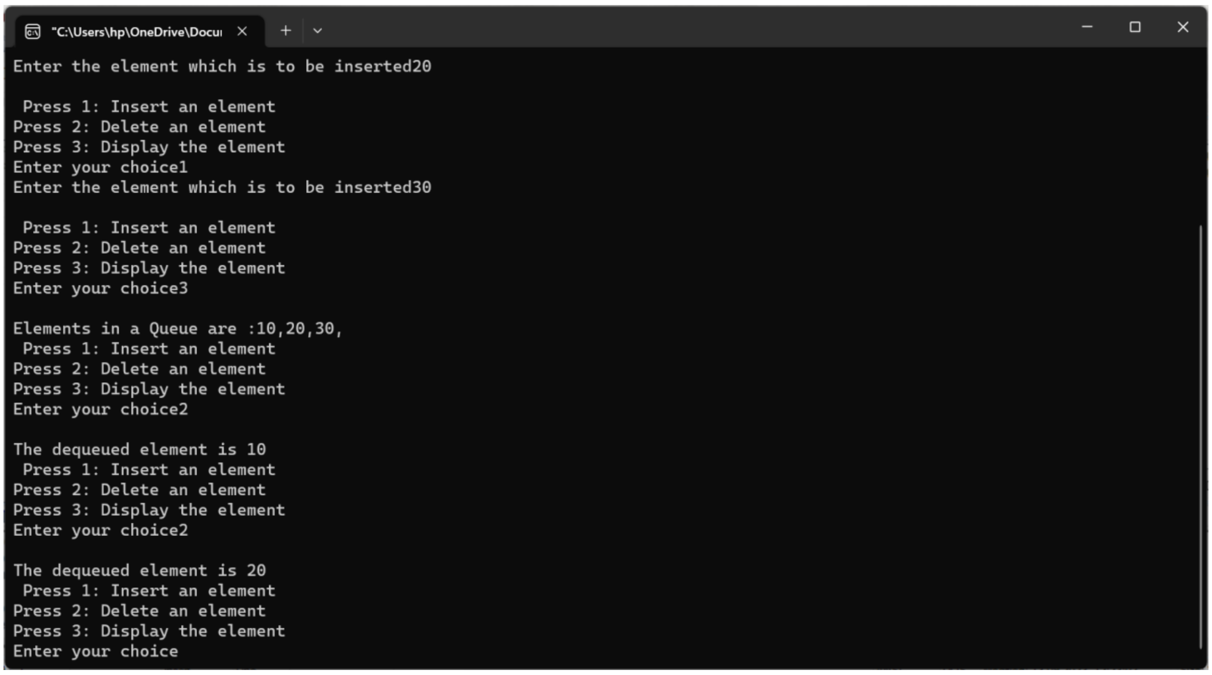
1. Initialize an array queue of size **n**, where n is the maximum number of elements that the queue can hold.
2. Initialize two variables front and rear to -1.
3. **Enqueue:** To enqueue an element **x** into the queue, do the following:
   - Increment rear by 1.
   - If **rear** is equal to n, set **rear** to 0.
   - If **front** is -1, set **front** to 0.
   - Set queue[rear] to x.
4. **Dequeue:** To dequeue an element from the queue, do the following:
   - Check if the queue is empty by checking if **front** is -1.
   - If it is, return an error message indicating that the queue is empty.
   - Set **x** to queue[front].
   - If **front** is equal to **rear**, set **front** and **rear** to -1.
   - Otherwise, increment **front** by 1 and if **front** is equal to n, set **front** to 0.
   - Return x.


Circular Queue Operations
Front
Rear
Initially the queue is empty and hence Front and rear point to the same location.

Circular Queue Implementation

```
#include <stdio.h>
#include <cstdlib>
# define max 6
int queue[max];  // array declaration
int front=-1;
int rear=-1;
// function to insert an element in a circular queue
void enqueue(int element)
{
   if(front==-1 && rear==-1)   // condition to check queue is empty
   {
      front=0;
      rear=0;
      queue[rear]=element;
   }
   else if((rear+1)%max==front)  // condition to check queue is full
```

```
    {
        printf("Queue is overflow..");
    }
    else
    {
        rear=(rear+1)%max;       // rear is incremented
        queue[rear]=element;     // assigning a value to the queue at the rear position.
    }
}

// function to delete the element from the queue
int dequeue()
{
    if((front==-1) && (rear==-1))  // condition to check queue is empty
    {
        printf("\nQueue is underflow..");
    }
 else if(front==rear)
{
   printf("\nThe dequeued element is %d", queue[front]);
   front=-1;
   rear=-1;
}
else
{
    printf("\nThe dequeued element is %d", queue[front]);
   front=(front+1)%max;
}
}
// function to display the elements of a queue
void display()
{
    int i=front;
    if(front==-1 && rear==-1)
    {
        printf("\n Queue is empty..");
    }
    else
    {
        printf("\nElements in a Queue are :");
        while(i<=rear)
        {
            printf("%d,", queue[i]);
            i=(i+1)%max;
        }
    }
}
int main()
{
    int choice=1,x;   // variables declaration
    while(choice<4 && choice!=0)   // while loop
    {
    printf("\nPress 1: Insert an element");
    printf("\nPress 2: Delete an element");
    printf("\nPress 3: Display the element");
    printf("\nPress 4: Quit");
    printf("\nEnter your choice");
    scanf("%d", &choice);
    switch(choice)
    {
        case 1:
        printf("Enter the element which is to be inserted");
        scanf("%d", &x);
        enqueue(x);
        break;
        case 2:
        dequeue();
        break;
        case 3:
        display();
        break;
        case 4:
        exit(0);
    }
}
    return 0;
}
```



```
Enter the element which is to be inserted20

 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice1
Enter the element which is to be inserted30

 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice3

Elements in a Queue are :10,20,30,
 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice2

The dequeued element is 10
 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice2

The dequeued element is 20
 Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
```

