Internal Assessment Test 1 – June 2025

Data Structures and Algorithms							Sub Code:	MMC203
01/07/2025	Duration:	90 min's	Max Marks:	50	Sem:	II	Branch:	MCA

PART I

1. Define data structure. Explain the classification of data structure with examples

A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways. In computer programming, a data structure may be selected or designed to store data for the purpose of working on it with various algorithms

CLASSIFICATION OF DATA STRUCTURES

Data structures are generally classified into

Primitive data Structures

Non-primitive data Structures

Primitive data Structures: Primitive data structures are the fundamental data types which are supported by a programming language. Basic data types such as integer, real, character and Boolean are known as Primitive data Structures. These data types consists of characters that cannot be divided and hence they also called simple data types.

Non- Primitive data Structures: Non-primitive data structures are those data structures which are created using primitive data structures. Examples of non-primitive data structures is the processing of complex numbers, linked lists, stacks, trees, and graphs Based on the structure and arrangement of data, non-primitive data structures is further classified into

1. Linear Data Structure

2. Non-linear Data Structure

1. Linear Data Structure:

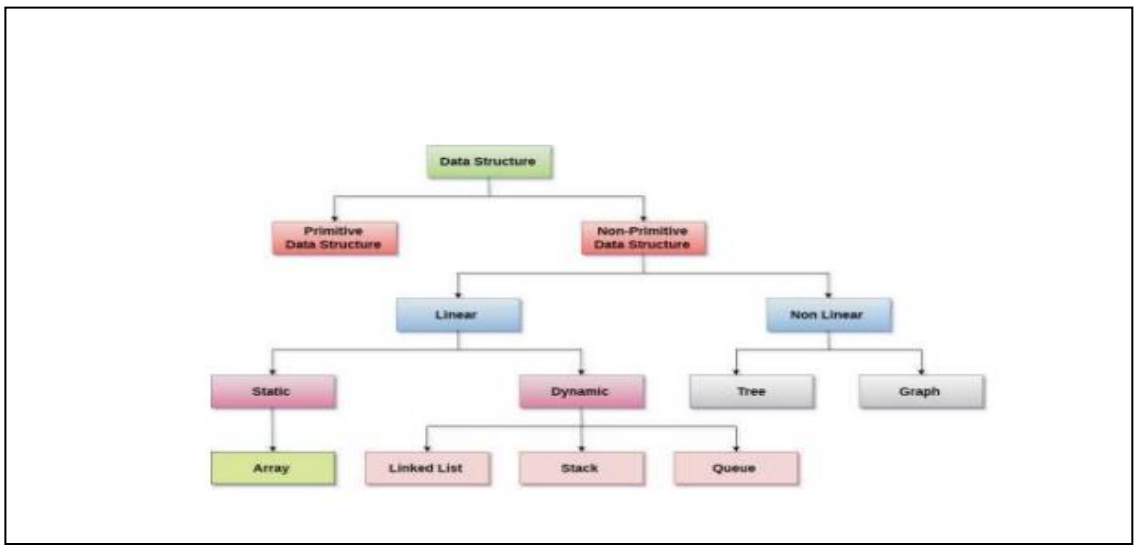
A data structure is said to be linear if its elements form a sequence or a linear list. There are basically two ways of representing such linear structure in memory.

1. One way is to have the linear relationships between the elements represented by means of sequential memory location. These linear structures are called arrays.

2. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked lists.

The common examples of linear data structure are Arrays, Queues, Stacks, Linked lists

2. Non-linear Data Structure:
A data structure is said to be non-linear if the data are not arranged in sequence or a linear. The insertion and deletion of data is not possible in linear fashion. This structure is mainly used to represent data containing a hierarchical relationship between elements. Trees and graphs are the examples of non-linear data structure.



Arrays:
The simplest type of data structure is a linear (or one dimensional) array. A list of a finite number *n* of similar data referenced respectively by a set of *n* consecutive numbers, usually 1, 2, 3 . . . . . *n*. if **A** is chosen the name for the array, then the elements of **A** are denoted by subscript notation a1, a2, a3….. an
by the bracket notation A [1], A [2], A [3] . . . . . A [n]

Trees
Data frequently contain a hierarchical relationship between various elements. The data structure which reflects this relationship is called a rooted tree graph or a tree. Some of the basic properties of tree are explained by means of examples

1. Stack: A stack, also called a fast-in first-out (LIFO) system, is a linear list in which insertions and deletions can take place only at one end, called the top. This structure is similar in its operation to a stack of dishes on a spring system as shown in fig.

Note that new 4 dishes are inserted only at the top of the stack and dishes can be deleted only from the top of the Stack

Queue: A queue, also called a first-in first-out (FIFO) system, is a linear list in which deletions can take place only at one end of the list, the "from'' of the list, and insertions can take place only at the other end of the list, the “rear” of the list.

This structure operates in much the same way as a line of people waiting at a bus stop, as pictured in Fig. the first person in line is the first person to board the bus. Another analogy is with automobiles waiting to pass through an intersection the first car in line is the first car through.

Graph: Data sometimes contain a relationship between pairs of elements which is not necessarily hierarchical in nature. For example, suppose an airline flies only between the cities connected by lines in Fig. The data structure which reflects this type of relationship is called a graph.

DATA STRUCTURES OPERATIONS

The data appearing in data structures are processed by means of certain operations. The following four operations play a major role in this text:

1. **Traversing:** accessing each record/node exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called “**visiting**” the record.)

2. **Searching:** Finding the location of the desired node with a given key value, or finding the locations of all such nodes which satisfy one or more conditions.

3. **Inserting:** Adding a new node/record to the structure.

4. **Deleting:** Removing a node/record from the structure.

The following two operations, which are used in special situations:

1. **Sorting:** Arranging the records in some logical order (e.g., alphabetically according to some NAME key, or in numerical order according to some NUMBER key, such as social security number or account number)

Merging: Combining the records in two different sorted files into a single sorted file.

2. **Write a c program to demonstrate the following operations on a singly linked list a) Insert node at beginning b) Insert node at end c) delete at a given position.**

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

// Define structure for node
struct Node {
    int data;
    struct Node* next;
};

// Head pointer to the list
struct Node* head = NULL;

// Function to insert node at the beginning
void insertAtBeginning(int value) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
    head = newNode;
    printf("Node inserted at beginning.\n");
}

// Function to insert node at the end
void insertAtEnd(int value) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
    } else {
        struct Node* temp = head;
        while (temp->next != NULL)
            temp = temp->next;
        temp->next = newNode;
    }
    printf("Node inserted at end.\n");
}

// Function to delete node at a given position
void deleteAtPosition(int position) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
```

```
    if (position == 1) {
        head = temp->next;
        free(temp);
        printf("Node deleted at position 1.\n");
        return;
    }

    for (int i = 1; temp != NULL && i < position - 1; i++) {
        temp = temp->next;
    }

    if (temp == NULL || temp->next == NULL) {
        printf("Position out of range.\n");
        return;
    }

    struct Node* nextNode = temp->next->next;
    free(temp->next);
    temp->next = nextNode;
    printf("Node deleted at position %d.\n", position);
}

// Function to display the linked list
void displayList() {
    struct Node* temp = head;
    if (temp == NULL) {
        printf("List is empty.\n");
        return;
    }
    printf("Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

// Main function with menu
int main() {
    int choice, value, position;
    while (1) {
        printf("\n--- Menu ---\n");
        printf("1. Insert at beginning\n");
        printf("2. Insert at end\n");
        printf("3. Delete at position\n");
        printf("4. Display list\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
       scanf("%d", &choice);

        switch (choice) {
```

```
            case 1:
                printf("Enter value to insert at beginning: ");
                scanf("%d", &value);
                insertAtBeginning(value);
                break;
            case 2:
                printf("Enter value to insert at end: ");
                scanf("%d", &value);
                insertAtEnd(value);
                break;
            case 3:
                printf("Enter position to delete: ");
                scanf("%d", &position);
                deleteAtPosition(position);
                break;
            case 4:
                displayList();
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice.\n");
        }
    }
}
```

Sample Output:

```
--- Menu ---
1. Insert at beginning
2. Insert at end
3. Delete at position
4. Display list
5. Exit
Enter your choice: 1
Enter value to insert at beginning: 10
Node inserted at beginning.

