
Explanation of (implementation of static stack using array with push() & display() function.)

📌 What the program does?

It implements a stack using an array with two operations:

1. push() → Add element to stack
2. display() → Print all elements of stack
   A stack works on LIFO → Last In, First Out.

🔍 **Code Explanation (Step-by-Step)**

1️⃣ **Header and Definitions**

```
#include <stdio.h>
#define MAX 5
int stack[MAX];
int top = -1;
```

✓ #include <stdio.h>

Allows input/output functions (printf, scanf).

✓ #define MAX 5

Stack can hold maximum 5 elements.

✓ int stack[MAX];

Creates an integer array of 5 elements → this is the stack.

✓ int top = -1;

top stores index of the top element. Initially, stack is empty → so top = -1.

2️⃣ **push() Function**

```
void push(int val) {
if (top == MAX - 1)
printf("Stack Overflow!\n");
else {
top++;
stack[top] = val;
printf("%d pushed.\n", val);
}
}
```

✓ What does push do?

Adds an element to stack.

✓ Step-by-step:

Check Overflow: If top == MAX - 1, stack is full → cannot insert.

Otherwise:

Increase top by 1

Store value at new top position

Print confirmation message

Example: If you push 10, 20, 30 → stack becomes [10, 20, 30]

3️⃣ display() Function

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

✓ What does display do?

Prints all elements of the stack from TOP to BOTTOM.

✓ Why top to bottom?

Because stack is LIFO → last pushed element is printed first.

4️⃣ main() Function

```
int main() {
int choice, val;
do {
printf("\n1. Push 2. Display 3. Exit\nEnter choice: ");
scanf("%d", &choice);
```

User chooses an action.

Case 1 → Push

```
case 1:
printf("Enter value: ");
```

scanf("%d", &val);
push(val);
break;
User enters a value → program calls push().

Case 2 → Display
case 2:
display();
break;
Prints all elements.

Case 3 → Exit
case 3:
printf("Exiting...\n");
break;
Stops program.

Invalid Choice
default:
printf("Invalid choice!\n");

Loop Until Exit
} while (choice != 3);
Program keeps running until the user enters 3.

✅ Simplified Summary
Component | Purpose
stack[] | Stores stack elements
top | Points to latest pushed element
push() | Adds element (checks overflow)
display() | Prints stack top → bottom
main() | Menu-driven program

Q.2 program to sort an array using bubble sort.

📌 **What this program does?**
It sorts an array of numbers using Bubble Sort, a simple sorting technique where large elements "bubble" to the end by swapping.

🔍 Explanation (Step-by-Step like Program 1)

1️⃣ Variable declarations

int arr[100], n, i, j, temp;

arr[100] → Array to store numbers

n → Number of elements

i, j → Loop counters

temp → Used for swapping

2️⃣ Input section

scanf("%d", &n);

for (i = 0; i < n; i++)

scanf("%d", &arr[i]);

User enters:

how many numbers (n)

the actual numbers

3️⃣ Bubble sort logic

```
for (i = 0; i < n - 1; i++) {
for (j = 0; j < n - i - 1; j++) {
if (arr[j] > arr[j + 1]) {
temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
}
}
}
```

✓ Outer loop (i)

Runs the bubble process multiple times.

✓ Inner loop (j)

Compares each element with the next one.

✓ Swap condition

If arr[j] is greater than arr[j+1], swap them.

This moves the biggest element to the end first, then the next biggest, and so on.

4️⃣ Print sorted array

for (i = 0; i < n; i++)

printf("%d ", arr[i]);

Displays the sorted numbers.

⭐ Simple Summary
Bubble sort works by:
Repeatedly comparing adjacent numbers
Swapping if in wrong order
Largest element moves to the end each pass

Q. Postfix Expression Evaluation

📌 **What this program does?**

It evaluates a postfix expression (example: 23+ = 5) using a stack.