Applications of Circular Queue:

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.

3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

Applications of Circular Queue- Job Scheduling

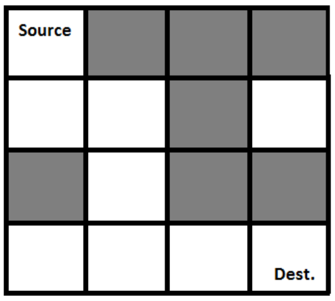
The jobs that are to be executed by the computer is scheduled to be executed one by one. There are many jobs like keyboard press, mouse click etc. in the system. These jobs are brought in the main memory. These jobs are assigned to the processor one by one which is organized using a queue.e.g. First In First Out and Round Robin processor scheduling in queues.

A Maze Problem

A Maze is given as N*N binary matrix of blocks where source block is the upper left most block i.e., maze[0][0] and destination block is lower rightmost block i.e., maze[N-1][N-1]. A rat starts from source and has to reach the destination. The rat can move only in two directions: forward and down.
In the maze matrix, 0 means the block is a dead end and 1 means the block can be used in the path from source to destination. Note that this is a simple version of the typical Maze problem. For example, a more complex version can be that the rat can move in 4 directions and a more complex version can be with a limited number of moves.

Following is an example maze.

Gray blocks are dead ends (value = 0).



Following is a binary matrix representation of the above maze.

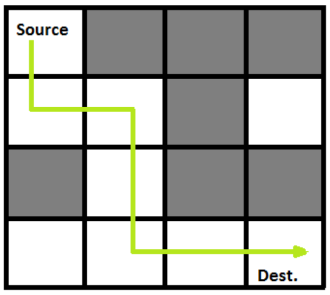
{1, 0, 0, 0}

{1, 1, 0, 1}

{0, 1, 0, 0}

{1, 1, 1, 1}

Following is a maze with highlighted solution path.



Following is the solution matrix (output of program) for the above input matrix.