--- Menu ---
1. Insert at beginning
2. Insert at end
3. Delete at position
4. Display list
5. Exit
Enter your choice: 2
Enter value to insert at end: 20
Node inserted at end.

--- Menu ---
1. Insert at beginning
2. Insert at end
```

3. Delete at position
4. Display list
5. Exit
Enter your choice: 4
Linked List: 10 -> 20 -> NULL

--- Menu ---
1. Insert at beginning
2. Insert at end
3. Delete at position
4. Display list
5. Exit
Enter your choice: 3
Enter position to delete: 2
Node deleted at position 2.

--- Menu ---
1. Insert at beginning
2. Insert at end
3. Delete at position
4. Display list
5. Exit
Enter your choice: 4
Linked List: 10 -> NULL

--- Menu ---
1. Insert at beginning
2. Insert at end
3. Delete at position
4. Display list
5. Exit
Enter your choice: 5

=== Code Execution Successful ===

PART II

3. Define algorithm. Explain the steps involved in algorithm design and analysis process with neat diagram.

ALGORITHM

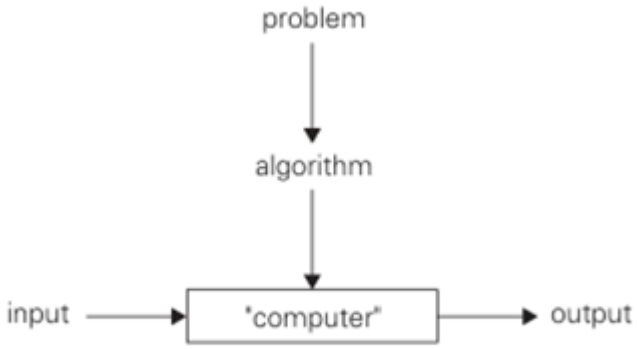
An algorithm is a well-defined computational procedure having well defined steps for solving a particular problem. Algorithm is finite set oflogic or instructions, written in order for accomplishing the certain predefined task.

Each algorithm must have:

- **Specification: Description of the computational procedure.**
- **Pre-conditions: The condition(s) on input.**
- **Body of the Algorithm: A sequence of clear and unambiguous instructions.**

✱**Post-conditions: The condition(s) on output.**

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time. This definition can be illustrated by a simple diagram

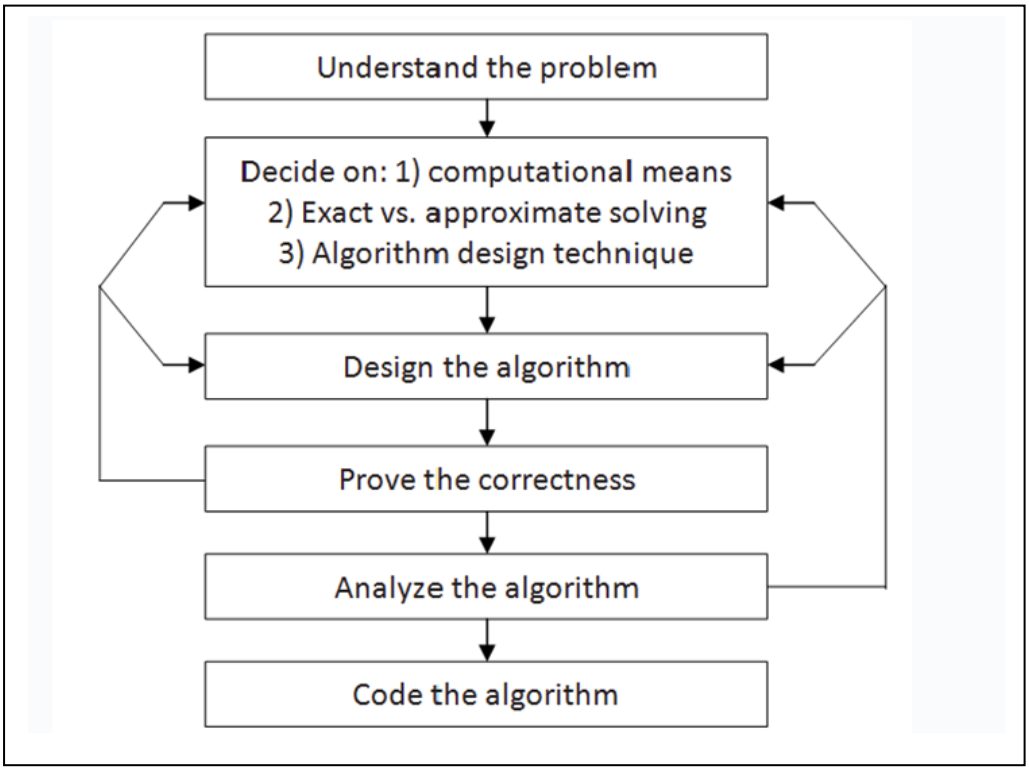


The notion of the algorithm.

- **Understanding the Problem –** From a practical perspective, the first thing you need to do before designing an algorithm is to understand completely the problem given. An input to an algorithm specifies an instance of the problem the algorithm solves. It is very important to specify exactly the set of instances the algorithm needs to handle.
  - **Ascertaining the Capabilities of the Computational Device** – Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. Select appropriate model from sequential or parallel programming model.
  - **Choosing between Exact and Approximate Problem Solving –**The next principal decision is to choose between solving the problem exactly and solving it approximately. Because, there are important problems that simply cannot be solved exactly for most of their instances and some of the available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity.
  - **Algorithm Design Techniques** – An algorithm design technique (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. They provide guidance for designing algorithms for new problems, i.e., problems for which there is no known satisfactory algorithm.
  - **Designing an Algorithm and Data Structures** – One should pay close attention to choosing data structures appropriate for the operations performed by the algorithm. For example, the sieve of Eratosthenes would run

longer if we used a linked list instead of an array in its implementation. ***Algorithms + Data Structures = Programs***

- **Methods of Specifying an Algorithm-** Once you have designed an algorithm; you need to specify it in some fashion. These are the two options that are most widely used nowadays for specifying algorithms. Using a ***natural language*** has an obvious appeal; however, the inherent ambiguity of any natural language makes a concise and clear description of algorithms surprisingly difficult. ***Pseudocode*** is a mixture of a natural language and programming language like constructs.Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions.
- **Proving an Algorithm's Correctness –** Once an algorithm has been specified, you have to prove its correctness. That is, you have to prove that the algorithm yields a required result for every legitimate input in a finite amount of time. For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex. A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
- **Analyzing an Algorithm –** After correctness, by far the most important is ***efficiency***. In fact, there are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses. Another desirable characteristic of an algorithm is ***simplicity***. Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder.