🔍 **Explanation (Step-by-Step)**

1️⃣ **Global variables**

int stack[MAX];
int top;
stack[] → used to store numbers during calculation
top → points to top of stack

2️⃣ **init()**

void init() {
top = -1;
}
Starts with an empty stack.

3️⃣ **push()**

stack[++top] = val;
Adds a number to the top.

4️⃣ **pop()**

return stack[top--];
Returns and removes the top number.

5️⃣ evaluate() — Main logic

for (i = 0; postfix[i] != '\0'; i++) {
ch = postfix[i];
Reads postfix expression character by character.

✓ If character is a digit

if (isdigit(ch))
push(ch - '0');

Example: '5' - '0' becomes 5
Push number to stack.
✓ If character is an operator
val2 = pop();
val1 = pop();
Take two numbers from stack.
✓ Perform calculation
switch (ch) {
case '+': result = val1 + val2; break;
case '-': result = val1 - val2; break;
case '*': result = val1 * val2; break;
case '/': result = val1 / val2; break;
}
Then push result back.
✓ After entire loop
return pop();
Final answer is the last element left in stack.
⭐ Simple Summary
Numbers → pushed
Operator → pops 2 numbers, calculates, pushes result
Last number in stack = final answer

Insertion Sort

📌 **What this program does?**

It sorts an array using Insertion Sort, where each element is inserted into its correct position in the sorted part of the array.

🔍 **Explanation (Step-by-Step)**

1️⃣ **Variable declarations**

int arr[100], n, i, j, key;

arr[] → array of numbers

n → size

i, j → loop counters

key → current number to be inserted

2️⃣ Input

scanf("%d", &n);

for (i = 0; i < n; i++)

scanf("%d", &arr[i]);

User gives:

number of elements

array elements

3️⃣ Insertion Sort Logic

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

Start from index 1

key is the value we want to position correctly

✓ Move larger values to the right

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

If previous values are bigger than key, shift them right.

✓ Insert the key at correct position

arr[j + 1] = key;

Now place key in the empty spot.

4️⃣ Print sorted array

for (i = 0; i < n; i++)

printf("%d ", arr[i]);

Displays sorted numbers.

⭐ Simple Summary

Insertion sort works like sorting playing cards in your hand: You pick a card and insert it at the correct spot in the sorted part.

Static Queue (isFull + insert)

📌 What this program does?

It creates a queue using an array and supports:

isFull()

insert()
display()
A queue works on FIFO → First In, First Out.

🔍 Explanation (Step-by-Step)

1️⃣ Variables

int queue[MAX];
int front = -1, rear = -1;
queue[] → stores elements
front → points to first element
rear → last element
-1, -1 → queue is empty

2️⃣ isFull()

return rear == MAX - 1;
If rear reaches last index → queue is full.

3️⃣ insert()

if (isFull()) {
printf("Queue Overflow!\n");
return;
}
Prevents inserting when queue is full.
✓ First element insertion
if (front == -1)
front = 0;
When queue is empty, set front to 0.
✓ Insert new element
rear++;
queue[rear] = val;
printf("%d inserted into queue.\n", val);
Increase rear
Add value
Print confirmation

4️⃣ display()

if (front == -1 || front > rear) {
printf("Queue is empty!\n");
return;

}
Checks if queue has no elements.
✓ Print queue
for (int i = front; i <= rear; i++)
printf("%d ", queue[i]);
Prints from front → rear.

5️⃣ main()
Provides menu for:

1. Insert
2. Display
3. Exit

⭐ Simple Summary
Queue stored in array
isFull checks if no space
insert adds element at rear
display prints in FIFO order

Static Stack (isEmpty + pop)
📌 What this program does?
Implements a stack using array with:
isEmpty()
pop()
push()
display()

🔍 Explanation (Step-by-Step)
1️⃣ Variables
int stack[MAX];
int top = -1;
stack[] → stores elements
top → index of topmost element

2 isEmpty()

return top == -1;

When top == -1, stack has no elements.