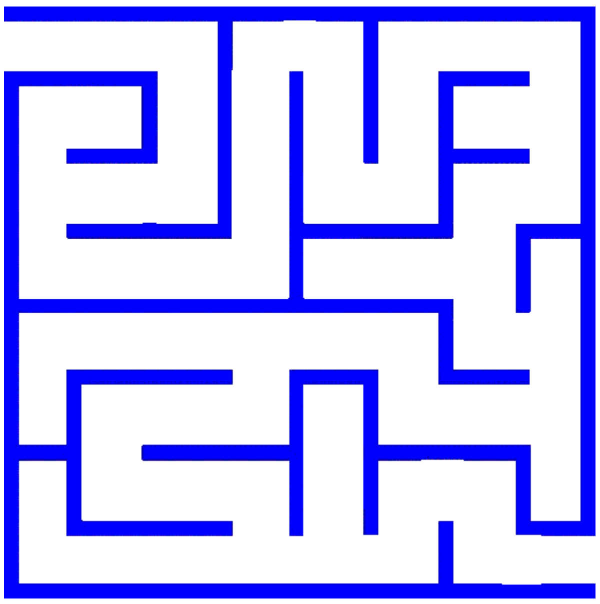
{1, 0, 0, 0}

{1, 1, 0, 0}

{0, 1, 0, 0}

{0, 1, 1, 1}

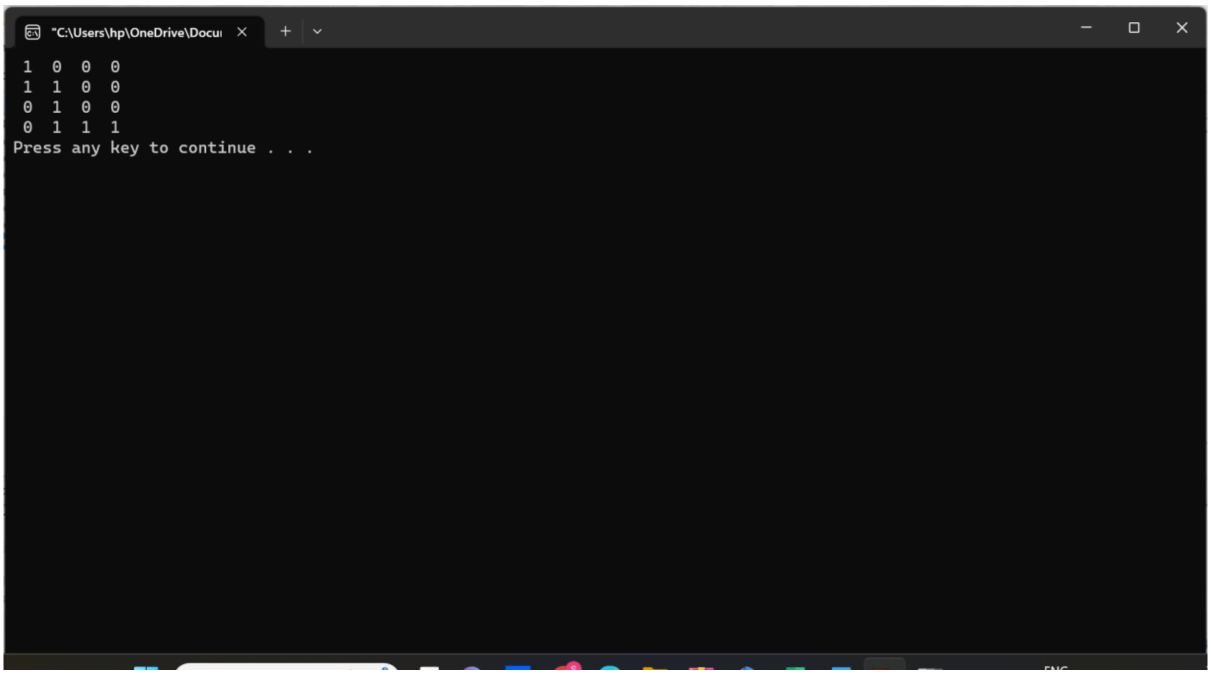
All entries in solution path are marked as 1.



```
/* C program to solve Rat in a Maze problem using backtracking */
#include <stdio.h>

// Maze size
#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N]);

/* A utility function to print solution matrix sol[N][N] */
void printSolution(int sol[N][N])
{
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++)
                        printf(" %d ", sol[i][j]);
                printf("\n");
        }
```

```
}

/* A utility function to check if x, y is valid index for N*N maze */
bool isSafe(int maze[N][N], int x, int y)
{
        // if (x, y outside maze) return false
        if (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)
                return true;

        return false;
}

/* This function solves the Maze problem using Backtracking. It mainly uses solveMazeUtil() to solve
the problem. It returns false if no path is possible, otherwise return true and prints the path in the form
of 1s. Please note that there may be more than one solutions, this function prints one of the feasible
solutions.*/
bool solveMaze(int maze[N][N])
{
        int sol[N][N] = { { 0, 0, 0, 0 },
                                        { 0, 0, 0, 0 },
                                        { 0, 0, 0, 0 },
                                        { 0, 0, 0, 0 } };

        if (solveMazeUtil(maze, 0, 0, sol) == false) {
                printf("Solution doesn't exist");
                return false;
        }

        printSolution(sol);
        return true;
}

/* A recursive utility function to solve Maze problem */
bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])
{
        // if (x, y is goal) return true
        if (x == N - 1 && y == N - 1) {
                sol[x][y] = 1;
                return true;
        }

        // Check if maze[x][y] is valid
        if (isSafe(maze, x, y) == true) {
                // mark x, y as part of solution path
                sol[x][y] = 1;

                /* Move forward in x direction */
                if (solveMazeUtil(maze, x + 1, y, sol) == true)
                        return true;

                /* If moving in x direction doesn't give solution then
                Move down in y direction */
                if (solveMazeUtil(maze, x, y + 1, sol) == true)
                        return true;
/* If none of the above movements work then BACKTRACK: unmark x, y as part of solution path */
                sol[x][y] = 0;
                return false;
        }

        return false;
}

// driver program to test above function
int main()
{
        int maze[N][N] = { { 1, 0, 0, 0 },
                                { 1, 1, 0, 1 },
                                { 0, 1, 0, 0 },
                                { 1, 1, 1, 1 } };

        solveMaze(maze);
        return 0;
}
```

OUTPUT



```
 1  0  0  0
 1  1  0  0
 0  1  0  0
 0  1  1  1
Press any key to continue . . .
```