- **Coding an Algorithm –** Most algorithms are destined to be ultimately implemented as computer programs. Implementing an algorithm correctly is necessary but not sufficient: you would not like to diminish your algorithm's power by an inefficient implementation. Modern compilers do provide a certain safety net in this regard, especially when they are used in their code optimization mode.



4. Define time complexity and space complexity with examples.

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two n-bit integers takes n steps. Consequently, the total computational time is T(n) = c ∗ n, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

 A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.

 A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)

Step 1 - START

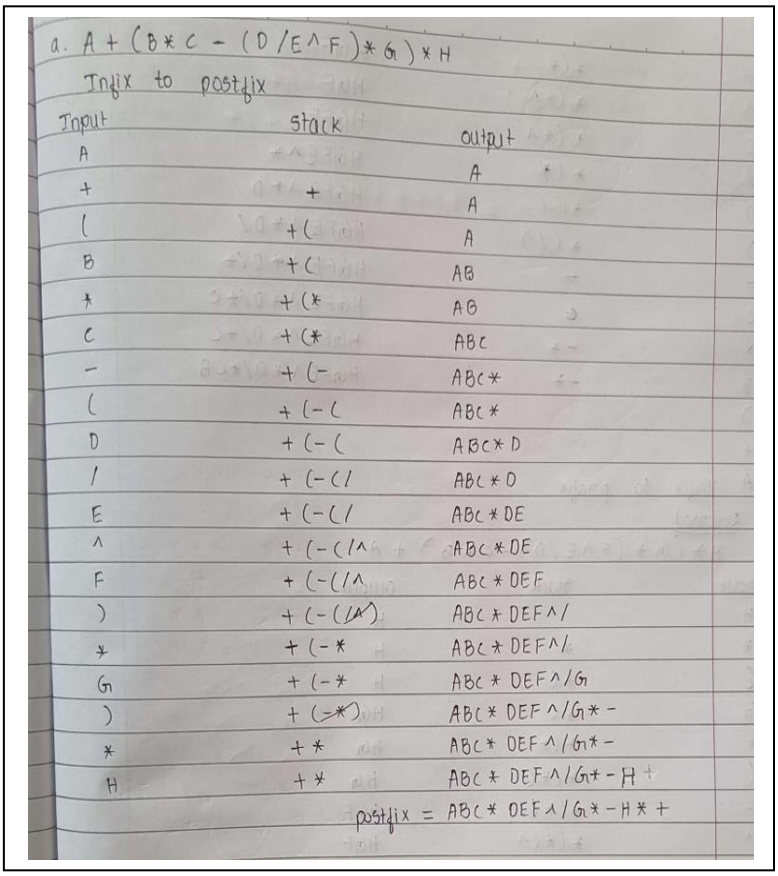
Step 2 - C ← A + B + 10

Step 3 - Stop
Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

PART III

5. **Convert the following infix expression to postfix and prefix expression. a. A+(B* C - (D/E^F)*G)*H b. (A+B^C)/D+E**

a) Infix to postfix : A+(B* C - (D/E^F)*G)*H

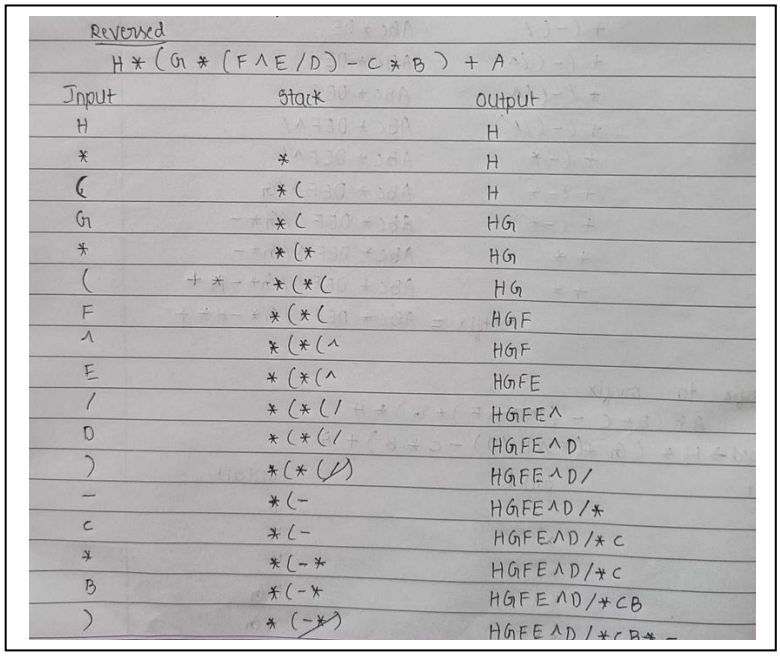

a. A + (B * C - (D / E ^ F) * G) * H

Infix to postfix

Input	stack	output
A		A
+	+	A
(	+(	A
B	+(	AB
*	+(*	AB
C	+(*	ABC
-	+(-	ABC*
(	+(-(	ABC*
D	+(-(	ABC*D
/	+(-(/	ABC*D
E	+(-(/	ABC*DE
^	+(-(/^	ABC*DE
F	+(-(/^	ABC*DEF
)	+(-(/^)	ABC*DEF^/
*	+(-*	ABC*DEF^/
G	+(-*	ABC*DEF^/G
)	+(-*)	ABC*DEF^/G*-
*	+*	ABC*DEF^/G*-
H	+*	ABC*DEF^/G*-H
		postfix = ABC*DEF^/G*-H*+

Ans: Infix to Postfix expression: A B C * D E F ^ / G * - H * +

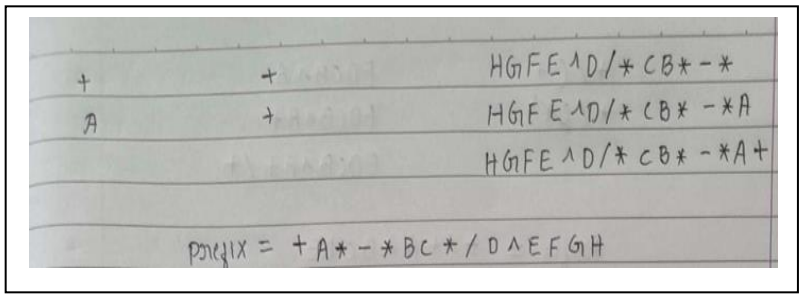
a) Infix to prefix : A+(B* C - (D/E^F)*G)*H



Reversed

H * (G * (F ^ E / D) - C * B) + A

Input	Stack	Output
H		H
*	*	H
(	* (	H
G	* (	HG
*	* (*	HG
(	* (* (	HG
F	* (* (	HGF
^	* (* (^	HGF
E	* (* (^	HGFE
/	* (* (/	HGFE^
D	* (* (/	HGFE^D
)	* (* (/)	HGFE^D/
-	* (-	HGFE^D/*
C	* (-	HGFE^D/*C
*	* (- *	HGFE^D/*C
B	* (- *	HGFE^D/*CB
)	* (- *)	HGFE^D/*CB*-

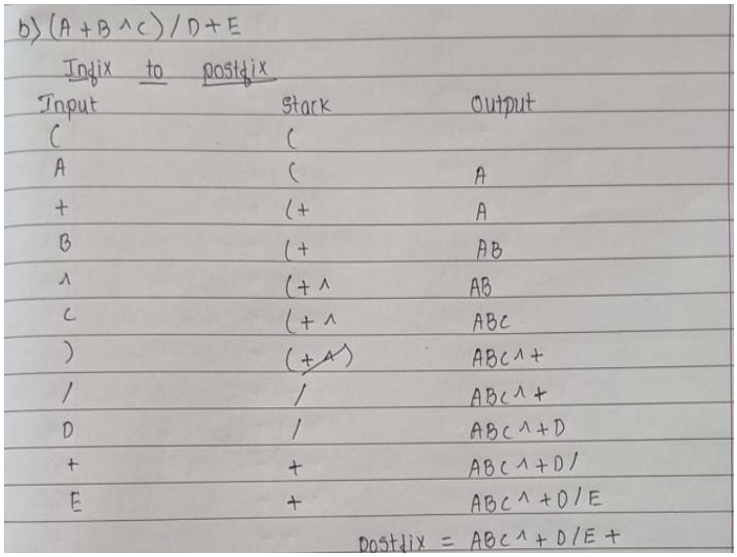


+	+	HGFE^D/*CB*-*
A	+	HGFE^D/*CB*-*A
		HGFE^D/*CB*-*A+

prefix = +A*-*BC*/D^EFGH

Ans: Infix to Prefix expression: +A*-* B C * /D^ E F G H

b. Infix to postfix : (A+B^C)/D+E



b) (A+B^C)/D+E

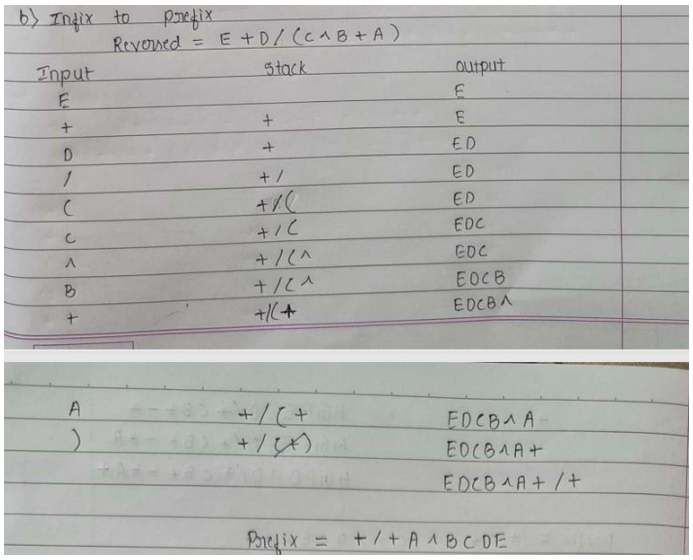
Infix to postfix

Input	Stack	Output
(	(	
A	(	A
+	(+	A
B	(+	AB
^	(+^	AB
C	(+^	ABC
)	(+^)	ABC^+
/	/	ABC^+
D	/	ABC^+D
+	+	ABC^+D/
E	+	ABC^+D/E

Postfix = ABC^+D/E+

Ans: Infix to Postfix expression: A B C^+ D /E +

b. Infix to Prefix : (A+B^C)/D+E



b) Infix to prefix

Reversed = E+D/(C^B+A)

Input	Stack	Output
E		E
+	+	E
D	+	ED
/	+/	ED
(	+/(	ED
C	+/C	EDC
^	+/C^	EDC
B	+/C^	EDCB
+	+/(+	EDCB^
A	+/(+	EDCB^A
)	+/(+)	EDCB^A+
		EDCB^A+/+

Prefix = +/+A^BCDE

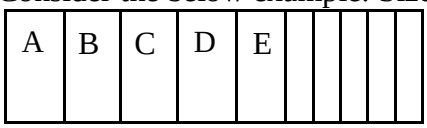
Ans: Infix to Prefix expression: +/+A ^B CD E

6. **What is stack? Explain the stack operations PUSH and POP with suitable C functions.**

Stack is a linear data structure which follows a particular order in which the operations are performed. In stack, insertion and deletion of elements happen only at one end, i.e., the most recently inserted element is the one deleted first from the set.
Few real world examples are given as:
Stack of plates in a buffet table. The plate inserted at last will be the first one to be removed out of stack. Stack of Compact Discs

Push Operation : The push operation is used to insert an element into the stack. The element is added always at the topmost position of stack. Since the size of array is fixed at the time of declaration, before inserting the value, check if top=Size-1, if so stack is full and no more insertion is possible. In case of an attempt to insert a value in full stack, an OVERFLOW message gets displayed.

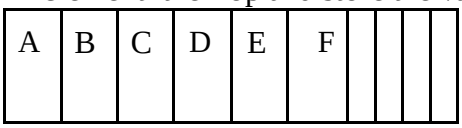
Consider the below example: Size=10



0 1 2 3 Top=4 5 6 7 8 9

Fig.3.2.3.2 Array Representation of Stack, Push(E)

To insert a new element F, first check if Top==Size-1. If the condition fails, then increment the Top and store the value. Thus the updated stack is:



0 1 2 3 4 Top=5 6 7 8 9

Fig.3.2.3.3 Array Representation of Stack, Push(F)

Algorithm:

Push(X)