3 push()

```
if (top == MAX - 1)
printf("Stack Overflow!\n");
else {
top++;
stack[top] = val;
printf("%d pushed onto stack.\n", val);
}
```

Checks overflow

Increases top

Adds value

4 pop()

```
if (isEmpty()) {
printf("Stack Underflow!\n");
return -1;
}
else {
int val = stack[top];
top--;
return val;
}
```

Checks if empty

Removes top element

Returns popped value

5 display()

```
for (int i = top; i >= 0; i--)
printf("%d ", stack[i]);
```

Prints from top → bottom.

6 main()

Menu:

1. Push

2. Pop
3. Display
4. Exit

⭐ Simple Summary
isEmpty() checks if stack has no items
pop() removes top value
push() adds new value
display() prints stack in LIFO order

QUICK SORT

✓ WHAT IT DOES

Quick Sort sorts an array by choosing a pivot and dividing the array into:
elements smaller than pivot (left side)
elements greater than pivot (right side)
Then it recursively sorts both sides.

✓ HOW IT DOES IT (Step-by-step with code)

🔹 **swap() — HOW it works**

```
void swap(int *a, int *b) {
int temp = *a;
*a = *b;
*b = temp;
}
```

✓ HOW

Makes a temporary copy
Exchanges values of two variables
Quick Sort needs this to rearrange numbers around pivot

🔹 **partition() — HOW it divides the array**

```
int partition(int arr[], int low, int high) {
int pivot = arr[high];
int i = low - 1;
```

✓ WHAT

Selects the last element as pivot.

✓ HOW

i tracks position for small numbers

Loop checks each number:

If a number is < pivot, increase i and swap

```
for (int j = low; j < high; j++) {
if (arr[j] < pivot) {
i++;
swap(&arr[i], &arr[j]);
}
}
swap(&arr[i + 1], &arr[high]);
return i + 1;
```

✓ HOW it arranges

All small numbers go before pivot

All large numbers go after pivot

Returns pivot index

This index splits array into two halves.

- quickSort() — HOW it sorts

```
void quickSort(int arr[], int low, int high) {
if (low < high) {
int pi = partition(arr, low, high);
quickSort(arr, low, pi - 1);
quickSort(arr, pi + 1, high);
}
}
```

✓ HOW

Calls partition → gets pivot position

Recursively sorts left side

Recursively sorts right side

It keeps dividing until each part has 1 element (automatically sorted).

- main() — HOW it runs

Takes input

Calls quickSort

Prints sorted array

⭐ FINAL SUMMARY (Program 8)
Quick Sort repeatedly splits array around a pivot and sorts both sides.

Singly Linked List (create + display)
✓ WHAT IT DOES
Creates a linked list using malloc and prints all nodes.
✓ HOW IT DOES IT (Step-by-step)

🔹 **Node Structure**

```
struct Node {
int data;
struct Node *next;
};
```

✓ HOW
Every node stores:
a value (data)
address of next node (next)
This creates a chain-like structure.

🔹 create() — HOW it builds the linked list

```
void create(int n) {
struct Node *newNode, *temp;
```

✓ WHAT
Creates n nodes.
✓ HOW

1. malloc allocates new node
2. Store user data in node
3. Set newNode->next = NULL
4. If first node → head = newNode
5. Otherwise:
   Traverse to last node using a loop
   Attach new node to last node
   temp = head;
   while (temp->next != NULL)
   temp = temp->next;

temp->next = newNode;
This keeps adding nodes at the end.

🔹 **display() — HOW it prints list**

```
struct Node *temp = head;
while (temp != NULL) {
printf("%d -> ", temp->data);
temp = temp->next;
}
```

✓ HOW

Starts at head → walks through each node using .next until NULL. Prints data in each node.

🔹 **main() — HOW it operates**

User chooses option
create() builds list
display() shows list

⭐ FINAL SUMMARY (Program 9)

Linked list is built by connecting nodes using next pointers. display() moves through the chain and prints all values.

Dynamic Queue (insert + display)

✓ WHAT IT DOES

Implements a queue using linked list.