```
if Top == Size-1
    Write "Stack Overflow"
```

else

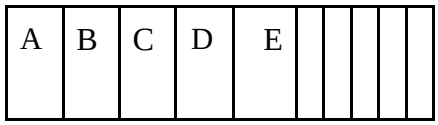
End

Pop Operation

return Top = Top + 1 // St represents an Array with maximum limit as Size St[Top] = X // X element to be inserted .

The pop operation is used to remove the topmost element from the stack. In this case, first check the presence of element, if top == -1 (indicates no array elements), then it indicates empty stack and thereby deletion not possible. In case of an attempt to delete a value in an empty stack, an UNDERFLOW message gets displayed.

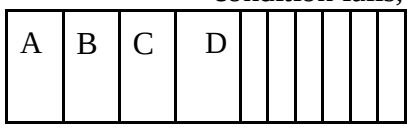
Consider the below example: Size=10



0 1 2 3 Top=4 5 6 7 8 9

Fig.3.2.3.4 Array Representation of Stack, Pop()

To delete the topmost element, first check if Top== -1. If the condition fails, then decrement the Top.Thus the updated stack is:



0 1 2 Top=3 4 5 6 7 8 9

Fig.3.2.3.5 Array Representation of Stack, Pop()

Algorithm:

Pop()

if Top == -1

Write “Stack Underflow”

return

else

X = St[Top] // St represents an Array with maximum limit as Size

Top = Top-1 // X represents element removed

return X

End

Program code:

```
#include <stdio.h>
#define SIZE 100

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

void push(int value) {
   if (top == SIZE - 1)
      printf("Stack Overflow!\n");
   else {
      top++;
      stack[top] = value;
      printf("Pushed %d\n", value);
   }
}

void pop() {
   if (top == -1)
      printf("Stack Underflow!\n");
   else {
      printf("Popped %d\n", stack[top]);
      top--;
   }
}

void display() {
   if (top == -1)
      printf("Stack is empty\n");
   else {
      printf("Stack: ");
      for (int i = top; i >= 0; i--)
         printf("%d ", stack[i]);
      printf("\n");
   }
}

int main() {
   int choice, value;
   while (1) {
      printf("\n1. PUSH\n2. POP\n3. DISPLAY\n4. EXIT\nEnter your choice: ");
      scanf("%d", &choice);
      switch (choice) {
         case 1:
            printf("Enter value to push: ");
```

```
            scanf("%d", &value);
            push(value);
            break;
        case 2:
            pop();
            break;
        case 3:
            display();
            break;
        case 4:
            return 0;
        default:
            printf("Invalid choice\n");
    }
  }
}
```

Sample output:

```
1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter value to push: 10
Pushed 10

1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter value to push: 20
Pushed 20

1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 1
Enter value to push: 30
Pushed 30

1. PUSH
2. POP
3. DISPLAY
4. EXIT
```

Enter your choice: 2
Popped 30

1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 3
Stack: 20 10

1. PUSH
2. POP
3. DISPLAY
4. EXIT
Enter your choice: 4

=== Code Execution Successful ===

PART IV

7. **What is a queue? Write a C Program to implement Enqueue, Dequeue and display using array as a data structure**

Queue is a linear data structure where elements are ordered in special fashion i.e. FIFO (First In First Out). Which means element inserted first to the queue will be removed first from the queue.
In real life you have come across various queue examples. Such as queue of persons at ticket counter, where the first person entering queue gets ticket first.
There are two basic operations that we generally perform on queue.

- Enqueue (Insertion)
- Dequeue (Removal)

Enqueue is the process of inserting an element to queue. In queue elements are always inserted at rear of queue.

Step by step descriptive logic to enqueue an element to queue.

Verify queue overflow and throw error if queue overflow happens, otherwise move to next step. You can use if (size >= CAPACITY) to verify queue overflow.
Increment rear size by 1. Note, that the increment should not cross array index bounds.
Which means if suppose queue capacity is 100 and size is 10 and rear is at 99 which means front will be at 89. Now when you enqueue a new element to queue, rear must get updated to 0 instead of 100. Otherwise array index will go beyond its bounds.

To do so we use rear = (rear + 1) % CAPACITY;.

Increment size of the queue by 1.
Insert new element at the rear of queue i.e. queue[rear] = data;.

Dequeue is the process of removing an element from queue. Elements from queue are always removed from front of queue.

Step by step descriptive logic to dequeue element from queue.

Check queue underflow and throw error if queue is empty, otherwise move to next step. You can use size to check empty queue i.e. if (size <= 0).
Copy current element at front of queue to some temporary variable. Say data = queue[front];
Increment front by 1. Similar to enqueue, the increment should not cross array index bounds.
Which means if suppose queue capacity is 100 and size is 10 and rear is at 9 which means front will be at 99. Now when you dequeue element from queue, front must get updated to 0 instead of 100. Otherwise array index will go beyond its bounds.

To do so we use front = (front + 1) % CAPACITY;.

Decrement queue size by 1.
data is required element dequeued from queue.

Program code:

```
#include <stdio.h>
#define SIZE 100

int queue[SIZE];
int front = -1, rear = -1;

// Function to add element to the queue
void enqueue(int value) {
    if (rear == SIZE - 1) {
        printf("Queue Overflow! Cannot insert %d\n", value);
    } else {
        if (front == -1)
            front = 0;  // First element being inserted
        rear++;
        queue[rear] = value;
        printf("Enqueued %d\n", value);
    }
}

// Function to remove element from the queue
void dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue Underflow! Cannot dequeue\n");
    } else {
        printf("Dequeued %d\n", queue[front]);
        front++;
    }
}

// Function to display the queue
void display() {
```

```
    if (front == -1 || front > rear) {
        printf("Queue is empty\n");
    } else {
        printf("Queue elements: ");
        for (int i = front; i <= rear; i++) {
            printf("%d ", queue[i]);
        }
        printf("\n");
    }
}

// Main function with menu
int main() {
    int choice, value;

    while (1) {
        printf("\nQueue Operations Menu:\n");
        printf("1. Enqueue\n2. Dequeue\n3. Display\n4. 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:
                dequeue();
                break;

            case 3:
                display();
                break;

            case 4:
                return 0;

            default:
                printf("Invalid choice! Try again.\n");
        }
    }

    return 0;
}
```

8. **Write an algorithm to evaluate postfix expression using stack. Trace the algorithm for the following expressions showing the stack contents 6 5 1 -4 2 3* ^ /+**

Step 1: Start
Step 2: Check all symbols from left to right and repeat steps 3 & 4 for each symbol of expression 'E' until all symbols are over.
(i) If the symbol is an operand, push it onto a stack.
(ii) (ii) If the symbol is an operator then:
(iii) (iii) Pop the top two operands from the stack and apply an operator in between them.
(iv) (iv) Evaluate the expression and place the result back on the stack.
Step 3: Set result equal to a top element on the stack.
Step 4: Stop

Trace the algorithm for the following expressions showing the stack contents 6 5 1 -4 2 3* ^ /+

Operation	**Expression**	**Stack**
	6 5 1 -4 2 3* ^ /+	Empty
Push	**5 1 -4 2 3*** ^ /+	6
Push	**1 -4 2 3*** ^ /+	6 5
Push	**-4 2 3*** ^ /+	6 5 1
pop	**4 2 3*** ^ /+	6 (5-1)
push	**2 3*** ^ /+	6 4 4
push	**3*** ^ /+	6 4 4 2 3
pop	^ /+	6 4 4 (2*3)
pop	/+	6 4 4 ^ 6
pop	+	6 4 / 4096
pop		6 0.0009765625 +

Ans: 6. 0009765625

PART V

9. **Develop a C recursive Program for Tower of Hanoi Problem.**

Tower of Hanoi is one of the main applications of recursion.

It says if you can solve n-1 cases, then you can solve the nth case.

It is also called as the Tower of Brahma or Lucas Tower.

It is a mathematical puzzle having applications in computer algorithms and programs as well as being used in psychology and medicine field as well.

Applications of Tower of Hanoi:

It has been used to determine the extent of brain injuries and helps to build/rebuild neural pathways in the brain as attempting to solve, Tower of Hanoi uses parts of the brain that involve managing time, foresight of whether the next move will lead us closer to the solution or not.

Psychologists use this to identify problem-solving skills of the subjects, which gives them insight as to how their minds are functioning and making decisions etc.

After so many interesting applications, aren't you curious to know what is this problem and why is it so widely prevalent?
Without much ado, let's dive into the problem and how to implement the same using C!

Problem Statement:
Given 3 towers/pegs/poles and n disks of decreasing sizes, move all the disks from one pole to another one without ever putting a larger disk on the smaller one.

The initial state of the puzzle is, when all the disks in the ascending order that is the smallest disk being on top is stacked on the first pole.

Rules of the puzzle:

- The following rules are to be followed while solving this problem which makes it more challenging and interesting.
- Only one disk can be moved at a time.
- A larger disk cannot be placed on a smaller disk.
- Each move involves, taking the disk at the top and placing it over another stack or empty pole.

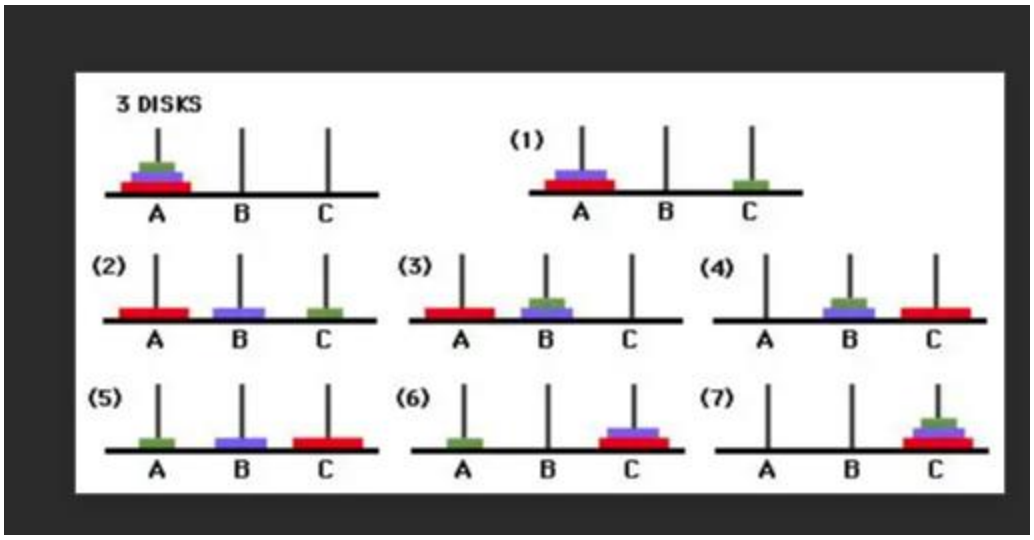
Note:
If you have n disks, the minimum number of moves required to solve it is $2n - 1$. If $n = 3$, you will require 7 moves minimum.

Understanding the problem and solution:

In our case, let us take $n = 3$. Let us name the poles serially as A, B, C with A being the source pole and C being the destination. B will be used as a spare pole.

As seen above, we have moved the 3 disks following all the rules in the minimum number of steps for $n = 3$ which is 7.
The recursive task is to keep moving the disks from one pole to another pole.
Let us see how we shall arrive at our base case and the recursive case for our program.
As stated initially, our objective is to move all the disks placed on pole A – source pole to pole C – destination pole.

To transfer all disks from A to C, we would have to move the two upper disks i.e n-1 disks, from A to B which is the spare pole.
Now, once n-1 disks are placed on spare peg we can now move the largest/last i.e nth disk onto the destination pole C.
The last set of the task would involve moving disks from spare peg B to destination peg C.
Thus, our cases have been derived and they are as given below:



Pseudocode

Base case:
if n == 1:
Move the disk from A to C using B as spare.

Recursive case:
Move n - 1 disks from A to B using C as spare.
Move the nth disk from A to C using B as spare.
Move the n - 1 disks from B to C using A as spare.

Program code:

```
#include <stdio.h>

// Recursive function to solve Tower of Hanoi
void towerOfHanoi(int n, char source, char auxiliary, char destination) {
    if (n == 1) {
        printf("Move disk 1 from %c to %c\n", source, destination);
        return;
    }
    // Move n-1 disks from source to auxiliary
    towerOfHanoi(n - 1, source, destination, auxiliary);

    // Move nth disk from source to destination
    printf("Move disk %d from %c to %c\n", n, source, destination);

    // Move n-1 disks from auxiliary to destination
    towerOfHanoi(n - 1, auxiliary, source, destination);
}

int main() {
    int n;
    printf("Enter number of disks: ");
    scanf("%d", &n);
```

```
    printf("\nSteps to solve Tower of Hanoi:\n");
    towerOfHanoi(n, 'A', 'B', 'C');  // A = source, B = auxiliary, C = destination

    return 0;
}
```

Sample output:

```
Enter number of disks: 3

Steps to solve Tower of Hanoi:
Move disk 1 from A to C
Move disk 2 from A to B
Move disk 1 from C to B
Move disk 3 from A to C
Move disk 1 from B to A
Move disk 2 from B to C
Move disk 1 from A to C


=== Code Execution Successful ===
```

10. Write an algorithm and C program to solve the 4 queen's problem.

The Working of an Algorithm to solve the 4-Queens Problem

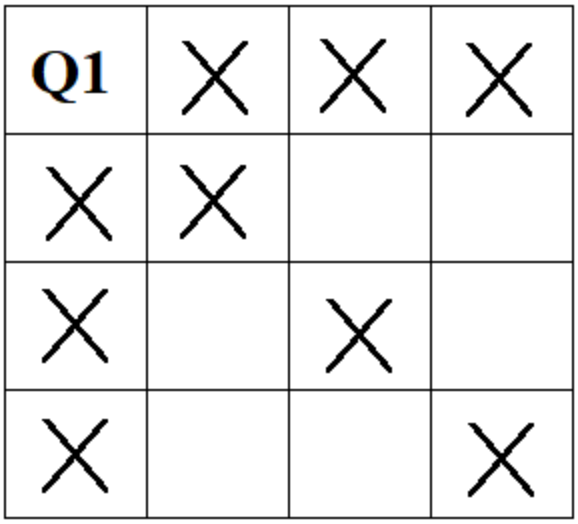
- To solve the problem place a queen in a position and try to find out all the possibilities for any queen being under attack or not.
- Based on the findings place only one queen in each row/column.
- If we found that the queen is under attack at its chosen position, then try for the next position.
- If a queen is under attack at all the positions in a row, we backtrack and change the position of the queen placed prior to the current position.
- Repeat this process of placing a queen and backtracking until all the N queens are placed successfully.

Step 1 -

As this is the 4-Queens problem, therefore, create a 4×4 chessboard.

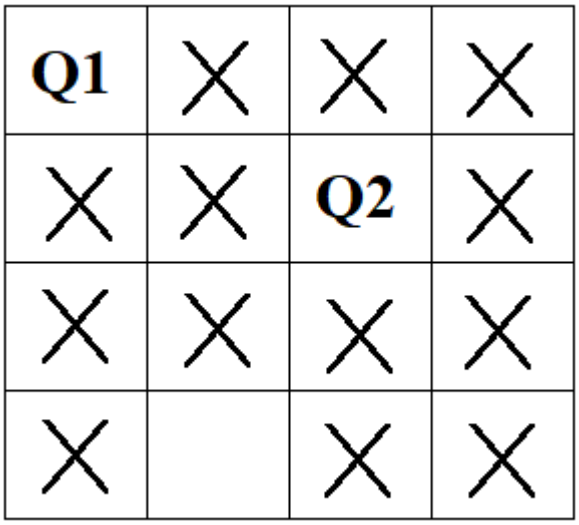
Step 2 -

- Place the Queen Q1 at the left-most position which means row 1 and column 1.
- Mark the cells of the chessboard with cross marks that are under attack from a queen Q1. (horizontal, vertical, and diagonal move of the queen)
- The chessboard looks as follows after placing **Q1 at [1, 1] position:**



Step 3 -

- The possible safe cells for Queen Q2 at row 2 are of column3 and 4 because these cells do not come under the attack from a queen Q1.
- So, here we place Q2 at the first possible safe cell which is row 2 and column 3.
- Mark the cells of the chessboard with cross marks that are under attack from a queen Q2. (horizontal, vertical, and diagonal move of the queen)
- The chessboard looks as follows after placing **Q2 at [2, 3] position:**