✓ HOW IT DOES IT

🔹 **Node Structure**

```
struct Node {
int data;
struct Node *next;
};
```

✓ HOW

Each node stores: value pointer to next node
This allows queue to grow dynamically with malloc.

- **front & rear pointers**

```
struct Node *front = NULL;
struct Node *rear = NULL;
```

✓ HOW

These pointers mark:

front → first element

rear → last element

Needed for queue operations.

- **insert() — HOW enqueue works**

```
newNode = (struct Node*) malloc(sizeof(struct Node));
newNode->data = value;
newNode->next = NULL;
```

✓ HOW

Creates a new node to add at the end.

✓ Case 1 — Queue empty

```
if (rear == NULL)
front = rear = newNode;
```

HOW: First element is both front & rear.

✓ Case 2 — Queue not empty

```
rear->next = newNode;
rear = newNode;
```

HOW:

Connect newNode after last node

Update rear pointer

- **display() — HOW it prints queue**

```
temp = front;
while (temp != NULL) {
printf("%d ", temp->data);
temp = temp->next;
}
```

✓ HOW

Starts at front and prints each node's value until NULL.

- **main() — HOW user interacts**

Insert values

Display queue
Exit

⭐ FINAL SUMMARY (Program 10)
Queue is created using linked list. insert() adds at rear. display() prints from front → rear.

LINEAR SEARCH
✓ WHAT IT DOES
Searches for a particular element in an array by checking each element one by one.
✓ HOW IT DOES IT (Step-by-Step with Code Blocks)

🔹 **1. Input Array**

```
scanf("%d", &n);
for (i = 0; i < n; i++) {
scanf("%d", &arr[i]);
}
```

👉 **HOW**
User enters number of elements (n)
Then enters n values into the array

🔹 **2. Input the element to search**

```
printf("Enter the element to search: ");
scanf("%d", &key);
```

👉 **HOW**
Stores the value to be searched in key.

🔹 **3. Linear Search Logic**

```
for (i = 0; i < n; i++) {
if (arr[i] == key) {
printf("Element %d found at position %d.\n", key, i + 1);
found = 1;
break;
}
}
```

👉 **HOW**
Loop goes from start to end

Compares arr[i] with key
If match found → print position and stop
This is called linear search because search happens in a straight line.

🔹 **4. If not found**

```
if (!found) {
printf("Element %d not found in the array.\n", key);
}
```

👉 **HOW**

If found is still 0, it means key was not present in array.

⭐ FINAL SUMMARY (Program 11)

Linear search moves through array from start to end until it finds the key.

DYNAMIC STACK USING LINKED LIST

✓ WHAT IT DOES

Implements a stack using linked list with:
push() → add element
display() → show stack
Stack works on LIFO → Last In, First Out.

✓ HOW IT DOES IT (Step-by-Step with Code)

🔹 **Node Structure**

```
struct Node {
int data;
struct Node* next;
};
```

👉 **HOW**

Each stack element is a node containing:
a value (data)
link to next node

🔹 **Global top pointer**

```
struct Node* top = NULL;
```

👉 **HOW**

top always points to the most recently added node.

🔹 push() — HOW it adds element

```
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
newNode->next = top;
top = newNode;
```

👉 HOW

1. malloc creates a new node
2. store value in node
3. newNode links to old top
4. update top to new node
   This way the new node becomes the top of the stack.

🔹 **display() — HOW it prints stack**

```
struct Node* temp = top;
while (temp != NULL) {
printf("%d ", temp->data);
temp = temp->next;
}
```

👉 **HOW**

Starts at top and prints each node until end. Shows stack from top to bottom.

🔹 **main()**

```
push(10);
push(20);
push(30);
display();
```

👉 **HOW**

Pushes 3 elements, then displays them.

⭐ FINAL SUMMARY (Program 12)

Stack is built using linked nodes where each push places a new node on top.