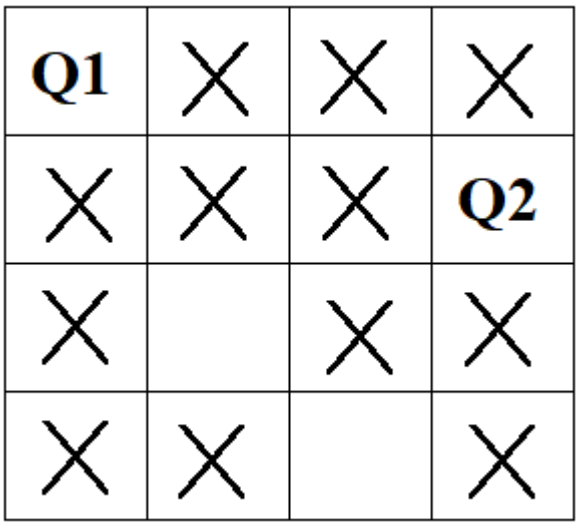


Step 4 -

- We see that no safe place is remaining for the Queen Q3 if we place Q2 at position [2, 3]
- Therefore make position [2, 3] false and backtrack.

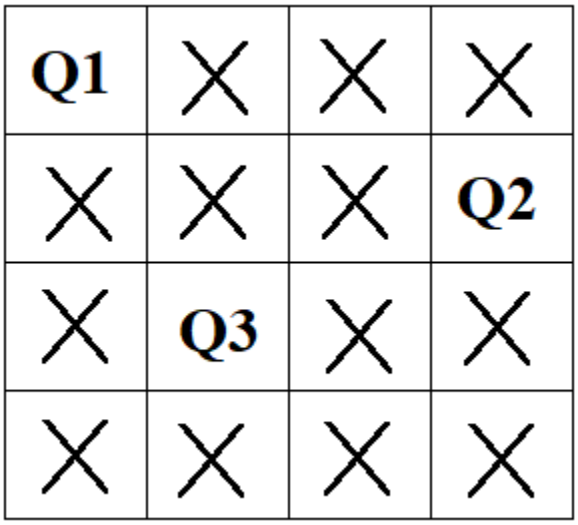
Step 5 -

- So, we place Q2 at the second possible safe cell which is row 2 and column 4.
- Mark the cells of the chessboard with cross marks that are under attack from a queen Q2. (horizontal, vertical, and diagonal move of the queen)
- The chessboard looks as follows after placing **Q2 at [2, 4] position:**



Step 6 -

- The only possible safe cell for Queen Q3 is remaining in row 3 and column 2.
- Therefore, we place Q3 at the only possible safe cell which is row 3 and column 2.
- Mark the cells of the chessboard with cross marks that are under attack from a queen Q3. (horizontal, vertical, and diagonal move of the queen)
- The chessboard looks as follows after placing **Q3 at [3, 2] position:**

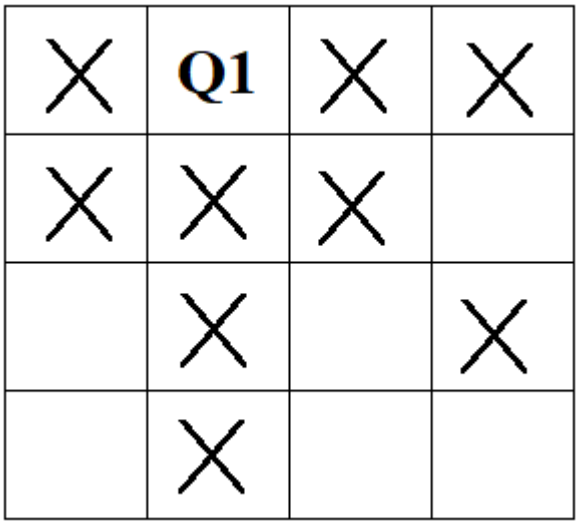


Step 7 -

- We see that no safe place is remaining for the Queen Q4 if we place Q3 at position [3, 2]
- Therefore make position [3, 2] false and backtrack.

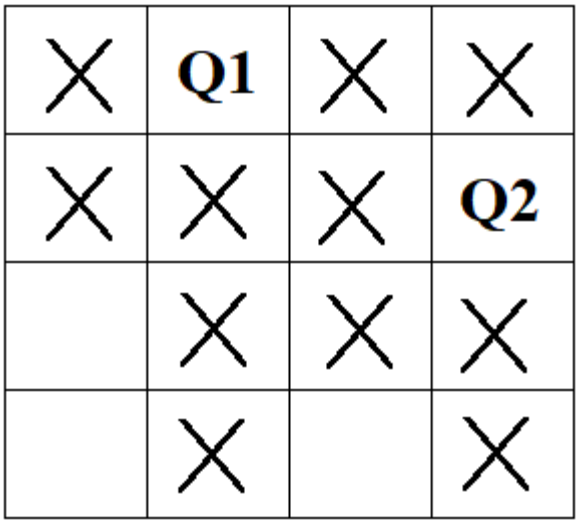
Step 8 -

- This time we backtrack to the first queen Q1.
- Place the Queen Q1 at column 2 of row 1.
- Mark the cells of the chessboard with cross marks that are under attack from a queen Q1. (horizontal, vertical, and diagonal move of the queen)
- The chessboard looks as follows after placing **Q1 at [1, 2] position:**



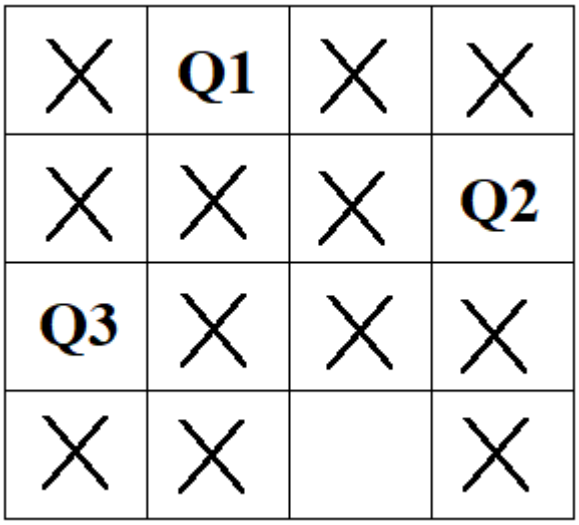
Step 9 -

- The only possible safe cell for Queen Q2 is remaining in row 2 and column 4.
- Therefore, we place Q2 at the only possible safe cell which is row 2 and column 4.
- Mark the cells of the chessboard with cross marks that are under attack from a queen Q2. (horizontal, vertical, and diagonal move of the queen)
- The chessboard looks as follows after placing **Q2 at [2, 4] position:**



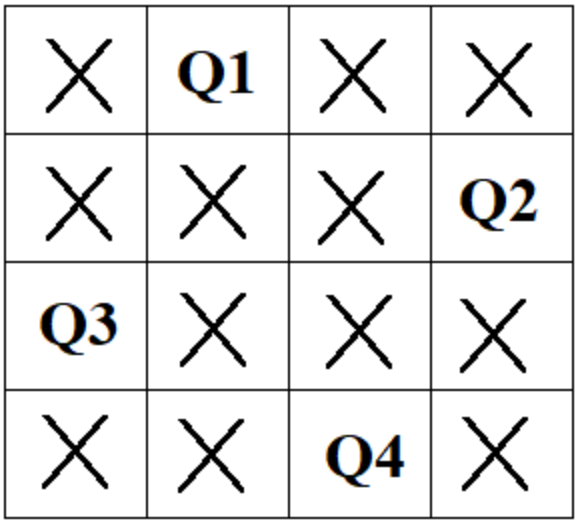
Step 10 -

- The only possible safe cell for Queen Q3 is remaining in row 3 and column 1.
- Therefore, we place Q3 at the only possible safe cell which is row 3 and column 1.
- Mark the cells of the chessboard with cross marks that are under attack from a queen Q3. (horizontal, vertical, and diagonal move of the queen)
- The chessboard looks as follows after placing **Q3 at [3, 1] position:**



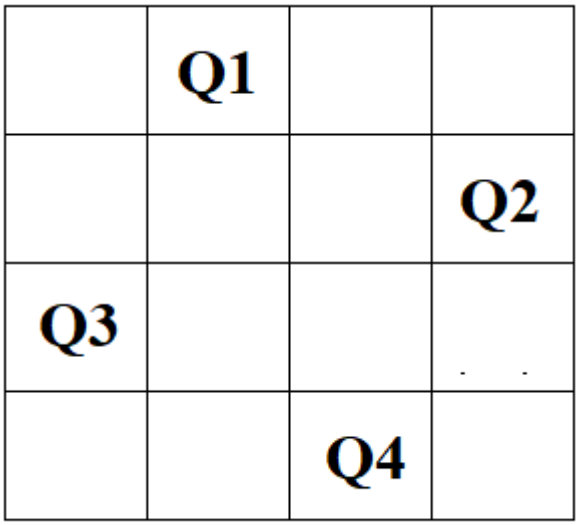
Step 11 -

- The only possible safe cell for Queen Q4 is remaining in row 4 and column 3.
- Therefore, we place Q4 at the only possible safe cell which is row 4 and column 3.
- The chessboard looks as follows after placing Q3 at [4, 3] position:

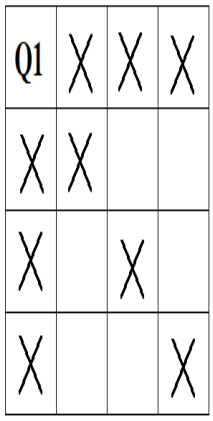


Step 12 -

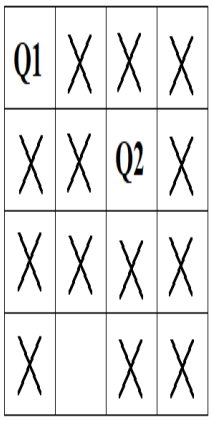
- Now, here we got the solution for the 4-queens problem because all 4 queens are placed exactly in each row/column individually.



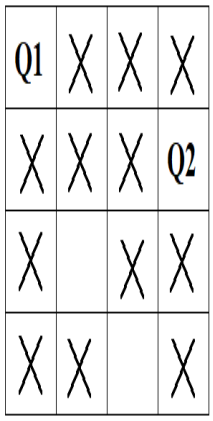
All these procedures are shown sequentially in the below figure:



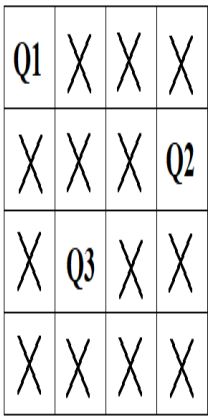
Step 1



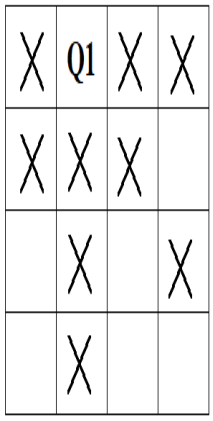
Step 2



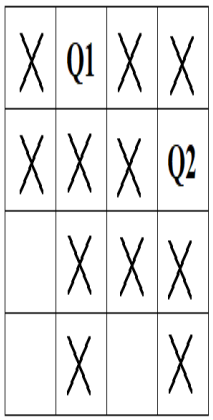
Step 3



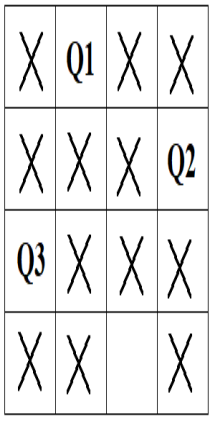
Step 4



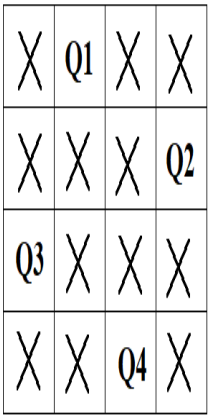
Step 5



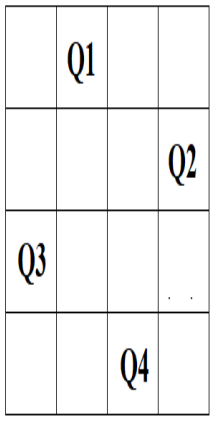
Step 6



Step 7

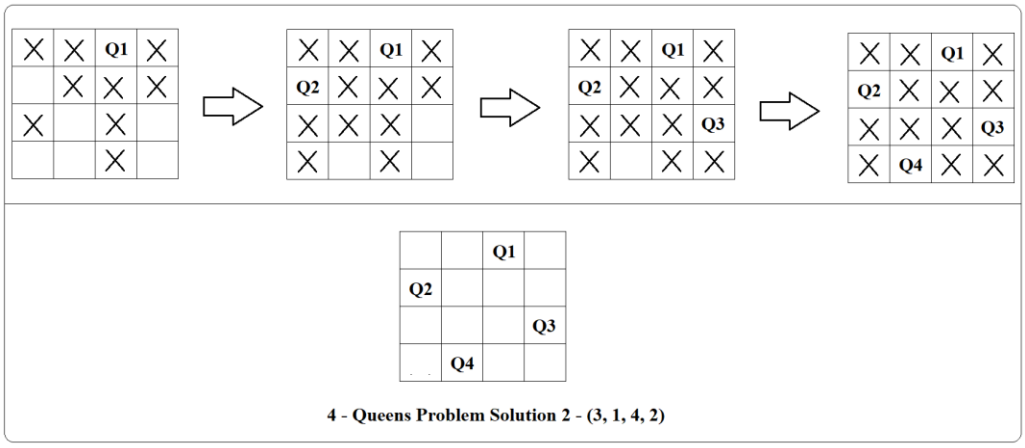


Step 8



4 Queens Problem Solution 1 - (2, 4, 1, 3)

- This is not the only possible solution to the problem.
- If we move each queen one step forward in a clockwise manner, we get another solution.
- This is also shown sequentially in the below figure:



4 - Queens Problem Solution 2 - (3, 1, 4, 2)

- In this example we placed the queens according to rows, we can do the same thing column-wise also.
- In that case, each queen will belong to a column.

Program code:

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

// Function to check if the given positions are a valid solution
int isValid(int q[]) {
    for (int i = 0; i < 4; i++) {
        for (int j = i + 1; j < 4; j++) {
            // Same column or same diagonal check
            if (q[i] == q[j] || abs(q[i] - q[j]) == abs(i - j)) {
                return 0;  // Conflict found
            }
        }
    }
    return 1;  // No conflict
}

// Function to display the chessboard
void displayBoard(int q[]) {
    printf("\n4-Queens Board Representation:\n\n");
    for (int i = 0; i < 4; i++) {
        for (int j = 1; j <= 4; j++) {
            if (q[i] == j)
                printf(" Q ");
```

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

int main() {
    int q[4];

    printf("Enter queen positions (q1 q2 q3 q4) as column numbers for rows 1 to 4 (1-4):\n");
    for (int i = 0; i < 4; i++) {
        printf("Row %d, Column: ", i + 1);
        scanf("%d", &q[i]);
        if (q[i] < 1 || q[i] > 4) {
            printf("Invalid input! Columns should be between 1 and 4.\n");
            return 1;
        }
    }

    if (isValid(q)) {
        printf("\nValid 4-Queens configuration.\n");
        displayBoard(q);
    } else {
        printf("\nInvalid 4-Queens configuration. Queens are attacking each other.\n");
    }

    return 0;
}
```

Sample output:

```
Enter queen positions (q1 q2 q3 q4) as column numbers for rows 1 to 4 (1-4):
Row 1, Column: 2
Row 2, Column: 4
Row 3, Column: 1
Row 4, Column: 3

Valid 4-Queens configuration.

4-Queens Board Representation:

 .  Q  .  .
 .  .  .  Q
 Q  .  .  .
 .  .  Q  .
=== Code Execution Successful ===
```