STATIC QUEUE (init + isEmpty)

✓ WHAT IT DOES

Implements a static queue with:
init() → initialize
isEmpty() → check if empty
✓ HOW IT DOES IT (Step-by-Step)

🔹 **Queue Variables**

```
int queue[SIZE];
int front, rear;
```

👉 **HOW**
queue[] stores values
front = index of the first element
rear = index of last element

🔹 **init() — HOW initialization works**

```
void init() {
front = -1;
rear = -1;
}
```

👉 **HOW**
Both front and rear set to -1 means queue is empty.

🔹 **isEmpty() — HOW it checks empty state**

```
return (front == -1 && rear == -1);
```

👉 **HOW**
If both front and rear are -1 → no elements present
This accurately checks whether queue is empty.

🔹 **main()**

```
init();
if (isEmpty())
printf("Queue is empty\n");
```

👉 **HOW**
First initialize queue
Then check and print whether it is empty

⭐ FINAL SUMMARY (Program 13)
Queue is empty initially because both front and rear = -1. isEmpty() verifies that condition.

DOUBLY LINKED LIST (create + display)

✓ WHAT IT DOES

Implements a doubly linked list with:

create() → inserts node at end

display() → prints the list

A doubly linked list has:

prev pointer

next pointer

Allowing movement in both directions.

✓ HOW IT DOES IT (With Code Blocks)

🔹 1. Node Structure

```
struct Node {
int data;
struct Node* prev;
struct Node* next;
};
```

👉 HOW

Each node stores:

data → value

prev → pointer to previous node

next → pointer to next node

This makes two-way movement possible.

🔹 2. Head Pointer

```
struct Node* head = NULL;
```

👉 HOW

head stores the address of the first node. NULL means list is empty.

🔹 3. create() — HOW nodes are inserted at end

```
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = value;
newNode->prev = NULL;
newNode->next = NULL;
```

👉 HOW

New node is created

Data is stored

Initially, prev and next are NULL

✓ Case 1 — Empty list

```
if (head == NULL) {
head = newNode;
}
```

👉 HOW

First node becomes the head.

✓ Case 2 — Insert at end of list

```
else {
struct Node* temp = head;
while (temp->next != NULL)
temp = temp->next;
temp->next = newNode;
newNode->prev = temp;
}
```

👉 HOW

Moves to last node

Connects new node to end

Updates both next and prev pointers

🔹 4. display() — HOW it prints the list

```
struct Node* temp = head;
printf("Doubly Linked List: ");
while (temp != NULL) {
printf("%d ", temp->data);
temp = temp->next;
}
```

👉 HOW

Starts at head and follows next pointer until end.

⭐ FINAL SUMMARY (Program 14)

Doubly linked list allows navigation forward and backward. Nodes are added at end using both next and prev pointers.

————————————————————————————————

STATIC QUEUE (isEmpty + insert)

✓ WHAT IT DOES

Implements a static queue using array with:

isEmpty() → checks if queue empty

insert() → adds element at rear
Queue works on FIFO → first element out first.
✓ HOW IT DOES IT (With Code Blocks)

🔹 1. Queue Variables

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

👉 HOW
front = index of first element
rear = index of last element
(-1, -1) means empty queue

🔹 2. isEmpty() — HOW it checks emptiness

```
return (front == -1 && rear == -1);
```

👉 HOW
If both pointers are -1, queue is empty.

🔹 3. insert() — HOW elements are added

```
if (rear == SIZE - 1) {
printf("Queue is Full\n");
return;
}
```

👉 HOW
Checks if rear reached maximum index → queue full.
✓ Case 1 — First element

```
if (isEmpty()) {
front = rear = 0;
}
```

👉 HOW
If queue was empty, set both pointers to 0.
✓ Case 2 — Normal insert

```
else {
rear++;
}
queue[rear] = value;
```

👉 HOW
Increase rear
Insert element at position rear

🔹 4. main()

Calls isEmpty and insert to test functionality.

⭐ FINAL SUMMARY (Program 15)

Queue stored in array. Insert adds at rear. isEmpty checks if front & rear are both -1.

————————————————————————————————

STATIC STACK (init + insert/push)

✓ WHAT IT DOES

Implements a stack using array with:

init() → initializes stack

insert() / push → adds element to top

Stack works on LIFO → Last In First Out.

✓ HOW IT DOES IT (With Code Blocks)

🔹 1. Stack Variables

```
int stack[SIZE];
int top;
```

👉 HOW

stack[] → holds elements

top → index of top element

🔹 2. init() — HOW stack is initialized

```
void init() {
top = -1;
}
```

👉 HOW

top = -1 means stack is empty.

🔹 3. insert() (push) — HOW element is added

```
if (top == SIZE - 1) {
printf("Stack is Full\n");
return;
}
```

👉 HOW

Checks if stack reached max size.

✓ Push element

```
stack[++top] = value;
```

printf("%d inserted\n", value);

👉 **HOW**

Increment top

Store new value at stack[top]

New element becomes the top

🔹 **4. main()**

Calls init()

Menu lets user push values

⭐ FINAL SUMMARY (Program 16)

Stack is stored in array. Insert/push adds element on top by increasing top.

Dynamic Queue (delete + display)

✓ WHAT IT DOES

This program implements a dynamic queue using a linked list and provides:

1. initQueue() → creates queue with predefined values
2. delete() → removes element from front (dequeue)
3. display() → prints entire queue
   Queue works on FIFO: First inserted → First removed

✓ HOW IT DOES IT (Step-by-Step with Code Blocks)

🔹 **1. Node Structure**

struct Node {

int data;

struct Node* next;

};

👉 **HOW**

Every queue element is stored inside a linked list node. Each node has:

data → value

next → pointer to next node

This allows queue to grow without size limit.

🔹 **2. Queue Pointers**

struct Node *front = NULL, *rear = NULL;

👉 HOW

front points to first element

rear points to last element

If both are NULL → queue is empty.

🔹 3. Predefined values

```
int values[] = {10, 20, 30, 40, 50};
int n = 5;
```

👉 HOW

Queue starts with 5 fixed values, and these will be added automatically.

🔹 4. initQueue() — HOW it builds initial queue

```
for (int i = 0; i < n; i++) {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
newNode->data = values[i];
newNode->next = NULL;
```

👉 HOW

Creates 5 nodes using malloc

Stores each value from the array

✓ Case 1 — queue empty

```
if (rear == NULL) {
front = rear = newNode;
}
```

👉 HOW

First node becomes both:

front

rear

✓ Case 2 — queue already has elements

```
else {
rear->next = newNode;
rear = newNode;
}
```

👉 HOW

Connect new node to end

Move rear to new node

This builds the queue like:

10 → 20 → 30 → 40 → 50

🔹 5. delete() — HOW element is removed from queue

```
if (front == NULL) {
printf("Queue is empty\n");
return;
}
```

👉 HOW

Checks if queue is empty before deleting.

✓ Remove front node

```
struct Node* temp = front;
printf("%d deleted\n", temp->data);
front = front->next;
```

👉 HOW deletion works

1. temp holds current front
2. Print its value
3. Move front pointer to next node
4. Free removed node

   ✓ If queue becomes empty after deletion

   ```
   if (front == NULL)
   rear = NULL;
   ```

   👉 HOW

   If last element is removed, both pointers must be NULL to indicate empty queue.

🔹 **6. display() — HOW it prints queue**

```
struct Node* temp = front;
printf("Queue: ");
while (temp != NULL) {
printf("%d ", temp->data);
temp = temp->next;
}
```

👉 **HOW**

Starts at front, prints each node until NULL.

🔹 **7. main() — HOW user interacts**

```
initQueue();
while (1) {
printf("1. Delete 2. Display 3. Exit");
```

👉 **HOW**

Menu-driven:

Delete → call delete()

Display → call display()

Exit program

⭐ FINAL SUMMARY

Dynamic queue using linked list:

initQueue() builds initial queue

delete() removes from front (FIFO)

display() shows all elements
